CodeWiki: Evaluating AI's Ability to Generate Holistic Documentation for Large-Scale Codebases

Nguyen Hoang Anh[1] **Minh Le-Anh**[1] **Bach Le**[2] **Nghi D.Q. Bui**[1,†,§]

[1]FPT Software AI Center, Vietnam [2]The University of Melbourne, Australia

{anhnh2220,minhla4}@fpt.com, bach.le@unimelb.edu.au, bdqnghi@gmail.com *

Abstract

Comprehensive software documentation is crucial yet costly to produce. Despite recent advances in large language models (LLMs), generating holistic, architecture-aware documentation at the repository level remains challenging due to complex and evolving codebases that exceed LLM context limits. Existing automated methods struggle to capture rich semantic dependencies and architectural structure. We present **CodeWiki**, a unified framework for automated repository-level documentation across seven mainstream programming languages. CodeWiki combines top-down hierarchical decomposition with a divide-and-conquer agent system to preserve architectural context and scale documentation generation, and a bottom-up synthesis that integrates textual descriptions with visual artifacts such as architecture and data-flow diagrams. We also introduce **CodeWikiBench**, a benchmark with hierarchical rubrics and LLM-based evaluation protocols. Experiments show that CodeWiki achieves a 68.79% quality score with proprietary models, outperforming the closed-source DeepWiki baseline by 4.73%, with especially strong gains on scripting languages. To support future research, we publicly release both CodeWiki[1] and CodeWikiBench.[2]

1 Introduction

In the rapidly evolving landscape of software development, maintaining comprehensive and up-to-date documentation has become increasingly challenging as codebases grow in size and complexity. Studies indicate that developers spend approximately 58% of their working time on program comprehension activities (Xia et al., 2018), highlighting the critical importance of accessible and accurate documentation. Despite documentation being recognized as essential for software maintenance and collaboration (de Souza et al., 2005; Zhi et al., 2015; Chen and Huang, 2009), the manual creation and maintenance of repository-level documentation remain a labor-intensive and often neglected aspect of software engineering practice (McBurney et al., 2018). The emergence of Large Language Models (LLMs) (Vaswani et al., 2017; Zhang et al., 2023; He et al., 2025b) has opened opportunities for automating documentation generation tasks. However, current approaches still face two fundamental limitations.

Limitations of Scalability at the Repository Level. Recent advances in code understanding and natural language generation have demonstrated promising results in function and file documentation tasks (Feng et al., 2020; Khan and Uddin, 2023; Poudel et al., 2024; Makharev and Ivanov, 2025; Lomshakov et al., 2024; Ahmed and Devanbu, 2023; Choi et al., 2023; Luo et al., 2024; Yang et al., 2025). However, existing approaches face significant challenges when scaling to repository documentation. Unlike function documentation that focuses on individual components, repository documentation must capture architectural patterns, interactions between modules, data flows, and design decisions that span the entire system (Rai et al., 2022; Treude et al., 2020). Furthermore, the complex hierarchical nature of software systems requires documentation that can serve multiple purposes, such as for stakeholders seeking architectural overviews or developers requiring detailed implementation guidance (Nassif and Robillard, 2025).

↪ To address these challenges, we present **CodeWiki**, an open-source framework designed to generate holistic repository documentation that includes diverse content types such as system architecture diagrams, data flow visualizations, and sequence diagrams while scaling effectively for

*†: Project Lead, §: Corresponding Author

[1]https://github.com/FSoft-AI4Code/CodeWiki

[2]https://github.com/FSoft-AI4Code/CodeWikiBench

Findings of the Association for Computational Linguistics: ACL 2026, pages 5812–5827

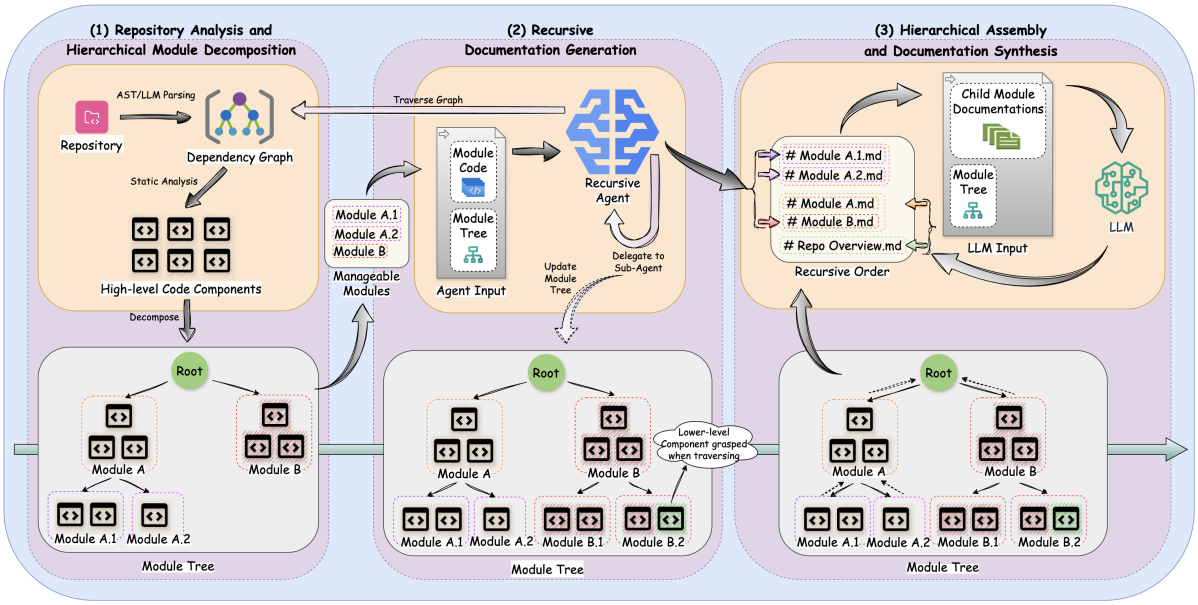

Figure 1: CodeWiki Framework Architecture Overview. The framework operates in three main phases: (1) Repository analysis through AST/LLM parsing to construct dependency graphs and identify high-level components, followed by hierarchical decomposition into manageable modules; (2) Recursive documentation generation where specialized agents process leaf modules with dynamic delegation capabilities, creating markdown documentation while maintaining cross-module references; (3) Hierarchical assembly where parent modules are synthesized from child documentation using LLM-based synthesis, culminating in comprehensive repository overview documentation. The module tree evolves throughout the process, enabling scalable processing of repositories of arbitrary size.

projects of arbitrary size. While prior work such as RepoAgent (Luo et al., 2024) aggregates function and class documentation into repository structures, and open-source alternatives like Open-DeepWiki apply LLMs to entire repositories, CodeWiki differs through its hierarchical synthesis approach where parent modules are synthesized from child documentation, combined with integrated visual artifacts capturing architectural patterns. CodeWiki introduces three key innovations: (1) a hierarchical decomposition strategy inspired by dynamic programming principles that breaks complex repositories into manageable modules while preserving architectural coherence; (2) a scalable divide-and-conquer agent system that dynamically determines the relevant context for each module, recursively partitions it into sub-modules handled by dedicated sub-agents; and (3) a bottom-up synthesis process that integrates textual descriptions with visual artifacts such as architecture diagrams and data-flow representations to generate high-level documentation.

