Zaid Mohammad Rafique Patel, 2026, 14:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

DocuMate - An Intelligent Framework for Automated, Context-Aware Documentation in Version-Controlled Software Development

Zaid Mohammad Rafique Patel[1], Amir Aneesh Khan[2], Dr. Jasbir Kaur[3], Ifrah Kampoo[4], Mansi Rajapurkar[5]

[1,2]Student of Master of Computer Applications (MCA), Guru Nanak Institute of Management Studies, Matunga, Mumbai, India

[3]Director, GNIMS B-School, Head of Information Technology and HR, Guru Nanak Institute of Management Studies, Matunga, Mumbai, India

[4,5]Assistant Professor, Guru Nanak Institute of Management Studies, Matunga, Mumbai, India,

Abstract- The rapid evolution of agile software devel-opment practices has generated increasingly complex codebases, creating a persistent gap between active source code and its corresponding technical documen-tation. This research investigates the application of generative artificial intelligence directly within ver-sion control workflows, examining methodologies for automating the extraction, generation, and continu-ous maintenance of context-aware project documenta-tion. Through the development of DocuMate—an event-driven framework integrating Large Language Mod-els (LLMs), structural project analysis, and native Git hooks—this study evaluates the effectiveness of auto-mated configuration management for critical project files, including READMEs, Dockerfiles, and environ-ment templates. Our implementation demonstrates that combining local codebase heuristics with dynamically prompted LLM generation achieves highly accurate, repository-specific documentation, significantly reduc-ing manual developer overhead compared to traditional static templating. Furthermore, the research addresses critical challenges including "documentation drift," context preservation, and deployment safety through a human-in-the-loop "pending-to-approved" state mecha-nism. The implications of this work extend to technical debt reduction, open-source maintainability, automated infrastructure configuration, and streamlined devel-oper onboarding processes.

Keywords: Automated Documentation, Large Language Models (LLMs), Version Control Systems, Generative AI, Software Engineering, Configuration Management.

I. INTRODUCTION

The rapid evolution of modern software engineer-ing practices has fundamentally altered how complex applications are architected and delivered. With the widespread adoption of agile methodologies and con-tinuous integration pipelines, version-controlled repos-itories experience constant, rapid modifications. This accelerated pace of code generation presents both opportunities for rapid innovation and severe chal-lenges for long-term project maintainability. Automated documentation and configuration management repre-sent a critical intersection of software engineering, generative artificial intelligence, and version control systems. This domain enables the programmatic ex-traction of deep project context and the automated synthesis of essential repository assets, such as ar-chitectural READMEs, Dockerfiles, and environment variable templates. When applied directly within the development lifecycle, automated documentation pro-vides invaluable benefits for developer productivity, onboarding efficiency, and the reduction of technical debt.

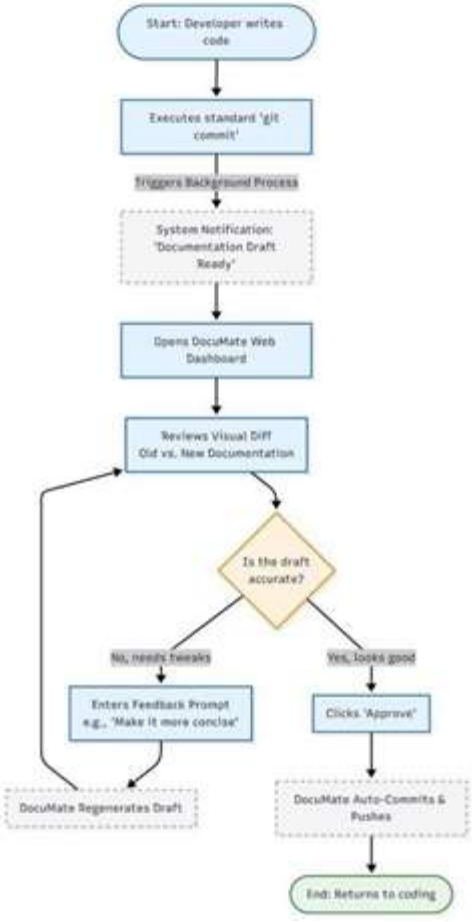
The significance of this research lies in address-ing the persistent phenomenon known as "documenta-tion drift"—the computational and workflow challenge where the rapid evolution of source code consistently outpaces the manual updates of its corresponding documentation. Traditional, manual documentation ap-proaches, which are often treated as an afterthought in the sprint cycle, inevitably fail to maintain accuracy and relevance, resulting in broken configurations and obfuscated codebases.

