RepoScope: Bridging Physical Structure and Logical Flow for Intent-Driven Repository Documentation

Zezhou Yang[1], **Chaozheng Wang**[2], **Hailiang Huang**[3]
Peihua Zhang[3], **Ting Peng**[3], **Yuetang Deng**[3], **Chenxiong Qian**[1*]
[1]The University of Hong Kong, Hong Kong, China
[2]The Chinese University of Hong Kong, Hong Kong, China
[3]Tencent, Guangzhou, China
zezhouyang@connect.hku.hk, adf111178@gmail.com
eraserhuang@tencent.com, patphzhang@tencent.com
sakurapeng@tencent.com, yuetangdeng@tencent.com, cqian@hku.hk

Abstract

Repository-level documentation generation is vital for human comprehension and autonomous agents, yet existing approaches fail to align high-level architectural intent with granular implementation details, resulting in documentation that lacks both global coherence and local fidelity. To address this, we propose REPOSCOPE, a repository documentation generation framework that achieves alignment via two synergistic innovations. Specifically, REPOSCOPE employs Physical-Logical Partitioning by jointly modeling directory structures and call graphs, effectively bridging the gap between physical layouts and logical dependencies while creating coherent documentation units. Within this partitioned space, we introduce Seeded Architectural Intent Lift, a mechanism that projects architectural intent top-down to steer granular analysis, which is then revised and lifted bottom-up via code evidence to ensure global-local consistency. Our experimental results on CodeWikiBench demonstrate state-of-the-art performance in aligning with human-authored documentation. Furthermore, the validation on real-world industrial systems shows that REPOSCOPE yields superior correctness and completeness in empowering autonomous agents for CLI extraction compared to existing baselines.

1 Introduction

High-quality software documentation serves as a vital bridge between high-level architectural design and low-level code implementation, providing essential context for both human developers [1–3] and autonomous agents to comprehend the repository [4–6]. However, the growing adoption of LLM-assisted development is accelerating the pace at which codebases are generated [7, 8], modified [9–11], and integrated [12–14]. As repositories expand and evolve more rapidly, manual documentation maintenance becomes costly and prone to drift, motivating robust automated documentation solutions.

Repository-level documentation generation primarily follows a decomposition-and-aggregation paradigm. This paradigm partitions a repository into local code units, such as files, classes, and functions [15], summarizes these units with dependency-derived context [16–18], and composes the resulting summaries into module-level and repository-level documentation [19–21]. While this paradigm preserves fine-grained implementation facts, existing methods are primarily driven by call dependencies and semantic similarity, frequently overlooking architectural intent, such as architectural decomposition and the system-level roles of implementation units. As a result, the generated documentation can be locally informative but remains globally under-modeled.

*Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

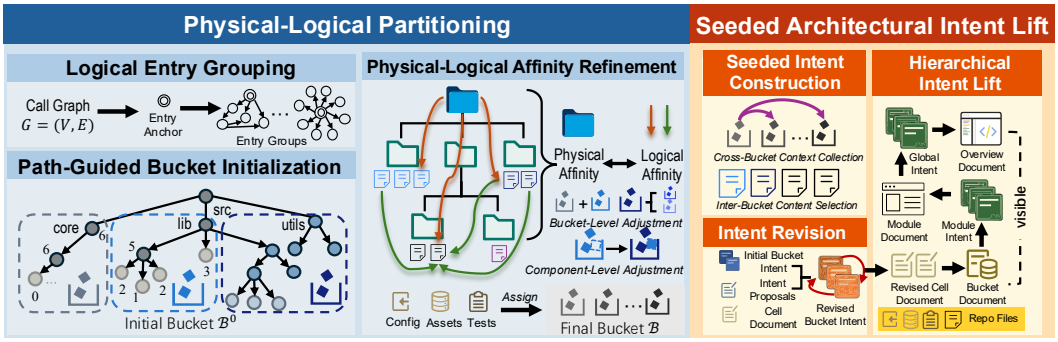


Figure 1: Overview of the REPOSCOPE pipeline. The blue region shows Physical-Logical Partitioning, where entry anchors are grouped by call-graph reachability, projected onto the repository path tree to initialize buckets, and refined through physical-logical affinity at both bucket and component levels. The orange region shows Seeded Architectural Intent Lift, where each bucket receives an evidence-grounded intent seed from cross-bucket context, revises this intent through cell-level code observations, and is lifted into bucket, module, and repository-level documentation.