Limitations of the Evaluation Landscape. Despite progress in automated documentation, there is still a lack of systematic and standardized benchmarks. Existing work at the function-level and file-level has mostly been evaluated using general-purpose text generation metrics (e.g., BLEU, ROUGE), which fail to capture the nuanced quality of software documentation (Roy et al., 2021; Evtikhiev et al., 2023). For repository-level documentation, the large scale and complexity of software systems make evaluation particularly challenging, as the quality of documentation cannot be adequately judged by surface-level text-similarity metrics alone. A single repository can be described through documentation with different structures and presentations, yet all share the same underlying architectural content. This highlights the lack of a robust evaluation system that can fairly assess documentation quality regardless of how it is presented.

↪ To fill this gap, we introduce **CodeWikiBench**, the first benchmark specifically designed for repository-level documentation. Instead of relying on surface-level similarity, we automatically derive hierarchical evaluation rubrics from the ground-truth official documentation maintained by project developers, enabling the automated evaluation of machine-generated documentation using the derived rubrics. This approach is inspired by recent advances in hierarchical rubric-based eval-

uation (Starace et al., 2025), which demonstrates the effectiveness of decomposing complex evaluation tasks into smaller, well-defined sub-tasks with clear grading criteria. For robustness, CodeWikiBench adopts an agentic assessment framework in which judge agents evaluate generated documentation against rubric requirements and aggregate results hierarchically, producing both quality scores and reliability measures.

Limitations of Multilingual Support. Most studies on code documentation generation have primarily focused on Python, with limited consideration of other widely used programming languages such as Java, JavaScript, C, or C++. This narrow scope restricts generalizability and overlooks the structural diversity inherent in real-world software projects, where repositories often involve multiple languages.

↪ CodeWiki supports repository-level documentation generation across 7 programming languages (Python, Java, JavaScript, TypeScript, C, C++ and C#). In addition, CodeWikiBench provides a multilingual benchmark built from multiple repositories in the same set of languages, enabling comprehensive evaluation.

Using CodeWikiBench, we evaluate the effectiveness of CodeWiki against multiple repository-level documentation systems, including open-source baselines (**OpenDeepWiki**[3], **deepwiki-open**[4]) and the closed-source **DeepWiki**. The results show that CodeWiki achieves an overall quality score of 68.79%, outperforming all baselines with improvements of up to 18.54% over the strongest baseline, DeepWiki, on individual repositories. CodeWiki produces comprehensive documentation encompassing architectural diagrams, usage patterns, and cross-module dependency visualizations (see Appendix A for an example).

In summary, our main contributions are:

1. **CodeWiki**, an open-source framework for scalable repository documentation generation combining top-down and bottom-up analyses.
2. **CodeWikiBench**, a benchmark for repository documentation featuring hierarchical rubrics and an agentic assessment framework.
3. Support for documentation generation across seven programming languages (Python, Java, JavaScript, TypeScript, C, C++, and C#).
4. Experimental validation demonstrating improvements over all baselines, with gains of up to 18.54% on individual repositories and 4.73% on average compared to the closed-source DeepWiki system.

A detailed discussion of related work and positioning relative to prior approaches is provided in Appendix B.

2 Our Approach - CodeWiki

CodeWiki is a semi-agentic framework that automatically generates comprehensive repository-level documentation by addressing context limitations through hierarchical decomposition. As illustrated in Figure 1, our approach operates through three phases: repository analysis and module decomposition, recursive documentation generation, and hierarchical assembly, inspired by dynamic programming principles.

2.1 Repository Analysis and Hierarchical Module Decomposition

We begin with static analysis to construct a dependency graph capturing structural relationships, crucial for understanding architecture and identifying decomposition boundaries.

Dependency Graph Construction Following DocAgent (Yang et al., 2025), we employ Tree-Sitter parsers to extract Abstract Syntax Trees (ASTs). We systematically identify functions, methods, classes, structs, modules, and interdependencies including: function calls (runtime dependencies), class inheritance (structural hierarchies), attribute access (data dependencies), and module imports (compilation dependencies). We normalize these to a unified `depends_on` relation for cross-language generalization.

This creates a directed graph $G = (V, E)$ where vertices V represent components and edge $(u, v) \in E$ indicates component u depends on v. Understanding v is prerequisite to comprehending u, fundamental to our hierarchical processing (Liu et al., 2023).

Entry Point Identification and Decomposition Topological sorting identifies zero-in-degree components, independent entry points where users interact with the project (main functions, API endpoints, CLI, public interfaces). Components from these files represent high-level features and documentation starting points.

[3] https://github.com/AIDotNet/OpenDeepWiki
[4] https://github.com/AsyncFuncAI/deepwiki-open

High-level components are hierarchically decomposed into manageable modules through recursive partitioning. For scalability, only component IDs (relative paths) serve as input. Decomposition considers: component interdependencies (grouping related functionality) and semantic coherence (preserving logical boundaries). The result is a feature-oriented module tree where each node contains component lists and sub-modules.

2.2 Recursive Documentation Generation

CodeWiki's core innovation is recursive agent processing, enabling arbitrary repository size handling while maintaining bounded complexity and architectural coherence.

Agent Architecture Each leaf module is assigned a specialized agent equipped with: (1) complete source code access, (2) full module tree for cross-module understanding, (3) documentation workspace tools, and (4) dependency graph traversal for contextual exploration.

The agent workflow: analyzes components to understand functionality and interfaces, explores context by traversing dependencies, and generates comprehensive markdown documentation including descriptions, usage examples, API specifications, and architectural insights.

Dynamic Delegation Our key innovation is adaptive scalability through dynamic delegation. When module complexity exceeds single-pass capacity, agents delegate submodules to specialized sub-agents. Delegation criteria include code complexity metrics, semantic diversity, and context window utilization.

The recursive process follows bottom-up processing (see Algorithm 1 in Appendix C). Upon delegation, agents provide submodule specifications, the module tree updates, and newly created leaves process recursively. This enables handling modules of any size while maintaining quality.

Cross-Module Reference Management Maintaining coherence across boundaries requires sophisticated reference management. When agents encounter external components, our resolution system creates cross-references rather than duplicating content. A global registry tracks documented components, enabling quick identification and navigation between related components.

2.3 Hierarchical Assembly and Documentation Synthesis

After leaf documentation, hierarchical assembly synthesizes component-level details into architectural overviews through recursive parent module processing.

Parent modules undergo LLM synthesis with carefully crafted prompting. LLMs receive: child module documentation, module tree structure, dependency information, and synthesis instructions for architectural patterns and feature interactions.

Synthesis involves multiple stages: analyzing child documentation for themes and patterns, generating architectural overviews explaining module collaboration, creating feature summaries distilling capabilities, developing usage guides for public interfaces, and producing architectural diagrams visualizing relationships and data flows.