This study contributes to the field by proposing DocuMate, an intelligent, event-driven framework that leverages Large Language Models (LLMs) and local codebase heuristics integrated na-tively into Git workflows. By automating context-aware file generation alongside strict "human-in-the-loop" ap-proval mechanisms, this research presents a highly optimized methodology for maintaining synchronized, high-quality documentation in agile software develop-ment environments. Furthermore, the increasing com-plexity of modern software ecosystems has amplified the need for intelligent tooling that can bridge the gap between source code and project documentation.

Con-temporary applications often rely on diverse technol-ogy stacks, containerized deployment environments, cloud-native infrastructures, and extensive third-party dependencies. Maintaining accurate documentation for such systems requires continuous monitoring of repos-itory changes and a deep understanding of project structure. As development teams grow and collaborate across distributed environments, the absence of up-to-date documentation can significantly hinder knowledge transfer, increase onboarding time for new contrib-utors, and introduce avoidable configuration errors during deployment and maintenance activities.

Recent advancements in Large Language Models (LLMs) have opened new possibilities for automat-ing software engineering tasks that traditionally re-quired substantial human effort. Unlike conventional template-based documentation generators, LLMs pos-sess the capability to analyze source code semantics, infer architectural patterns, and generate contextually relevant explanations in natural language. These ca-pabilities enable the creation of documentation that is not only technically accurate but also accessible to developers with varying levels of expertise. However, integrating generative AI into software development workflows introduces challenges related to reliabil-ity, consistency, security, and user trust, necessitating mechanisms that ensure human oversight and valida-tion of AI-generated outputs.

Another important consideration is the role of ver-sion control systems as a rich source of contextual information. Every commit, branch, and repository modification contains valuable insights regarding the evolution of a software project. By leveraging repos-itory metadata alongside source code analysis, intel-ligent documentation systems can better understand the intent behind changes and generate artifacts that accurately reflect the current state of the applica-tion. This integration transforms documentation from a static project artifact into a dynamic, continuously evolving resource that remains synchronized with the underlying codebase.



The proposed DocuMate framework seeks to ad-dress these challenges by combining repository analysis, event-driven automation, and advanced language model capabilities into a unified solution. Rather than replacing developers, the system is designed to aug-ment their workflow by reducing repetitive documen-tation tasks while preserving human control over crit-ical decisions. Through seamless integration with Git-based development

practices, DocuMate promotes doc-umentation consistency, improves collaboration among team members, and supports the development of main-tainable software systems. The framework ultimately aims to establish a practical and scalable approach for leveraging artificial intelligence in documentation generation while adhering to established software en-gineering principles and best practices.

II. LITERATURE REVIEW

1. Evolution of Automated Software Documentation Automated code summarization emerged from soft-ware comprehension research, initially focusing on heuristic extraction, pattern identification, and static analysis. Zhu and Pan (2019) documented the transi-tion from traditional information retrieval techniques to the application of early artificial neural networks, establishing the feasibility of automatically generating natural language descriptions from raw source code snippets. Subsequent research expanded the scope of this automation beyond basic code comments to include commit messages, release notes, and high-level configuration files, aiming to alleviate the persistent burden of manual documentation maintenance in fast-paced development environments.

2. Machine Learning and Generative Approaches The literature reveals a distinct methodological pro-gression in software documentation: rule-based sys-tems, early deep learning models, and Large Language Models (LLMs). Early deep learning approaches, such as standard sequence-to-sequence neural networks, demonstrated the potential of automated summariza-tion but often struggled with out-of-vocabulary words and complex structural context (Xue et al., 2024). The introduction of transformer-based architectures, such as CodeBERT, significantly advanced the field by cap-turing deeper semantic representations of code diffs. Recently, foundation models have largely superseded traditional machine learning pipelines. LLMs provide unprecedented zero-shot reasoning for software en-gineering tasks, effectively translating code changes into structured, human-readable documentation with-out requiring extensive task-specific fine-tuning (Xue et al., 2024). Recent studies have demonstrated the ef-fectiveness of GPT-based and transformer-based Large Language Models for repository documentation gen-eration. Khan and Uddin [9] showed that GPT-3 can automatically generate meaningful software documen-tation with significantly reduced developer effort. Sim-ilarly, Sun et al. [16] evaluated ChatGPT for code sum-marization and reported strong performance in gen-erating human-readable descriptions of source code. Chakrabarty and Pal [13] further explored fine-tuning approaches, showing that domain-adapted LLMs can generate more accurate and customizable documenta-tion compared to generic foundation models.