This limitation stems in part from treating repositories as collections of local semantics rather than organized design artifacts. During software development, architectural design is gradually instantiated into the repository structure, shaping how functionality is decomposed and responsibilities are assigned [22, 23]. However, physical layouts reflect static organizational conventions, while logical dependencies reveal realized interactions that often cross these boundaries. Consequently, such intent cannot be recovered from the physical structure alone. Furthermore, the automated extraction and propagation of architectural intent remains largely implicit and unexplored. Specifically, recovering this intent and leveraging it for documentation generation presents two key challenges:

(1) Physical–Logical Misalignment. Physical layouts and logical dependencies provide complementary but often inconsistent signals for repository decomposition. Features may span multiple directories, collaborations may cross module boundaries, and dependency graphs may be distorted by utilities or other cross-cutting relations. Relying on either view alone therefore yields documentation units that miss either architectural boundaries or logical coordination. The challenge is to reconcile physical locality with logical cohesion, producing units that remain grounded in the repository structure while preserving the interactions needed for architectural-intent recovery.

(2) Contextual Isolation of Architectural Intent. Even coherent documentation units cannot reveal architectural intent by themselves, which is rarely explicit in local code. The system-level role of a component depends on its broader repository context, so similar local behaviors may serve different design purposes. The documentation generated from isolated code summaries may describe what each unit does, but fails to explain why it exists, what responsibility it realizes, and how it supports the overall architecture. The challenge is to infer latent intent from contextual implementation evidence and propagate it consistently across different levels of documentation.

To address these challenges, we propose REPOSCOPE (Figure 1), a repository documentation generation framework that aligns architectural intent with implementation details to produce documentation that is both globally coherent and locally grounded. To handle physical-logical misalignment, REPOSCOPE introduces a **Physical-Logical Partitioning (PLP)** strategy that reorganizes a repository into document-oriented code units *Buckets*. These buckets provide a structured substrate for downstream documentation synthesis and serve as the fundamental units for architectural intent generation, while preserving both navigability and implementation-level coherence. Rather than relying solely on directory boundaries or dependency connectivity, our strategy integrates physical locality in the file hierarchy with logical locality in the call graph. This approach groups functionally related components while decoupling broadly shared utilities that would otherwise obfuscate module boundaries.

To mitigate contextual isolation of architectural intent, REPOSCOPE further introduces **Seeded Architectural Intent Lift (SAIL)**, which turns the reconstructed code units *Buckets* into an intent-grounded documentation hierarchy through a seed–revise–lift process. First, SAIL constructs an initial architectural intent seed for each bucket by interpreting its role together with representative contextual evidence. Second, guided by this seed, SAIL uses concrete code observations to verify, refine, or

correct the initial intent, and to generate fine-grained documentation. Finally, the revised bucket intents are lifted upward to organize bucket, module, and repository-level documentation, ensuring that higher-level descriptions are grounded in implementation evidence while local summaries remain aligned with architectural roles. This process enables REPOSCOPE to document not only what each component does, but also why it exists and how it contributes to the broader system design. The contribution lies in coupling PLP's responsibility-oriented structural scope with SAIL's revisable architectural hypotheses: PLP determines where intent is inferred, while SAIL governs how it guides local interpretation, is corrected by code evidence, and propagates through the hierarchy.

We conduct experiments on CodeWikiBench to validate the effectiveness of REPOSCOPE, where documentation quality is evaluated using rules extracted from human-authored documentation. Across 21 repositories, REPOSCOPE achieves the highest coverage, showing that it better aligns with the key information captured by human-written references. To further examine practical utility, we apply REPOSCOPE to three industrial closed-source systems provided by our collaborators and use the generated documentation to help autonomous agents for CLI extraction. Compared with existing baselines, REPOSCOPE enables agents to extract CLI with higher correctness and completeness, demonstrating the usefulness of REPOSCOPE for downstream software-engineering tasks.

2 Related Work

2.1 Code Summarization

Source code summarization generates concise natural-language descriptions to support program comprehension and software maintenance. Early neural approaches mainly relied on explicit structural inductive biases to model program semantics, such as AST representations [24, 25] and symbolic dataflow information [26, 27]. With the emergence of LLMs, recent work has improved code summarization through general prompting strategies, including zero-shot [28] and few-shot prompting [29, 30]. Reasoning-oriented methods, such as Chain-of-Thought [31] and critique-based prompting [32, 33], further enhance summary quality by eliciting intermediate reasoning or iterative self-correction. Beyond generic prompting, LLM-based code summarization methods also incorporate code-specific signals, including project-level context, domain knowledge, and API information [34–36], to better capture program behavior. However, most existing methods remain centered on isolated code snippets, and therefore cannot fully support repository-level documentation, which requires reasoning over global architecture, cross-file dependencies, and architectural intent.