3 The First Benchmarking System - CodeWikiBench

Repository-level documentation evaluation presents unique challenges due to complexity and hierarchical organization. Unlike code generation assessable through automated metrics, documentation quality requires nuanced evaluation of completeness, accuracy, and coherence across abstraction levels. Recent research highlights traditional metrics' limitations for code tasks (Evtikhiev et al., 2023; Haldar and Hockenmaier, 2024), while LLM-as-a-Judge paradigms offer new possibilities (Wang et al., 2025; Crupi et al., 2025; He et al., 2025a). We develop a comprehensive framework synthesizing repository-specific rubrics and employing agentic assessment.

3.1 Hierarchical Rubric Generation

Following (Starace et al., 2025), we synthesize evaluation rubrics tailored for each repository, illustrated in Figure 2. We collect official documentation from selected open-source projects and hierarchically parse into structured JSON format, leveraging directory structure and markdown syntax.

A Rubric Generator Agent processes each structure with tool access to detailed contents, generating hierarchical rubrics mirroring repository functionality. To enhance reliability, we employ multiple independent generations using different model families. Final rubrics emerge from synthesizing these perspectives, reducing single-model biases. Analysis shows generated rubrics demon-

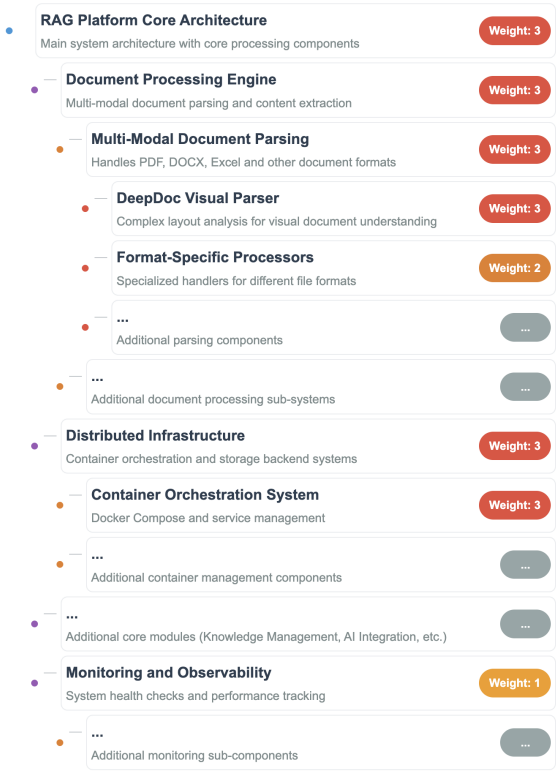

Figure 2: Example hierarchical evaluation rubric for the RAGFlow repository. The rubric mirrors the project's architectural structure with weighted requirements at multiple levels. Leaf nodes represent specific requirements assessed by Judge Agent, while parent scores are computed through weighted aggregation. The hierarchical organization ensures comprehensive coverage from high-level architectural components down to specific implementation details, with weights reflecting the relative importance of each component for understanding the repository

strate high consistency: 73.65% semantic reliability and 70.84% structural reliability (Appendix D).

3.2 Documentation Assessment

Our evaluation employs specialized Judge Agents operating under carefully designed protocols. Judges receive generated documentation in structured JSON format with hidden detailed contents, plus repository-specific rubrics. The systematic methodology: analyzes documentation structure and content, searches comprehensively for requirement coverage, makes binary adequacy decisions, and provides concise reasoning.

Critically, judges evaluate only leaf-level requirements, ensuring assessment grounds in specific concrete criteria rather than abstract concepts. For example, in Figure 2, agents assess "DeepDoc Visual Parser for complex layouts" rather than broader "Document Processing Engine."

For enhanced reliability, multiple Judge Agents using different model families evaluate each requirement. Final leaf scores average across assessments, reducing individual model biases.

3.3 Hierarchical Score Aggregation

Final quality scores compute through bottom-up weighted aggregation respecting rubric hierarchy while tracking reliability via standard deviation propagation. Let R be the rubric root, $C(n)$ denote node n's children, and $w(n)$ denote assigned weight.

Leaf Assessment and Reliability For leaf nodes ℓ, multiple assessments $s_1, s_2, \ldots, s_m$ aggregate as:

$$S(\ell) = \bar{s} = \frac{1}{m}\sum_{i=1}^{m} s_i$$

Reliability quantified via standard deviation:

$$\sigma_\ell = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(s_i - \bar{s})^2}$$

Lower deviations indicate higher judge consensus and assessment reliability.

Hierarchical Propagation For internal nodes n with children $C(n) = \{c_1, \ldots, c_k\}$ having scores $S(c_i)$, weights $w(c_i)$, and deviations σ_{c_i}:

$$S(n) = \frac{\sum_{i=1}^{k} w(c_i) \cdot S(c_i)}{\sum_{i=1}^{k} w(c_i)}$$

$$\sigma_n = \frac{\sqrt{\sum_{i=1}^{k} w(c_i)^2 \sigma_{c_i}^2}}{\sum_{i=1}^{k} w(c_i)}$$

This ensures uncertainty propagates appropriately, with weights determining score contribution and reliability influence.

Final Assessment Final repository score $S(R) \in [0, 1]$ with deviation σ_R represents weighted aggregation propagating assessments and reliability upward. Lower σ_R indicates higher methodology reliability. This dual-metric approach provides quality assessment with confidence bounds, enabling nuanced interpretation. Weights reflect component importance based on criticality and complexity while maintaining statistical rigor in uncertainty quantification.

Repository	Language	LOC	System	Score (%)	Coverage	Improvement (%)
All-Hands-AI–OpenHands	Python	229,909	OpenDeepWiki	58.12 ± 3.21	42/67	–
			deepwiki-open	61.35 ± 2.98	45/67	–
			DeepWiki	73.04 ± 2.54	54/67	–
			CodeWiki	**82.45 ± 2.65**	**59/67**	**+9.41**
sveltejs–svelte	JavaScript	124,576	OpenDeepWiki	55.23 ± 3.85	61/96	–
			deepwiki-open	57.89 ± 3.62	64/96	–
			DeepWiki	68.51 ± 3.31	76/96	–
			CodeWiki	**71.96 ± 3.73**	**80/96**	**+3.45**
puppeteer–puppeteer	TypeScript	136,302	OpenDeepWiki	51.82 ± 4.15	48/82	–
			deepwiki-open	54.67 ± 3.94	51/82	–
			DeepWiki	64.46 ± 3.72	60/82	–
			CodeWiki	**83.00 ± 3.37**	**74/82**	**+18.54**
Unity-Technologies–ml-agents	C#	86,106	OpenDeepWiki	62.45 ± 4.28	32/46	–
			deepwiki-open	65.12 ± 4.05	34/46	–
			DeepWiki	74.80 ± 3.69	39/46	–
			CodeWiki	**79.78 ± 5.02**	**42/46**	**+4.98**
elastic–logstash	Java	117,485	OpenDeepWiki	41.25 ± 4.52	28/57	–
			deepwiki-open	44.18 ± 4.31	31/57	–
			DeepWiki	54.80 ± 4.10	38/57	–
			CodeWiki	**57.90 ± 3.43**	**38/57**	**+3.10**
wazuh–wazuh	C	1,446,730	OpenDeepWiki	32.56 ± 5.82	18/46	–
			deepwiki-open	35.89 ± 5.45	21/46	–
			DeepWiki	**68.68 ± 4.74**	**39/46**	–
			CodeWiki	64.17 ± 5.44	34/46	-4.51
electron–electron	C++	184,234	OpenDeepWiki	28.45 ± 3.95	35/92	–
			deepwiki-open	31.22 ± 3.78	38/92	–
			DeepWiki	**44.10 ± 3.12**	**54/92**	–
			CodeWiki	42.30 ± 3.26	48/92	-1.80
Average			OpenDeepWiki	47.13 ± 4.25		–
			deepwiki-open	50.05 ± 4.02		–
			DeepWiki	64.06 ± 3.60		–
			CodeWiki	**68.79 ± 3.84**		**+4.73**