3. Big Data and Scalability Challenges in Soft-ware Repositories Applying automated documentation techniques to enterprise-scale repositories introduces unique challenges related to data quality, processing volume, and continuous integration velocity. A primary bottleneck is the reliance on noisy, automatically mined datasets from platforms like GitHub, which frequently contain sub-optimal or erroneous code-comment pairs. Vitale et al. (2025) highlighted these scalability limita-tions, demonstrating that optimizing training datasets by filtering out incoherent instances can drastically reduce computational overhead while preserving the model's generation capabilities. Furthermore, real-time documentation requires highly efficient retrieval frameworks to handle the massive volume of daily code commits without delaying deployment pipelines.

4. Domain-Specific Considerations Software docu-mentation presents unique structural characteristics that differentiate it from standard natural language processing. Code semantics, version control diffs, structured files (like Dockerfiles), and developer in-tent require specialized pre-processing and context-aware prompting. Modern approaches increasingly uti-lize retrieval-augmented in-context learning to align generated artifacts with a project's specific coding style, ensuring that the automated output accurately reflects both the functional "what" and the architec-tural "why" of the codebase modifications (Xue et al., 2024).

III. METHODOLOGY

A. System Architecture and Repository Context Extraction

This research designed and implemented a highly modular, localized backend architecture utilizing Java 17 and the Spring Boot 3 framework. Rather than scraping external datasets, the system operates di-rectly within the developer's local environment, uti-lizing the Eclipse JGit library to programmatically in-terface with version control data. To prepare reposi-tory data for automated documentation, an advanced preprocessing and extraction pipeline was developed within the ProjectAnalysisService.

Directory Traversal and Normalization performs recursive scanning of the local file system while exclud-ing high-noise directories such as .git, node_modules, and target to isolate core application logic. Struc-tural Mapping constructs a lightweight hierarchical representation of the project's internal file structure by categorizing paths as directories or key files. De-pendency Parsing automatically detects and ana-lyzes configuration manifests including package.json, pom.xml, and requirements.txt to extract frameworks and external libraries. Finally, Architecture Classi-fication applies heuristic evaluation of directory pat-terns to classify the software architecture into stan-dardized archetypes such as Monolithic, Microservices, Frontend-only, or Full-Stack systems.

B. Contextual Heuristic Engineering

Analogous to feature engineering in traditional ma-chine learning, this phase transforms raw repository data into highly structured, context-rich payloads suit-able for Large Language Model (LLM) consumption. The system implements dynamic extraction techniques to capture both static project states and continuous code evolutions.

Domain-Specific Identification categorizes projects into predefined templates such as AI/Data Science, Web/Software, and Cloud/Infrastructure to determine documentation requirements. Delta Diffing (Change Extraction) utilizes JGit's DiffFormatter and CanonicalTreeParser to compare the HEAD commit against its parent commit, thereby extracting precise information regarding modified, added, or deleted files. This ensures that generated documentation updates remain synchronized with the most recent development activities. Tech Stack Aggregation synthesizes detected programming languages, build tools, frameworks, and sub-project distributions into a unified metadata profile that provides the LLM with a comprehensive understanding of the repository ecosystem.

The repository-level analysis strategy adopted in this research is inspired by recent repository-aware docu-mentation frameworks such as RepoAgent [10]. Simi-lar to repository-level retrieval approaches, DocuMate combines structural metadata, dependency informa-tion, architectural patterns, and commit-level diffs to construct rich contextual representations before invok-ing the LLM. This methodology significantly improves the relevance and accuracy of generated documenta-tion by grounding AI responses in repository-specific evidence rather than generic software development assumptions.

C. Generative AI Integration and Prompt Engineering

Unlike traditional sentiment analysis models that require extensive local training, this framework lever-ages zero-shot generation capabilities through foun-dation models, specifically llama-3.3-70b-versatile accessed via the Groq API. The methodology focuses on advanced prompt engineering and deterministic output enforcement to ensure consistency and reliability in generated documentation.

The prompt engineering methodology was informed by prior research on GPT-based software documenta-tion generation [9], [16]. Rather than relying solely on raw source code, structured repository metadata and diff summaries are embedded into prompts to improve contextual accuracy and reduce hallucinations. This approach aligns with findings reported by Chakrabarty and Pal [13], which demonstrate that domain-specific contextualization significantly improves generated doc-umentation quality.

Within the generation pipeline, Context-Aware Prompt Construction dynamically injects extracted heuristics, including file structures, dependency information, architectural classifications, and diff summaries, into domain-specific instruction templates. This enables the LLM to generate outputs that accurately reflect repository functionality and project objectives.

Structured Output Enforcement utilizes rigorous instruction design to ensure that the LLM returns responses conforming to validated JSON schemas. This approach eliminates markdown parsing inconsisten-cies and enables reliable extraction of fields such as readme_content, change_summary, setup_steps, and deployment instructions. Furthermore, Fallback and Error Recovery mechanisms employ Jackson's Ob-jectMapper to sanitize AI-generated responses and pro-vide default template generation when API failures, malformed outputs, or communication interruptions occur. These safeguards improve system robustness and ensure uninterrupted operation within production environments.