2.2 Repository-Level Documentation Generation

Repository-level documentation captures the global architectural context of a codebase beyond isolated snippets. Current research follows three technical paradigms: **hierarchical aggregation** (e.g., HGEN [17], HMCS [21]), which stacks summaries from functions up to modules [20]; **dependency-aware approaches** (e.g., RepoAgent [16], RepoSummary [18]), which incorporate call graphs and directory hierarchies [18]; and **multi-agent frameworks** (e.g., CodeWiki [15], UserTrace [37]), which manage project-scale complexity through recursive processing [15]. Beyond academia, industrial closed-source systems like DeepWiki [38] have demonstrated practical efficacy in large-scale maintenance [15], while products such as Claude Code [6] showcase how holistic documentation empowers AI agents with enhanced reasoning and navigation [37]. These advancements also drive a transition toward LLM-based metrics [39, 40] for assessing documentation coherence [15]. Nevertheless, most existing methods still rely on unidirectional bottom-up synthesis [18]. This paradigm often leads to structural misalignment between logical dependencies and physical layouts, yielding fragmented documentation that fails to preserve the original architectural intent.

3 Method

Given a software repository $\mathscr{R}$, REPOSCOPE aims to generate a hierarchical documentation set $\mathcal{D}$ that is structurally coherent and semantically accurate. As shown in Figure 1, REPOSCOPE operates in two parts: **Physical-Logical Partitioning** (PLP, the blue-shaded region) transforms the unstructured codebase into documentation units *Bucket*, and **Seeded Architectural Intent Lift** (SAIL, the orange-shaded region) generates an intent-grounded documentation hierarchy. Ultimately, the structured

code units and their related metadata from PLP serve as the deterministic grounding for SAIL to verify architectural intents and synthesize implementation-consistent documentation.

3.1 Physical-Logical Partitioning

Physical-Logical Partitioning (PLP) decomposes a repository into coherent functional units for downstream documentation generation. We model a repository as $\mathscr{R} = (V, G, \pi)$, where V denotes code components (e.g., functions, classes), $G = (V, E)$ is a directed call graph over implementation dependencies, and $\pi : V \to \mathcal{P}$ maps each component to its source path. This formulation exposes two complementary but often misaligned views: call graph captures logical locality, while directory tree captures physical locality. PLP reconciles these views to avoid both over-coupling through shared utilities and over-fragmentation caused by directory boundaries.

The procedure consists of three stages: *Logical Entry Grouping*, *Path-Guided Bucket Initialization*, and *Physical-Logical Affinity Refinement*. First, PLP derives logical groups based on call reachability from entry anchors identified within the call graph. It then projects these groups onto the directory tree to initialize path-rooted bucket candidates. Finally, it refines bucket boundaries and component memberships by harmonizing logical affinity with directory locality. The output is a bucket set $\mathcal{B} = \{B_1, \ldots, B_K\}$, with globally shared or unattributable components retained as auxiliary contents.

3.1.1 Logical Entry Grouping

The first stage identifies logical entry groups from the call graph. We define an entry anchor as a source-like component in the call graph, characterized by sparse incoming calls from other internal components and non-empty outgoing reachability to implementation components. Such anchors serve as call origins for downstream implementation regions. We first extract a raw anchor set $\widetilde{\mathcal{A}}$ from G using these structural criteria. As the initial set is a high-recall approximation that may contain both noise and redundancy, we refine it using a two-step normalization procedure.

The first normalization step prunes anchors that cannot serve as stable function call starting points, including isolated, deprecated, and test-related candidates. The detailed filter process is described in Appendix A.1.2. This step yields a retained anchor set $\mathcal{A} \subseteq \widetilde{\mathcal{A}}$.

The second normalization step consolidates redundant anchors. Each anchor $a \in \mathcal{A}$ is represented by its downstream call-reachability profile:

$$\mathcal{R}(a) = \{v \in V \mid v \text{ is reachable from anchor } a\}, \qquad R(a) = \{(v, d(a, v)) \mid v \in \mathcal{R}(a)\}, \quad (1)$$

where $d(a, v)$ is the minimum call-graph distance under the breadth-first traversal convention, with $d(a, a) = 0$. An anchor is merged into another as a group when its downstream profile is substantially contained in the other's profile; anchors with distinguishable downstream profiles are kept separate to preserve functional coverage. The detailed consolidation procedure is described in Appendix A.1.3.