Table 1: Documentation generation performance comparison across repositories. Scores represent weighted averages with standard deviations indicating assessment consensus. Coverage shows satisfied leaf criteria out of total requirements. OpenDeepWiki, deepwiki-open are open-source alternatives; DeepWiki is the closed-source baseline.

4 Experiment

We conduct comprehensive experiments to assess CodeWiki's effectiveness in generating repository-level documentation across diverse programming languages and project types. Our evaluation is structured around 3 core research questions that address the fundamental challenges of automated repository documentation generation.

4.1 Research Questions

Our experimental evaluation is guided by the following research questions:

RQ1: Documentation Quality and Coverage How does CodeWiki compare to existing state-of-the-art repository documentation systems in terms of overall documentation quality and requirement coverage?

RQ2: Language Generalization Does CodeWiki demonstrate consistent performance across diverse programming languages, particularly comparing high-level scripting languages (Python, JavaScript, TypeScript) versus systems programming languages (C, C++, C#, Java)?

RQ3: Scalability and Reliability How does CodeWiki's hierarchical decomposition approach perform across repositories of varying sizes and complexity, and what is the reliability of our evaluation methodology?

4.2 Baselines

We compare CodeWiki against multiple repository-level documentation systems spanning both open-source and closed-source solutions:

Open-Source Baselines **OpenDeepWiki** and **deepwiki-open** are open-source alternatives applying LLMs to entire repositories for documentation generation. These systems represent the current state of open-source repository-level documenta-

tion tools.

Closed-Source Baseline **DeepWiki** (DeepWiki, 2025) is a commercial system has demonstrated effectiveness in industrial applications for automated repository-level documentation generation.

Positioning Relative to Function-Level Approaches We note that function-level documentation systems such as RepoAgent (Luo et al., 2024) and DocAgent (Yang et al., 2025) generate N separate documents for N components, representing a fundamentally different documentation paradigm from CodeWiki's hierarchically synthesized module-level outputs. Direct quantitative comparison would require forcing one approach into evaluation criteria it was not designed to satisfy, so we focus our comparison on systems targeting similar repository-level documentation goals.

4.3 Experimental Setup

Repository Selection We curate a benchmark of 7 open-source repositories spanning diverse programming languages, project scales, and application domains. Our selection prioritizes: (1) language diversity for cross-language generalization, (2) varying codebase sizes ranging from 86K to 1.4M LOC, (3) actively maintained projects with high-quality official documentation for rubric construction, and (4) coverage of multiple domains such as web frameworks, automation tools, and machine learning platforms.

Table 1 presents the repository characteristics, including primary languages and codebase sizes. The benchmark covers JavaScript (svelte), TypeScript (puppeteer), C (wazuh), C++ (electron), C# (ml-agents), Java (logstash), and Python (OpenHands), ensuring comprehensive coverage across major programming paradigms.

Temporal Separation for Data Leakage Prevention To ensure evaluation integrity, we use repository snapshots from August-September 2025, which postdate the knowledge cutoffs of all models used in our experiments (Claude Sonnet 4: March 2025). This temporal gap provides strong evidence against data leakage concerns. Detailed commit information is provided in Appendix E.

Model Configuration Our experimental configuration employs state-of-the-art language models: Claude Sonnet 4 for documentation generation, multiple models (Claude Sonnet 4, Gemini 2.5 Pro, and Kimi K2 Instruct) for rubric generation to reduce single-model bias, and three distinct judge models (Gemini 2.5 Flash, GPT OSS 120B, and Kimi K2 Instruct) for independent assessments ensuring robust evaluation through multi-model consensus. Configuration details are in Appendix G.

Evaluation Protocol For each repository, we execute the CodeWiki pipeline and generate comprehensive documentation including architectural overviews, component descriptions, and usage guides. The evaluation process follows our hierarchical rubric methodology: (1) generating repository-specific rubrics from official documentation, (2) performing agentic assessment of generated documentation, and (3) computing weighted scores with uncertainty quantification through standard deviation tracking.

4.4 Results and Analysis

4.4.1 RQ1: Documentation Quality and Coverage

Table 1 presents comprehensive evaluation results addressing the fundamental question of documentation quality. CodeWiki achieves an average documentation quality score of 68.79%, representing a 4.73 percentage point improvement over the closed-source DeepWiki baseline (64.06%) and substantial improvements over open-source alternatives OpenDeepWiki (47.13%) and deepwiki-open (50.05%). This demonstrates the effectiveness of our hierarchical synthesis approach compared to both simpler open-source methods and commercial solutions.

The coverage metrics reveal that CodeWiki consistently satisfies more leaf-level requirements than baselines in most cases. Notably, CodeWiki outperforms all baselines in 5 out of 7 repositories, with particularly strong improvements in TypeScript (83.00% vs. DeepWiki's 64.46%), Python (82.45% vs. 73.04%), and JavaScript (71.96% vs. 68.51%). The open-source alternatives show notably degraded performance on more complex codebases, suggesting that simpler whole-repository prompting approaches do not scale effectively.

Answer: CodeWiki demonstrates superior overall documentation quality and coverage compared to both open-source and closed-source systems, achieving improvements of 21.66% over OpenDeepWiki, 18.74% over deepwiki-open, and 4.73% over DeepWiki.

4.4.2 RQ2: Language Generalization

Our analysis reveals distinct performance patterns across programming language categories. CodeWiki demonstrates strong performance on scripting languages, achieving average scores of 79.14% (Python, JavaScript, TypeScript) repositories - a 10.47% improvement over DeepWiki's 68.67% average for the same languages.

For managed languages (C#, Java), CodeWiki achieves competitive performance with moderate but consistent improvements of 4.98% and 3.10% respectively, averaging 68.84% versus DeepWiki's 64.80%. Systems programming languages (C, C++) present challenges where both frameworks exhibit similar patterns: DeepWiki achieves 56.39% while CodeWiki reaches 53.24%. This suggests that specialized parsing strategies may benefit both approaches for handling complex constructs like pointer and template metaprogramming.