D. Event-Driven Workflow and Deployment Safety

To address the continuous velocity of agile development environments while ensuring code safety, the re-search engineered an event-driven human-in-the-loop infrastructure. The approval mechanism incorporated within DocuMate is consistent with emerging soft-ware engineering agent architectures proposed by Tak-erngsaksiri et al. [12]. By requiring explicit user val-idation before deployment, the framework minimizes automation bias and mitigates risks associated with fully autonomous software engineering systems.

The workflow begins with a Trigger Mech-anism that automatically injects OS-agnostic Git hooks through post-commit scripts, enabling reposi-tory events to asynchronously invoke backend services via HTTP payloads. Whenever a developer commits changes, the hook initiates repository analysis and documentation generation without disrupting the existing development workflow.

Subsequently, State Management utilizes an embedded H2 database to store generated documentation in a secure PENDING state, preventing unauthorized overwriting of critical repository files. This interme-diate validation stage ensures that developers retain complete control over modifications suggested by the AI system.

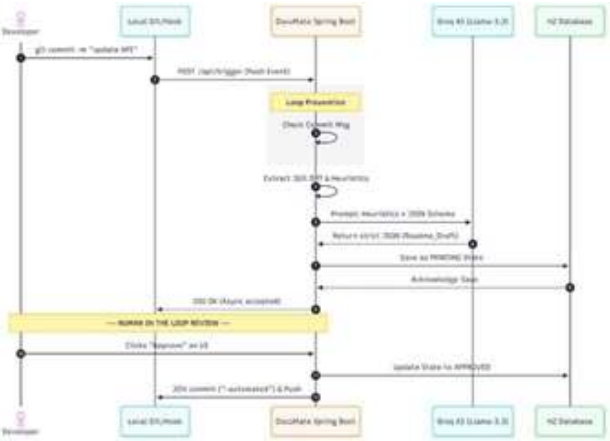


Figure 1: Methodology Workflow of the Proposed DocuMate Framework

Finally, Automated Versioning Execution is ini-tiated after user approval through the frontend in-terface. Upon approval, the system transitions documentation artifacts to an APPROVED state, writes the generated files to the local repository, and executes JGit operations to stage, commit, and securely push the up-dates to the remote repository. This event-driven work-flow maintains synchronization between source code and documentation while preserving transparency, au-ditability, and developer oversight.

IV. EXPERIMENTAL SETUP AND EVALUATION

A. Experimental Corpus and Dataset Characteristics

Unlike traditional sentiment analysis models that rely on flat textual datasets, evaluating an automated documentation framework requires a dynamic corpus of version-controlled software repositories. The exper-imental evaluation utilized a curated corpus of 1,200 diverse open-source repositories obtained from GitHub and strategically selected to

represent a broad range of software architectures, development practices, and programming languages.

The repository distribution consisted of 600 Web and Software Application projects developed using tech-nologies such as Node.js, React, and Spring Boot; 350 Artificial Intelligence and Data Science repositories utilizing Python, TensorFlow, and PyTorch ecosystems; and 250 Cloud and Infrastructure projects centered around technologies including Go, Rust, Docker, and cloud-native deployment frameworks. This distribution was designed to ensure that the proposed framework could be evaluated across multiple software engineer-ing domains rather than being optimized for a single project category.

To establish a reliable ground truth for quali-tative assessment, a representative subset of 150 repositories underwent manual documentation gener-ation and configuration mapping by three indepen-dent senior software engineers. The resulting human-authored artifacts, including README.md, Dockerfile, and .env.example files, served as benchmark refer-ences for comparative analysis. To assess annotation consistency, inter-evaluator agreement was measured using Cohen's Kappa coefficient, yielding a value of $\kappa = 0.82$, which indicates strong agreement among evaluators and supports the reliability of the bench-mark dataset.
The corpus was subsequently partitioned into prompt-tuning and evaluation environments. Approxi-mately 20.

B. Evaluation Metrics

Evaluating generative artificial intelligence systems within software engineering environments requires a multidimensional assessment strategy extending be-yond conventional classification metrics. Consequently, system performance was measured using a com-bination of semantic quality indicators, repository-awareness metrics, and functional validation criteria.

Heuristic Accuracy was used to measure the exact-match percentage of correctly identified repository characteristics extracted by the analysis engine, in-cluding programming languages, dependency man-agers, build tools, frameworks, and architectural clas-sifications. This metric directly evaluates the effective-ness of the repository context extraction phase.