Anchors that are not merged remain as singleton groups. Therefore, after pruning and consolidation, we obtain logical entry groups $\mathcal{S}$. Each group $s \in \mathcal{S}$ contains one or more normalized anchors and represents a consolidated functional origin. The same reachability representation is carried forward by aggregating member-anchor profiles and keeping the shortest depth to each component:

$$\mathcal{R}(s) = \bigcup_{a \in s} \mathcal{R}(a), \qquad d(s, v) = \min_{a \in s:\, v \in \mathcal{R}(a)} d(a, v), \qquad R(s) = \{(v, d(s, v)) \mid v \in \mathcal{R}(s)\}. \quad (2)$$

Each component is linked to the entry groups that reach it with the corresponding call depths. These group-level reachability profiles provide the functional basis for subsequent bucket initialization.

3.1.2 Path-Guided Bucket Initialization

Given the logical entry groups $\mathcal{S}$, the second stage constructs an initial bucket set over the repository path tree. The path tree is used as a structural scaffold rather than a hard directory partition. Each entry group is assigned to the path-tree node containing its anchor source file.

For an entry group s, let $\Phi(s)$ denote its effective reachable component set, obtained by removing broadly shared infrastructure components from its reachable components while retaining the anchor/member components of the consolidated group. For a path-tree node u, let $\mathcal{S}(u)$ be the entry

groups placed in its subtree. We define the node footprint as:

$$M(u) = \bigcup_{s \in \mathcal{S}(u)} \Phi(s). \tag{3}$$

Since $M(u)$ may still contain components shared by multiple logical entry groups, we score each node with a sharing-aware complexity measure rather than raw footprint cardinality. This score discounts components reused by multiple logical entry groups while preserving anchor-specific implementation coverage, providing a reliable indicator for the following partition.

The path tree is recursively partitioned according to a repository-adaptive complexity budget derived from entry-group footprint distributions. At each node u, the sharing-aware footprint complexity serves as a proxy for functional density under the corresponding path region. Regions whose footprint complexity exceeds the adaptive upper budget are split along child path regions, whereas under-sized regions are deferred for local merging within the parent neighborhood when the merged footprint remains within budget and coherent. All detailed implementations are provided in Appendix A.2.

This produces a structured initial $\mathcal{B}^0$ that respects the physical layout while using logical reachability to determine boundary granularity. Importantly, $\mathcal{B}^0$ is not intended to serve as the final component assignment. Instead, it provides path-rooted bucket candidates whose boundaries and component memberships are further adjusted in the subsequent cross-structure refinement stage.

3.1.3 Physical-Logical Affinity Refinement

The initial buckets $\mathcal{B}^0$ are rooted in the path tree and may still inherit mismatches between directory layout and functional structure. The third stage refines these buckets by using physical-logical affinity at two granularities: bucket boundaries and component memberships.

For two buckets B_i and B_j, we compute a logical affinity $\ell(B_i, B_j)$ from their overlap in entry-group reachability and the consistency of their call-depth profiles, and a physical affinity $p(B_i, B_j)$ from their proximity in the repository path tree and path-feature signatures. Since either view can be unreliable in a local region, we do not combine them with a fixed global weight. Instead, each view receives a local confidence score, denoted by q^{ℓ}_{ij} and q^{p}_{ij}, estimated from signal coverage, support margin, and call/path density of the corresponding view. These scores define local arbitration weights:

$$\tau^{\ell}_{ij} = \frac{q^{\ell}_{ij} + \epsilon}{q^{\ell}_{ij} + q^{p}_{ij} + 2\epsilon}, \qquad \tau^{p}_{ij} = \frac{q^{p}_{ij} + \epsilon}{q^{\ell}_{ij} + q^{p}_{ij} + 2\epsilon}. \tag{4}$$

The bucket-level affinity is $A(B_i, B_j) = \tau^{\ell}_{ij}\ell(B_i, B_j) + \tau^{p}_{ij}p(B_i, B_j)$. This affinity is used to decide whether bucket boundaries should be preserved, merged, or further decomposed. A candidate edit is accepted only if it does not increase a repository-level objective $\mathcal{J}$, defined as a weighted sum of normalized penalties for logical cohesion, physical locality, and engineering practicality.