Answer: CodeWiki demonstrates strong generalization with notable performance on scripting languages (79.14% average) and consistent improvements on managed languages, while systems programming languages present similar challenges for both frameworks.

4.4.3 RQ3: Scalability and Reliability

Repository size analysis demonstrates CodeWiki's scalability across diverse project scales. Our framework processes repositories ranging from 86K LOC to 1.4M LOC, with performance consistency observed within language categories regardless of repository size. For high-level languages, CodeWiki maintains strong performance across different scales: Unity ML-Agents (86K LOC) achieves 79.78%, while the larger OpenHands (230K LOC) reaches 82.45%. This consistency demonstrates that our hierarchical approach successfully addresses context limitation challenges that typically constrain documentation generation at scale.

Critically, both CodeWiki and DeepWiki exhibit similar behavioral patterns across different repository sizes within the same programming language families. Systems programming languages present consistent challenges for both frameworks regardless of project scale, suggesting that the primary factor affecting documentation quality is language-specific parsing complexity rather than repository size. The dynamic delegation mechanism in CodeWiki automatically adapts to varying complexity levels, maintaining bounded processing requirements through hierarchical decomposition.

Reliability analysis through standard deviation measurements validates our multi-model assessment methodology. The average standard deviations of 3.84% (CodeWiki) versus 3.60% (DeepWiki) indicate comparable assessment reliability while achieving superior overall quality scores. A pilot human study (Appendix F) provides additional validation, showing that human preferences align with automated evaluation results (CodeWiki preferred in 7/9 assessments across 3 participants and 3 repositories).

Answer: CodeWiki's hierarchical decomposition enables consistent scalability across repository sizes from 86K to 1.4M LOC, with performance variations primarily attributable to language-specific characteristics rather than scale limitations. Human evaluation validates alignment between our automated methodology and human judgment.

5 Conclusion

We present CodeWiki, an open-source framework for automated repository documentation using hierarchical decomposition and recursive processing. CodeWiki performs bottom-up hierarchical synthesis, generating parent modules from children while integrating visual artifacts. Across 7 multilanguage repositories, CodeWiki achieves 68.79% quality score, outperforming DeepWiki's 64.06%. Performance variations correlate primarily with language characteristics rather than repository size, with scripting languages showing consistent improvements while systems languages face challenges for all frameworks.

Future work includes improved parsing for systems languages, version-aware documentation aligned with code evolution, and using documentation as semantic bridges for tasks like code migration. We release CodeWiki as open source to advance research in hierarchical documentation generation.

Limitations

While CodeWiki demonstrates strong performance across multiple languages and repository scales, several limitations warrant discussion.

Systems Programming Languages Our framework exhibits reduced effectiveness on systems programming languages (C, C++) compared to high-level languages, achieving 53.24% versus 79.14% average scores. This gap stems from the inherent complexity of low-level constructs such as pointer operations, manual memory management, and template metaprogramming. For systems languages these constructs are the architecture, making this a limitation of our current analysis phase rather than a secondary concern. Developing specialized parsing modules that leverage language-specific characteristics represents an important direction for future work.

Rubric Generation The automatically generated rubrics, while showing 73.65% semantic and 70.84% structural reliability across model families (Appendix D), have not undergone comprehensive human verification. Some generated rubrics may be difficult to interpret, and future work should incorporate systematic human validation to ensure rubric quality and clarity.

Evaluation Methodology Our evaluation methodology relies on LLM-based assessment with multi-model consensus, which may introduce model-specific biases despite our mitigation strategies. A pilot human study (Appendix F) with 3 participants across 3 repositories showed alignment between human preferences and automated scores (CodeWiki preferred in 7/9 assessments), providing preliminary validation. However, broader human evaluation with more participants and systematic rubric-based assessment would further strengthen the benchmark's validity. The average standard deviation of 3.84% indicates reasonable consistency, yet edge cases may benefit from expert validation.

References

Toufique Ahmed and Premkumar Devanbu. 2023. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Rahul K. Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, Johannes Heidecke, and Karan Singhal. 2025. Healthbench: Evaluating large language models towards improved human health.

Jie-Cherng Chen and Sun-Jen Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *J. Syst. Softw.*, 82(6):981–992.

Yunseok Choi, Cheolwon Na, Hyojun Kim, and Jee-Hyong Lee. 2023. Readsum: Retrieval-augmented adaptive transformer for source code summarization. *IEEE Access*, 11:51155–51165.

Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the effectiveness of llm-as-a-judge for code generation and summarization.

Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, SIGDOC '05, page 68–75, New York, NY, USA. Association for Computing Machinery.

DeepWiki. 2025. Deepwiki.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: How should we assess quality of the code generation models? *J. Syst. Softw.*, 203(C).

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*, pages 1536–1547, Online. Association for Computational Linguistics.

Rajarshi Haldar and Julia Hockenmaier. 2024. Analyzing the performance of large language models on code summarization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 995–1008, Torino, Italia. ELRA and ICCL.

Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2025a. From code to courtroom: Llms as the new software judges.

Junda He, Christoph Treude, and David Lo. 2025b. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5).

Junaed Younus Khan and Gias Uddin. 2023. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Yuta Koreeda, Terufumi Morishita, Osamu Imaichi, and Yasuhiro Sogawa. 2023. Larch: Large language model-based automatic readme creation with heuristics. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, CIKM '23, page 5066–5070, New York, NY, USA. Association for Computing Machinery.

Bill Yuchen Lin, Yuntian Deng, Khyathi Chandu, Abhilasha Ravichander, Valentina Pyatkin, Nouha Dziri, Ronan Le Bras, and Yejin Choi. 2025. Wildbench: Benchmarking LLMs with challenging tasks from real users in the wild. In *The Thirteenth International Conference on Learning Representations*.

Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 345–357.

Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. 2024. ProConSuL: Project context for code summarization with LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 866–880, Miami, Florida, US. Association for Computational Linguistics.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 436–464, Miami, Florida, USA. Association for Computational Linguistics.

Vladimir Makharev and Vladimir Ivanov. 2025. Code summarization beyond function level. pages 153–160.

Paul W. McBurney, Siyuan Jiang, Marouane Kessentini, Nicholas A. Kraft, Ameer Armaly, Mohamed Wiem Mkaouer, and Collin McMillan. 2018. Towards prioritizing documentation effort. *IEEE Trans. Softw. Eng.*, 44(9):897–913.

Mathieu Nassif and Martin P. Robillard. 2025. Nonlinear software documentation with interactive code examples. *ACM Trans. Softw. Eng. Methodol.*, 34(2).

Bibek Poudel, Adam Cook, Sekou Traore, and Shelah Ameli. 2024. Documint: Docstring generation for python using small language models. *arXiv preprint arXiv:2405.10243)*.

Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. 2022. A review on source code documentation. *ACM Trans. Intell. Syst. Technol.*, 13(5).

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1105–1116, New York, NY, USA. Association for Computing Machinery.