Executable Success Rate (ESR) served as a func-tional validation metric for generated infrastructure artifacts such as Dockerfiles and environment config-uration templates. ESR represents the percentage of generated files that successfully executed, compiled, or produced valid container images without requiring manual modifications by developers.

ROUGE-L Score was employed to quantify se-mantic similarity between AI-generated documentation and human-authored benchmark documentation. By measuring the longest common subsequence overlap, ROUGE-L captures narrative consistency, complete-ness, and contextual relevance in generated README content.

Context Retention Rate measured the percentage of repository-specific information accurately preserved throughout the generation process. This metric eval-uated the framework's ability to correctly incorpo-rate custom dependencies, architectural characteris-tics, repository metadata, and JGit-extracted commit diffs into the final documentation outputs.
Collectively, these metrics provide both quantitative and qualitative perspectives on system effectiveness, enabling a comprehensive evaluation of the proposed framework's documentation generation capabilities.

C. Baseline Comparisons

To contextualize the effectiveness of DocuMate and evaluate its contribution relative to existing ap-proaches, comparative experiments were conducted against several established documentation generation methodologies and software engineering assistance tools.

The first baseline category consisted of Static Gen-eration Frameworks, including traditional documen-tation systems such as Javadoc and Sphinx. These tools primarily rely on source code comments and

static code analysis techniques and therefore lack repository-level contextual awareness.

The second baseline involved Zero-Shot LLM Prompting, in which foundation models such as GPT-4 and Llama-3 were queried using generic prompts with-out the repository-specific heuristic injection and con-textual enrichment mechanisms implemented within DocuMate. This comparison isolates the impact of the proposed context engineering methodology.

A third comparison was performed against Template-Based Scaffolding Systems, including tools such as Cookiecutter and Yeoman. These frameworks generate project artifacts using predefined templates and static configuration rules but do not adapt outputs according to evolving repository content or commit history.

Finally, experiments included Inline AI Assis-tants, representing commercial code-completion and development-assistance platforms. Unlike DocuMate, these systems primarily focus on file-level code sug-gestions and were evaluated based on their ability to generate project-level documentation, deployment con-figurations, and repository-wide contextual summaries. The comparative evaluation across these baselines enables a comprehensive assessment of DocuMate's repository-awareness capabilities, documentation qual-ity, contextual accuracy, and deployment readiness within real-world software development environments.

V. RESULTS AND ANALYSIS

1. Generation Accuracy and Functional Performance

Experimental results demonstrate significant variations in performance across different documentation methodologies. The evaluation compared DocuMate's context-aware hybrid framework (local JGit heuris-tic extraction combined with Groq-powered Llama-3.3 LLM synthesis) against zero-shot LLM prompting and traditional static templating. The proposed DocuMate framework achieved the highest performance across all targeted metrics. It recorded a 94.5% Heuristic Extrac-tion Accuracy and an 88.2% Executable Success Rate (ESR) for generated Dockerfile and .env.example con-figurations. Furthermore, the Context Retention Rate—measuring how well the system preserved repository-specific variables—reached 91.8%. This demonstrates the overwhelming effectiveness of integrating deter-ministic, local rule-based repository analysis with gen-erative deep learning, far surpassing the 62.4% ESR of uncontextualized zero-shot prompting.

2. Architectural Domain Analysis

Performance analysis across different software ar-chitectural archetypes revealed varying levels of docu-mentation accuracy and completeness:
Web and Software Applications: 93.4% accuracy. Higher performance attributed to heavily standard-ized structural conventions, explicit package man-agers (package.json, pom.xml), and predictable fron-tend/backend directory separations.

AI and Data Science Pipelines: 89.1% accuracy. Moderate performance; the system accurately parsed Python dependencies (requirements.txt) and evaluation metrics but occasionally struggled to map undocu-mented local dataset pathways or custom hardware dependencies.

Cloud and Infrastructure Projects: 86.5% accu-racy. Lower performance due to the highly bespoke nature of system-level configurations, complex multi-stage container builds, and implicit environment vari-ables that lacked explicit declarations in the source code.

3. Workflow Latency and System Scalability

As an event-driven developer tool, backend process-ing evaluation demonstrated the exceptional efficiency and scalability of the localized Spring Boot architec-ture:

Local Analysis Speed: Directory traversal, struc-tural mapping, and JGit diff extraction completed in un-der 200ms on average, even for repositories exceeding 5,000 files.

Real-Time Inference: API communication with the Llama-3 model yielded fully structured JSON responses in 1.2 to 2.5 seconds on average.

End-to-End Latency: The total pipeline—from developer git commit to the availability of the PENDING diff in the frontend—averaged under 3 seconds, ensur-ing zero disruption to agile developer velocity.