After bucket-level adjustment, we apply similar physical-logical arbitration at the component granularity. For a boundary component $v \in B_i$, we evaluate candidate destination buckets B_j by replacing bucket-pair affinities with component-to-bucket fit scores. The logical fit $\ell(v, B_j)$ measures whether v shares the canonical entry-group reachability signature of components in B_j, with a call-depth discount, while the physical fit $p(v, B_j)$ measures the proximity between v's file path and the path region of B_j. Using the arbitration weights τ^{ℓ}_{ij} and τ^{p}_{ij}, we define $A_{i \to j}(v, B_k) = \tau^{\ell}_{ij}\ell(v, B_k) + \tau^{p}_{ij}p(v, B_k), k \in \{i, j\}$. A reassignment is accepted only when the best destination provides a sufficient affinity gain over the current bucket and does not increase $\mathcal{J}$. All the details are given in Appendix A.3.

The refinement yields the buckets whose boundaries reconcile logical reachability with physical locality. Although the primary partitioning strategy is based on implementation components, repositories also contain artifacts that are not naturally modeled as call-graph nodes, such as configuration and asset files. PLP treats these files as bucket-attached repository artifacts rather than as primary documentation units. Specifically, each artifact is assigned to the bucket whose components, paths, references, or semantic cues provide the strongest support. Moreover, tests that primarily specify or exercise product behavior are attached to the corresponding implementation bucket, while reusable test frameworks or testing infrastructure may form their own bucket when they constitute a coherent implementation region. The assignment details can be found in Appendix A.4.

The final bucket set $\mathcal{B}$ therefore contains both implementation components and their assigned repository artifacts. Compared with the path-guided initialization, $\mathcal{B}$ can cross directory boundaries when logical evidence is reliable, while avoiding the over-coupling of unrelated regions through shared utilities or infrastructure components.

3.2 Seeded Architectural Intent Lift

Seeded Architectural Intent Lift (SAIL) takes the bucket set $\mathcal{B}$ from PLP, including implementation components and assigned repository artifacts, as its structural substrate and turns it into an intent-grounded documentation hierarchy. A bucket marks a coherent implementation region, but leaves its architectural intent unspecified. To bridge this gap, SAIL assigns each bucket an initial description of its architectural role, connects the description to supporting evidence, and updates it when code-level observations suggest a more accurate interpretation.

The procedure consists of three stages: *Seeded Intent Construction*, *Grounded Intent Revision*, and *Hierarchical Intent Lift*. First, SAIL constructs context from related buckets and selects representative evidence to derive an initial intent seed for each bucket. It then generates fine-grained documentation under these seeds, using code observations to propose and adjudicate evidence-backed intent revisions. Finally, it freezes the revised intent state and uses it to generate bucket, module, and repository documentation. The output is a documentation hierarchy $\mathcal{D}$ generated from evidence-revised architectural intent rather than from independent summaries of code regions.

3.2.1 Seeded Intent Construction

This stage maps the bucket partition into an initial intent–evidence representation. For each bucket $B_k \in \mathcal{B}$, SAIL derives a structured intent seed $\tilde{z}_k$ that records the bucket's provisional architectural role, expected responsibilities, supporting evidence, and unresolved questions. The seed is generated with reference to structural and semantic context rather than from the bucket in isolation.

To avoid isolated interpretation, SAIL expands the context for each bucket to include structurally or semantically related buckets while keeping the original bucket partition fixed. The main context signals come from physical-logical relations between buckets. In addition, SAIL uses the artifact and test links produced by PLP when they provide relevant evidence, such as configuration constraints.

Within this local context, SAIL performs model-guided top-down selection over the target bucket's components, the components of related buckets, and the artifacts or tests attached by PLP. It selects a bounded set of role-revealing materials, consisting primarily of code snippets and supplemented when useful by artifact excerpts or summaries and test-related evidence.

The resulting seed provides the semantic frame for downstream documentation generation, while remaining subject to revision when later code-level observations expose a more accurate interpretation. Algorithm 1 summarizes this construction procedure, with details in Appendix A.5.

3.2.2 Grounded Intent Revision

SAIL grounds the initial bucket intent seeds through bottom-up cell-level documentation and revises them only after aggregating cell-level evidence. Specifically, cells are fine-grained units within a bucket, constructed to group a dependency-coherent subset of code that can be processed within the LLM context budget. Each cell serves as both a local documentation unit and an observation unit: it exposes concrete behavior that can support, refine, or challenge the bucket-level interpretation.