Ved Sirdeshmukh, Kaustubh Deshpande, Johannes Mols, Lifeng Jin, Ed-Yeremai Cardona, Dean Lee, Jeremy Kritz, Willow Primack, Summer Yue, and Chen Xing. 2025. Multichallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier llms.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. 2025. Paperbench: Evaluating ai's ability to replicate ai research.

Christoph Treude, Justin Middleton, and Thushari Atapattu. 2020. Beyond accuracy: assessing software documentation quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1509–1512, New York, NY, USA. Association for Computing Machinery.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. 2025. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proc. ACM Softw. Eng.*, 2(ISSTA).

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: a large-scale field study with professionals. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 584, New York, NY, USA. Association for Computing Machinery.

Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. 2025. DocAgent: A multi-agent system for automated code documentation generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 460–471, Vienna, Austria. Association for Computational Linguistics.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A survey on large language models for software engineering. *ArXiv*, abs/2312.15223.

Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. 2015. Cost, benefits and quality of software development documentation. *J. Syst. Softw.*, 99(C):175–198.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*.

Appendices

A Example Generated Documentation

Figure 3 shows an example of repository-level documentation generated by CodeWiki for the All-Hands-AI–OpenHands repository. The documentation provides a comprehensive overview including the repository's purpose, end-to-end architecture with visual diagrams, and hierarchical navigation of core components. This demonstrates CodeWiki's ability to synthesize both textual explanations and visual artifacts that capture architectural patterns and module relationships, enabling developers to quickly understand the system's structure and design decisions.

B Background and Related Work

B.1 Code Summarization and Documentation

B.1.1 Existing Approaches

Advances in natural language processing (NLP) have enabled significant progress in the task of code summarization. Existing methods (Khan and Uddin, 2023; Poudel et al., 2024; Makharev and Ivanov, 2025; Lomshakov et al., 2024; Ahmed and Devanbu, 2023; Choi et al., 2023; Luo et al., 2024; Yang et al., 2025) primarily focus on function-level documentation, and can broadly be divided into two categories based on the type of context they exploit (see Table 2 for a comprehensive comparison):

Local-Level Context Early neural approaches established the foundation for automated documentation. CodeBERT (Feng et al., 2020) demonstrated effective bimodal training on code and natural language, while more recent LLM applications have shown substantial improvements. For instance, (Khan and Uddin, 2023) reported 20.6 BLEU scores across six languages using GPT-3, an 11.2% improvement over prior methods. DocuMint (Poudel et al., 2024) further explored the use of small language models for Python docstring generation with promising results. However, these approaches primarily rely on local code context and do not leverage repository-level information to improve documentation quality.

Codebase-Level Context Recognizing the limitations of function-only context, recent work has explored broader codebase information to enrich documentation. (Makharev and Ivanov, 2025) demonstrated that incorporating class- and repository-level context can substantially improve summary

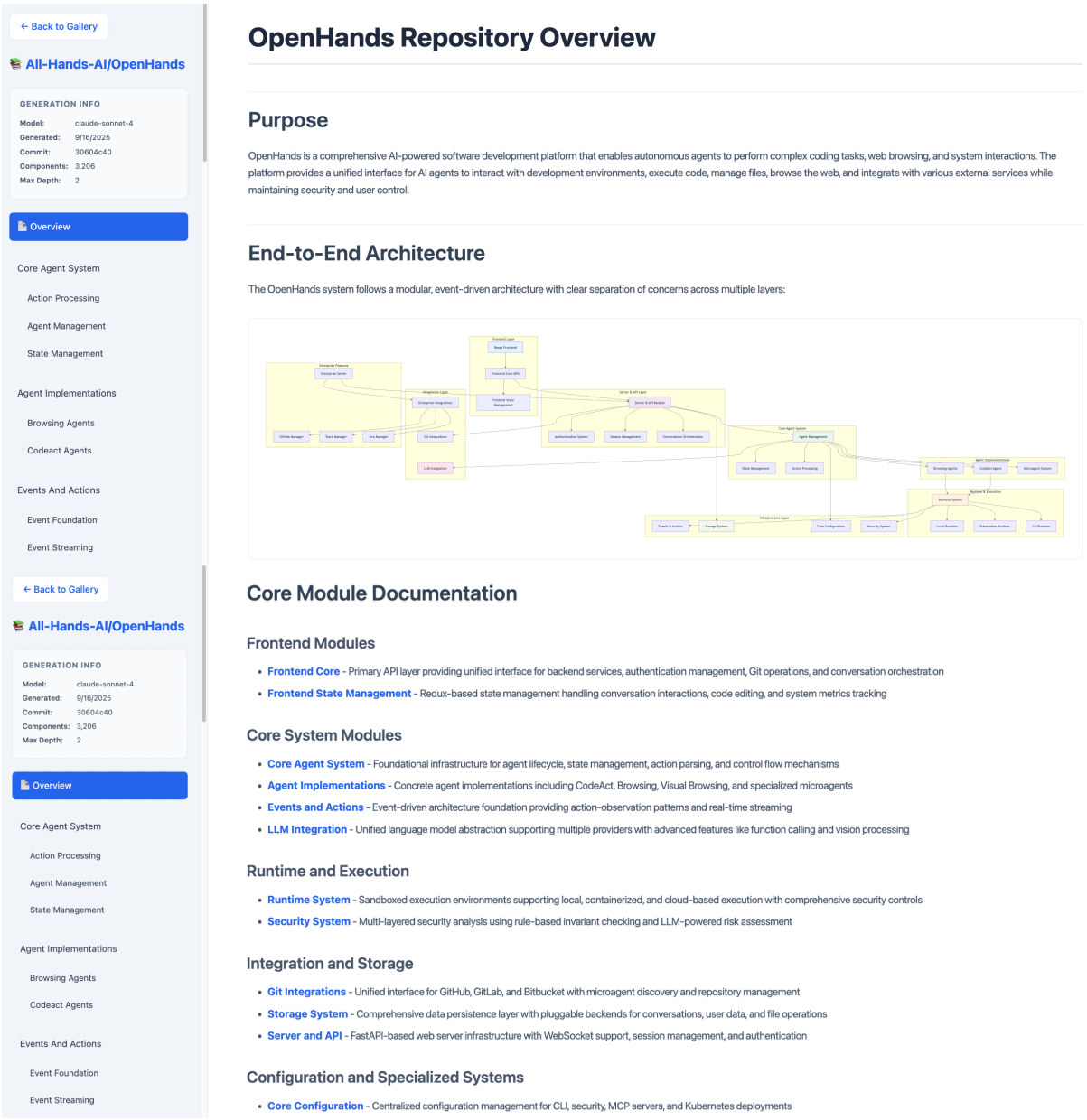

Figure 3: Example repository-level documentation generated by CodeWiki for the All-Hands-AI–OpenHands repository. The documentation includes a comprehensive overview, architectural diagrams showing the modular event-driven architecture, and hierarchical navigation of core components including the agent system, implementations, and event processing modules.

quality through few-shot learning and retrieval-augmented generation. ProConSuL (Lomshakov et al., 2024) incorporated project-level information via compiler analysis and call graphs, employing a two-phase training procedure to reduce hallucinations. RepoAgent (Luo et al., 2024) introduced comprehensive LLM-powered repository documentation, though it was limited in content diversity and architectural depth. DocAgent (Yang et al., 2025) advanced the field through multi-agent collaboration with topological processing, employing specialized agents that respect code dependencies to improve completeness and accuracy.