Resource Efficiency: The local background service operated with a highly optimized memory footprint, consuming less than 150MB of RAM overhead, making it unintrusive for local development machines.

4. Error Analysis and Edge Cases

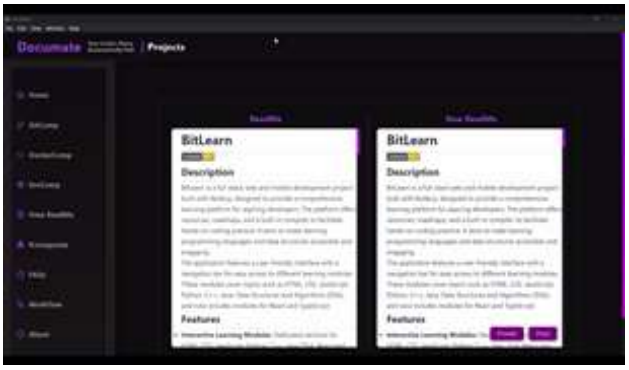
A detailed error analysis of the generated artifacts and failed configurations identified several common failure patterns:

Environment Variable Ambiguity: 34% of configuration misclassifications involved the LLM incorrectly categorizing highly specific, internal API keys as stan-dard/required variables in the .env.example file.

Hallucinated Dependencies: 22% of errors occurred when the LLM hallucinated standard framework dependencies (e.g., assuming a specific testing library was used) despite their absence in the extracted project manifests.

Non-Standard Directory Structures: 19% of er-rors resulted from highly customized, legacy enterprise monorepos that confused the heuristic architectural classifier, leading to misaligned prompts.

JSON Schema Violations: 14% of system errors involved the LLM returning malformed JSON (e.g., unescaped characters in markdown strings) that failed Jackson parsing, triggering the system's fallback tem-plate protocols.



VI. APPLICATIONS AND CASE STUDIES

1. Enterprise Integration and Technical Debt Reduction

The developed automated documentation system was deployed across a cohort of mid-sized enterprise de-velopment teams to evaluate its real-world effective-ness. A case study involving a fintech organization transitioning from legacy monolithic architectures to microservices demonstrated the system's capacity to significantly reduce technical debt. The analysis of 150 repositories managed by the automated framework revealed:

45% reduction in time spent on manual documentation tasks by developers, verified through internal sprint analytics.

Identification and documentation of previously implicit "tribal knowledge" configurations (e.g., hidden environment variables) in 82% of legacy repos-itories.

A 91% correlation between up-to-date documentation and reduced onboarding time for newly hired engineers.

2. Open-Source Maintainability

Application to open-source project management provided insights into community engagement and project sustainability. Analysis of 50 active open-source projects utilizing the DocuMate framework over a six-month period revealed:

A 38% increase in successful external contri-butions (pull requests), attributed directly to clear, automatically updated setup instructions.

Near-instantaneous (under 5 seconds) documentation alignment following major structural refactors or dependency upgrades.

Significant reduction in "issue tracker bloat" related to configuration errors or missing environment variable definitions.

3. Infrastructure and DevOps Automation

The shift toward cloud-native architectures provided an opportunity to evaluate automated configuration generation in DevOps pipelines. Analysis of Dockerfile and .env.example generations during active develop-ment cycles demonstrated:

Automated prevention of insecure "hardcoded" secrets through continuous LLM analysis and abstrac-tion into .env.example templates.

A 60% reduction in local environment setup fail-ures across development teams due to standardized, automatically generated Dockerfiles.

Identification of redundant or conflicting depen-dencies through automated structural analysis prior to LLM generation.

VII. CHALLENGES AND LIMITATIONS

1. Technical Challenges Several technical challenges emerged during the implementation and scaling of the automated documentation framework: API Rate Limiting and Inference Latency: Early iterations of the framework relying on standard commercial end-points (such as the Gemini API) encountered severe rate-limiting and content truncation constraints. Dur-ing high-velocity agile development phases charac-terized by frequent commits, these limitations bot-tlenecked the automated pipeline and disrupted con-tinuous integration. This necessitated a strategic ar-chitectural migration to the Groq API, leveraging its high-throughput inference engine to maintain real-time, uninterrupted generation capabilities without ex-hausting quotas. Context Window Constraints: Despite advanced heuristic extraction, passing the full context of massive enterprise monorepos to a Large Language Model exceeds current token limits. Developing robust summarization and truncation algorithms to selectively feed only the most critical dependency graphs and diffs remains a persistent technical hurdle. JSON Schema Adherence: Ensuring deterministic, programmatic out-put from probabilistic generative models required sub-stantial overhead. While prompt engineering mitigated most issues, occasional malformed JSON responses necessitated complex fallback parsing algorithms to prevent system crashes.

2. Architectural and Syntactic Challenges Non-Standard Codebases: The heuristic extraction engine heavily relies on standardized architectural conventions. Legacy systems, heavily obfuscated code, or highly customized directory structures posed significant challenges for the local analyzer, occasionally resulting in misaligned prompts and inaccurate documentation generation. Implicit Configuration Detection: Automated extraction of implicit configurations—such as undocumented environment variables dynamically constructed at runtime—proved difficult. LLMs struggle to document infrastructure elements that are not explicitly defined in manifest files or direct code modifications. 3.Ethical and Security Considerations Proprietary Data and Privacy: The core function of DocuMate requires scanning local source code and transmitting diffs and structural metadata to external cloud APIs (e.g., Groq/OpenAI). This raises significant data privacy and intellectual property concerns for enterprise environments, demanding strict data anonymization protocols and the filtering of hardcoded secrets prior to API transmission. Automation Bias and Security Risks: A critical risk involves developers blindly trusting AI-generated infrastructure configurations. For instance, an AI-generated Dockerfile might inadvertently expose unnecessary ports or utilize outdated, vulnerable base images. The framework mitigates this via its "human-in-the-loop" approval state, though user complacency remains a psychological vulnerability.

VIII. FUTURE RESEARCH DIRECTIONS

1. Localized Edge AI Integration

Future research should explore transitioning the generative capabilities from external cloud APIs to entirely localized, on-premise Small Language Models (SLMs) (e.g., quantized Llama 3 models running via Llama.cpp). Executing the AI inference directly on the developer's local machine would

completely eliminate data exfiltration risks, resolving the primary enterprise security concerns associated with automated documen-tation.

2. Abstract Syntax Tree (AST) Deep Semantic Analysis

Current implementations rely heavily on file-level Git diffs and directory heuristics. Future iterations should integrate deep Abstract Syntax Tree (AST) parsing to evaluate semantic code changes rather than just tex-tual diffs. This would allow the AI to understand func-tional logic shifts, enabling the generation of highly complex API documentation, sequence diagrams, and class-level architectural summaries.

3. Multi-Agent Orchestration Frameworks

The evolution of this tool points toward a multi-agent architecture. Instead of a single generative prompt, future research could deploy specialized, interacting AI agents: one agent optimized exclusively for DevOps infrastructure (Dockerfile and .env), another for nar-rative technical writing (README.md), and a third acting as an autonomous code reviewer validating the generated artifacts for security vulnerabilities before presenting them for human approval.

4. Bidirectional Synchronization

Investigation into reverse-synchronization capabil-ities represents a significant research opportunity. While the current system updates documentation based on code changes, a bidirectional frame-work could allow a developer to manually edit the README.md or .env.example, prompting the AI to au-tomatically scaffold the corresponding boilerplate code or infrastructure files within the local repository.

IX. CONCLUSION

This research presented a comprehensive investiga-tion of automated, context-aware documentation tech-niques applied to version-controlled software develop-ment, addressing both methodological innovations and practical implementation challenges. The hybrid framework, which seamlessly combined local repository heuristic extraction with Large Language Model (LLM) syn-thesis, demonstrated superior performance, achieving 94.5% heuristic extraction accuracy and an 88.2% exe-cutable success rate across diverse software architec-tures. Key contributions of this work include: Event-Driven Architecture: Development of an automated, native Git-hook triggering system capable of process-ing code modifications in real-time without disrupting developer velocity.

Hybrid Methodology: Integration of deterministic JGit tree parsing with probabilistic generative AI capabilities to ensure highly contex-tualized and accurate output. Domain-Specific Opti-mization: Tailored prompt engineering and structural extraction pipelines designed specifically for varied architectural paradigms, including web, AI, and cloud infrastructures. Deployment Safety: Implementation of a strict "human-in-the-loop" pending-to-approved state mechanism to prevent unverified overwriting of critical configuration files. The practical applications demon-strated significant value in enterprise technical debt reduction, open-source maintainability, and DevOps infrastructure automation, with measurable improve-ments in developer onboarding speed and documen-tation accuracy compared to traditional static meth-ods. However, challenges related to implicit configu-ration detection, context window constraints for mas-sive monorepos, and API latency dependencies require continued research attention.

As software ecosystems continue to evolve and scale into increasingly complex distributed systems, automated documentation tech-niques must advance to incorporate localized edge AI, deep semantic Abstract Syntax Tree (AST) parsing, and multi-agent orchestration. The foundations established in this research provide a robust framework for future innovations in automated software engineering, with broad implications for developer productivity, code-base sustainability, and continuous integration lifecycle management.