↪ *Limitations and Our Motivation:* Despite this progress, most approaches still target isolated snippets (e.g., functions or classes) and thus fail to capture the underlying architecture of the repository, as highlighted in Table 2. For a developer onboarding a completely new codebase, the primary challenge is not understanding what each individual function or class does, but rather grasping the high-level overview of the system, its dataflow and how different components interact. This underscores the need for an automatic methodology capable of gen-

erating documentation that spans from low-level details to high-level repository-wide perspectives, covering critical features and providing visual representations that support developer comprehension. Our approach, CodeWiki, addresses these limitations by generating true repository-level documentation while leveraging full codebase context and supporting multiple programming languages.

B.2 Repository-Level Documentation Systems

Several systems have addressed repository-level documentation generation with varying approaches and scopes:

Open-Source Alternatives **OpenDeepWiki** and **deepwiki-open** are community-developed tools that apply LLMs to entire repositories. However, as shown in Table 1, these approaches exhibit degraded performance on larger codebases (averaging 47.13% and 50.05% respectively), suggesting that simpler whole-repository prompting does not scale effectively to complex projects.

RepoAgent (Luo et al., 2024) aggregates function and class-level documentation into repository structures using dependency analysis. While effective for component cataloging, it does not perform hierarchical synthesis where parent modules summarize and contextualize their children.

codebase2tutorial[5] generates tutorial-style documentation using whole-repository prompting, focusing on educational walkthroughs rather than comprehensive architectural documentation.

LARCH (Koreeda et al., 2023) targets README generation specifically, addressing a narrower scope than full repository documentation.

Agent-as-a-Judge (Zhuge et al., 2024) includes an OpenWiki feature that leverages existing READMEs and applies LLM assessment, representing a different methodology focused on evaluation rather than generation.

CodeWiki's Distinguishing Characteristics CodeWiki differs from these approaches through: (1) true hierarchical synthesis where parent modules are generated by summarizing and contextualizing child documentation, (2) integrated visual artifacts (architecture diagrams, data flow visualizations) generated alongside textual content, (3) dynamic delegation enabling scalable processing of arbitrarily large repositories, and (4) cross-module reference management maintaining documentation coherence across module boundaries.

[5] https://code2tutorial.com/

Table 2: Comparison of different code summarization approaches. Repo-Level Doc. indicates whether the approach generates repository-level documentation output. Codebase Context specifies whether the codebase information is used as input to generate the documentation.

Approach	Repo-Level Doc.	Codebase Context	Multilingual
DocuMint	✗	✗	✗
ASAP	✗	✓	✓
ProConSul	✗	✓	✗
DocAgent	✗	✓	✗
RepoAgent	✗	✓	✗
CodeWiki	✓	✓	✓

B.2.1 Existing Evaluation Methods

Most existing works evaluate the quality of generated documentation using textual similarity metrics such as BLEU or ROUGE. These metrics measure the overlap between generated text and reference ground-truth documentation in terms of lexical or token-level similarity.

↪ *Limitations and Our Motivation:* While these methods provide a lightweight quantitative measure, they often fail to capture deeper semantic aspects of the documentation, such as clarity, correctness, and completeness in explaining the underlying code behavior. Recent studies (Roy et al., 2021; Evtikhiev et al., 2023) have shown that traditional metrics like BLEU and ROUGE do not consistently correlate with human judgment of documentation quality. This gap highlights the need for an assessment system that can fairly and accurately evaluate documentation, ensuring that the generated content truly supports developers in comprehending the system.

B.3 Rubric-based evaluation

A repository can be described through multiple representations, each using different structures and presentation styles while sharing the same underlying architecture. This diversity makes it difficult to evaluate open-ended outputs automatically, as no single metric can reliably evaluate which version is better. Although human evaluation provides trustworthy results, it is resource-intensive and slow. To overcome these limitations and facilitate rapid assessment, many benchmarks (Starace et al., 2025; Lin et al., 2025; Sirdeshmukh et al., 2025; Arora et al., 2025) often employ large language models (LLMs) as automated evaluators, using task-specific rubrics to judge the quality of

generated outputs. In our CodeWikiBench benchmark, we similarly construct a set of hierarchical rubrics based on the official documentation, providing a structured and fine-grained framework to evaluate the quality and completeness of generated repository-level documentation.

C Algorithm Details

Algorithm 1: Recursive Module Documentation

Input: Module tree T, dependency graph G
Output: Complete repository documentation

```
Initialize documentation workspace;
for each leaf module m in T do
    agent ← CreateAgent(m, G, T);
    while m not fully processed do
        agent.GenerateDoc(m.name);
        if delegation requested then
            sub ← agent.GetDelegationSpecs();
            UpdateModuleTree(T, m, sub);
            foreach new leaf m' in sub do
                Recursively process m';
            end
            ReviseDoc(m.name);
        end
    end
end
ProcessParentModules(T);
return Repository documentation;
```

D Rubric Reliability Assessment

To ensure the quality and trustworthiness of automatically generated evaluation rubrics, we developed a comprehensive reliability assessment framework that quantifies the consistency and structural integrity of rubrics produced by different language models. This assessment methodology provides empirical measures of rubric reliability across multiple dimensions.

Given a set of n rubrics $\{R_1, R_2, \ldots, R_n\}$ produced by different models for the same documentation corpus, we compute pairwise consistency scores across both semantic and structural dimensions.

D.1 Semantic Consistency Analysis

We evaluate semantic consistency by extracting all requirement texts from each rubric and compute semantic similarity using distributed representations. Let $T_i = \{t_1^{(i)}, t_2^{(i)}, \ldots, t_{m_i}^{(i)}\}$ denote the set of requirement texts extracted from rubric R_i, where m_i represents the total number of requirements in R_i.

For each requirement text $t_k^{(i)} \in T_i$, we identify its best semantic match in T_j using cosine similarity between embeddings:

$$\text{sim}_{\text{best}}(t_k^{(i)}, T_j) = \max_{t_\ell^{(j)} \in T_j} \cos(\mathbf{e}(t_k^{(i)}), \mathbf{e}(t_\ell^{(j)})) \tag{1}$$

where $\mathbf{e}(\cdot)$ represents the embedding function and $\cos(\cdot, \cdot)$ denotes cosine similarity. The bidirectional semantic similarity between rubrics is then:

$$\text{Sim}_{\text{semantic}}(R_i, R_j) = \frac{1}{m_i + m_j} \left(\sum_{k=1}^{m_i} \text{sim}_{\text{best}}(t_k^{(i)}, T_j) + \sum_{\ell=1}^{m_j} \text{sim}_{\text{best}}(t_\ell^{(j)}, T_i) \right) \tag{2}$$

This formulation ensures that both the coverage of requirements from R_i in R_j and vice versa are considered, providing a balanced measure of semantic overlap.