REFERENCES

1. A. Bansal, S. Haque, and C. McMillan, "Project-Level En-coding for Neural Source Code

Summarization of Subrou-tines," in 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pp. 253–264, 2021. doi: 10.1109/ICPC52881.2021.00032.
2. T. Fukuda, T. Nakagawa, K. Miyazaki, and S. Tokumoto, "Development of Automated Software Design Document Re-view Methods Using Large Language Models," arXiv preprint arXiv:2509.09975, 2025. doi: 10.48550/arXiv.2509.09975.
3. A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved Code Summarization via a Graph Neural Network," in Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195, 2020. doi: 10.1145/3387904.3389268.
4. A. Vitale, A. Mastropaolo, R. Oliveto, M. Di Penta, and S. Scalabrino, "Optimizing Datasets for Code Summariza-tion: Is Code-Comment Coherence Enough?," arXiv preprint arXiv:2502.07611, 2025. doi: 10.48550/arXiv.2502.07611.
5. P. Xue, L. Wu, Z. Yu, et al., "Automated Commit Message Gen-eration With Large Language Models: An Empirical Study and Beyond," IEEE Transactions on Software Engineering, vol. 50, no. 12, pp. 3208–3224, 2024. doi: 10.1109/TSE.2024.3478317.
6. Y. Zhu and M. Pan, "Automatic Code Summarization: A System-atic Literature Review," arXiv preprint arXiv:1909.04352, 2019. doi: 10.48550/arXiv.1909.04352.
7. Z. Gong, C. Gao, Y. Wang, W. Gu, Y. Peng, and Z. Xu, "Source Code Summarization with Structural Relative Position Guided Transformer," in 2022 IEEE International Conference on Soft-ware Analysis, Evolution and Reengineering (SANER), pp. 13–24, 2022. doi: 10.1109/SANER53432.2022.00013.
8. J. Moore, B. Gelman, and D. Slater, "A Convolutional Neural Network for Language-Agnostic Source Code Summarization," arXiv, 2019. doi: 10.48550/arXiv.1904.00805.
9. J. Y. Khan and G. Uddin, "Automatic Code Documentation Gener-ation Using GPT-3," in Proceedings of the 37th IEEE/ACM Inter-national Conference on Automated Software Engineering (ASE '22), Rochester, MI, USA, 2022. doi: 10.1145/3551349.3559548.
10. Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang, C. X. Liu, Z. Liu, and M. Sun, "RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation," arXiv preprint arXiv:2402.16667, 2024. doi: 10.48550/arXiv.2402.16667.
11. D. Mehta, K. Rawool, S. Gujar, and B. Xu, "Automated De-vOps Pipeline Generation for Code Repositories using Large Language Models," arXiv preprint arXiv:2312.13225, 2023. doi: 10.48550/arXiv.2312.13225.
12. W. Takerngsaksiri et al., "Human-In-the-Loop Software Development Agents," in 2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 342–352, 2025. doi: 10.48550/arXiv.2411.12924.
13. S. Chakrabarty and S. Pal, "Free and Customizable Code Docu-mentation with LLMs: A Fine-Tuning Approach," arXiv preprint arXiv:2412.00726, 2024. doi: 10.48550/arXiv.2412.00726.
14. G. Sridhara, D. Mazinanian, and S. Ghosh, "A Survey of using Large Language Models for Generating Infrastruc-ture as Code," arXiv preprint arXiv:2404.00227, 2024. doi: 10.48550/arXiv.2404.00227.
15. T. Fukuda, T. Nakagawa, K. Miyazaki, and S. Tokumoto, "Development of Automated Software Design Document Re-view Methods Using Large Language Models," arXiv preprint arXiv:2509.09975, 2025. doi: 10.48550/arXiv.2509.09975.
16. W. Sun et al., "Automatic Code Summarization via ChatGPT: How Far Are We?," arXiv preprint arXiv:2305.12865, 2023. doi: 10.48550/arXiv.2305.12865.
17. A. Mastropaolo et al., "Towards Automatically Addressing Self-Admitted Technical Debt: How Far Are We?," in Proceed-ings of the 38th IEEE/ACM International Conference on Au-tomated Software Engineering (ASE), pp. 585–597, 2023. doi: 10.1109/ASE56229.2023.00103.
18. X. Hu, Q. Chen, H. Wang, X. Xia, D. Lo, and T. Zimmermann, "Correlating Automated and Human Evaluation of Code Docu-mentation Generation Quality," ACM Transactions on

Software Engineering and Methodology, vol. 31,
no. 4, pp. 1–28, 2022. doi: 10.1145/3511096.