D.2 Structural Consistency Analysis

Structural consistency evaluates the architectural similarity between rubrics, focusing on hierarchical organization rather than content semantics. We define a comprehensive set of structural features for each rubric:

- **Maximum depth** (d): The deepest level in the rubric hierarchy
- **Total items** (N): The total number of rubric items across all levels
- **Weight distribution** (W): The frequency distribution of importance weights

The structural similarity between rubrics R_i and R_j combines multiple normalized metrics:

$$\text{Sim}_{\text{depth}}(R_i, R_j) = 1 - \frac{|d_i - d_j|}{\max(d_i, d_j, 1)} \tag{3}$$

$$\text{Sim}_{\text{items}}(R_i, R_j) = 1 - \frac{|N_i - N_j|}{\max(N_i, N_j, 1)} \quad (4)$$

For weight distribution similarity, we compute the overlap between normalized probability distributions:

$$\text{Sim}_{\text{weights}}(R_i, R_j) = \sum_{w \in W_i \cup W_j} \min(P_i(w), P_j(w)) \quad (5)$$

where $P_i(w)$ represents the normalized frequency of weight w in rubric R_i.

The overall structural similarity is computed as:

$$\text{Sim}_{\text{structural}}(R_i, R_j) = \frac{1}{3}\Big(\text{Sim}_{\text{depth}}(R_i, R_j) + \text{Sim}_{\text{items}}(R_i, R_j) \quad (6)$$

$$+ \text{Sim}_{\text{weights}}(R_i, R_j)\Big) \quad (7)$$

D.3 Overall Reliability Score

The overall reliability score aggregates consistency measures across all model pairs. For n models, we compute $\binom{n}{2}$ pairwise comparisons and derive summary statistics:

$$\text{Score}^{semantic}_{\text{reliability}} = \frac{1}{\binom{n}{2}} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \text{Sim}_{\text{semantic}}(R_i, R_j) \quad (8)$$

$$\text{Score}^{structural}_{\text{reliability}} = \frac{1}{\binom{n}{2}} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \text{Sim}_{\text{structural}}(R_i, R_j) \quad (9)$$

Table 3 presents the reliability assessment results across our benchmark repositories, demonstrating consistent rubric generation with average semantic reliability of 73.65% and structural reliability of 70.84%.

Table 3: Rubric Reliability Assessment: Semantic and Structural Consistency Scores

Repository	Semantic (%)	Structural (%)
All-Hands-AI–OpenHands	73.64	68.82
sveltejs–svelte	75.56	68.88
puppeteer–puppeteer	72.97	76.36
Unity-Technologies–ml-agents	75.95	71.22
elastic–logstash	73.13	68.22
wazuh–wazuh	72.20	67.62
electron–electron	72.12	74.79
Average	**73.65**	**70.84**

E Temporal Separation Details

Table 4 presents the specific commit information for each benchmark repository, demonstrating temporal separation from model training data.

Table 4: Repository commit information demonstrating temporal separation from model training data.

Repository	Language	Commit ID	Commit Date
OpenHands	Python	30604c4	Sep 13, 2025
svelte	JavaScript	be645b4	Aug 31, 2025
puppeteer	TypeScript	c1105f1	Aug 29, 2025
ml-agents	C#	4cf2f49	Aug 15, 2025
logstash	Java	895cfa5	Aug 29, 2025
wazuh	C	44b7cd3	Aug 30, 2025
electron	C++	828fd59	Aug 30, 2025

F Human Evaluation Pilot Study

To provide additional validation of our automated evaluation methodology beyond multi-model consensus, we conducted a pilot human study comparing CodeWiki and DeepWiki documentation outputs.

F.1 Study Design

Participants We recruited three participants with diverse professional backgrounds relevant to documentation consumption:

- **Business Analyst**: Represents stakeholders seeking high-level architectural understanding
- **AI Researcher**: Represents technical users evaluating documentation for research purposes
- **Software Engineer**: Represents developers using documentation for implementation guidance

None of the participants were prior contributors to the evaluated repositories, simulating the common scenario where developers approach unfamiliar codebases.

Evaluation Protocol For each of three repositories (OpenHands, svelte, puppeteer), participants:

1. Received access to the official codebase and its official documentation
2. Reviewed anonymized documentation outputs from both CodeWiki and DeepWiki (presentation order randomized)

3. Assessed which documentation better reflected: (a) correctness relative to official documentation, and (b) coverage of actual repository architecture

4. Provided a binary preference judgment

The evaluation was conducted blind, with system identities hidden from participants.

F.2 Results

Table 5: Pilot human evaluation results showing participant preferences between CodeWiki (CW) and DeepWiki (DW) across three repositories.

Repository	BA	AI	SE	CW Pref.
OpenHands	CW	CW	CW	3/3
svelte	CW	CW	DW	2/3
puppeteer	CW	DW	CW	2/3
Total	3/3	2/3	2/3	**7/9 (77.8%)**

Table 5 presents the evaluation results. CodeWiki was preferred in 7 out of 9 assessments (77.8%). Key observations:

Alignment with Automated Scores Human preferences correlate with the magnitude of automated evaluation improvements:

- **OpenHands** (+9.41% automated improvement): Unanimous CodeWiki preference (3/3)
- **puppeteer** (+18.54% automated improvement): Strong CodeWiki preference (2/3)
- **svelte** (+3.45% automated improvement): Closer preferences (2/3 for CodeWiki)

This alignment between human judgment and automated evaluation supports the validity of our LLM-based assessment methodology.

F.3 Limitations

We acknowledge several limitations of this pilot study:

- **Scale**: Three participants and three repositories provide preliminary validation but not statistical significance
- **Simplified Protocol**: Due to budget constraints, we used preference judgment rather than full rubric-based evaluation
- **Repository Selection**: The evaluated repositories may not represent the full diversity of our benchmark

Comprehensive human evaluation with larger participant pools, more repositories, and detailed rubric-based assessment represents important future work to further validate the benchmark methodology.

G Implementation Details

G.1 Model Configurations

Documentation Generation Agent

- Model: Claude Sonnet 4 (claude-sonnet-4-20250514)
- Temperature: 0.0
- Max tokens: 16384
- Context window: 200K tokens

Rubric Generation Agents

- Model 1: Claude Sonnet 4 (Temperature: 0.0)
- Model 2: Gemini 2.5 Pro (Temperature: 0.0)
- Model 3: Kimi K2 Instruct (Temperature: 0.0)
- Max tokens per generation: 32768

Judge Agents

- Judge 1: Gemini 2.5 Flash (Temperature: 0.0)
- Judge 2: GPT OSS 120B (Temperature: 0.0)
- Judge 3: Kimi K2 Instruct (Temperature: 0.0)
- Binary scoring: 0 (not satisfied) or 1 (satisfied)

G.2 Processing Parameters

- Module decomposition threshold: 32768 tokens per leaf module
- Maximum delegation depth: 3 levels