For each cell C_j assigned to bucket $B_{b(j)}$, the generator documents the cell contents with the initial bucket seed $\tilde{z}_{b(j)}$ as a provisional semantic scaffold, producing a cell document d_j. Cells are processed in dependency order so that prerequisite or shared cells are documented before dependent cells. The resulting cell document d_j may include an evidence-backed revision proposal when the observed code indicates that the current bucket interpretation is incomplete, overly broad, or inaccurate. Revision proposals are optional and must be grounded in concrete code observations, so a cell document can only flag a candidate change for later adjudication rather than revise a bucket intent by itself.

For each bucket B_k, the proposals are first filtered by concrete code evidence. Generated summaries can provide context but cannot be the sole evidence for a revision. The remaining proposals are then evaluated jointly against the generated cell documents and their cited code observations. The

adjudicator accepts, rejects, or defers each proposal; accepted revisions are merged into a revised intent $\hat{z}_k$, while buckets without accepted revisions retain their initial seed. Thus, bucket intent is revised through aggregated cell evidence rather than by any single local observation.

To maintain consistency between revised intents and generated cell documentation, SAIL applies targeted replay, i.e., selectively rerunning documentation generation for affected cells. When an accepted revision changes the bucket-level intent, the cell documents for the proposing and affected cells are regenerated under the revised intent $\hat{z}_k$. Appendix A.6 provides implementation details on cell construction, revision adjudication, and targeted replay.

3.2.3 Hierarchical Intent Lift

After grounded revision, SAIL freezes the revised bucket intents $\{\hat{z}_k\}_{k=1}^{K}$ and the replay-consistent cell documents. The lift stage then constructs documentation from bottom to top. It first generates a bucket document for each B_k by aggregating the cell documents assigned to the bucket under the revised intent $\hat{z}_k$. The intent provides the bucket-level organization, while the cell documents provide grounded descriptions of local behavior. While generation primarily relies on the validated lower-level documents, the generator may request targeted source context for relevant files or components when the available evidence is insufficient to support a specific claim.

SAIL next groups bucket documents into modules and derives a module intent for each group. The module intent summarizes the common responsibility and interaction pattern among its member buckets. Conditioned on this intent, the module generator produces module-level documentation from the corresponding bucket documents and, when needed, supporting cell documents or targeted source context. This step lifts revised bucket-level interpretations into higher-level architectural units without changing the finalized bucket intents, thereby preserving the evidence-traceable foundation.

Finally, SAIL constructs the repository-level documentation from the integrated module and bucket hierarchy. Before generating the overview, residual units such as shared components and uncovered or orphaned regions are assigned into the documentation tree. The final overview is generated from this integrated tree, the finalized intent state, and any targeted source context needed to substantiate repository-level claims. The result is a documentation hierarchy $\mathcal{D}$ that lifts grounded intent from cells to buckets, from buckets to modules, and from modules to the repository overview, producing a coherent architectural narrative rooted in validated code observations.

4 Experiment

4.1 Setup

Benchmarks and Tasks. We evaluate REPOSCOPE on both public and industrial settings. For public evaluation, we use CodeWikiBench, a repository-level documentation benchmark that assesses whether documentation captures architectural structure, implementation logic, and usage details. Detailed metric definitions and experimental settings are provided in Appendix B.

For real-world evaluation, we use three anonymized industrial systems provided by our collaborators, denoted as System 1, System 2, and System 3. All three systems are production C++ codebases selected from larger business platforms. Each system contains the core business logic together with the supporting modules required for compilation and testing, forming a minimal self-contained codebase for the corresponding functionality. This setting is substantially different from evaluating isolated repositories or individual files, as the agent must understand large-scale dependencies, interface boundaries, build assumptions, and domain-specific implementation patterns.

Appendix B.2 provides detailed statistics of the three anonymized industrial systems. They contain 23K–25K files, 4.4K–4.8K directories, and up to 1.06GB of code artifacts. Each system includes more than 6.6K C/C++ source files, over 15K header files, and more than 1.2K protocol definition files. These characteristics make them suitable for evaluating whether repository-level documentation can support downstream agentic program understanding in realistic industrial environments. Inspired by CLI Anything[1], we design an agent-oriented CLI extraction task: given an expert-written `HARNESS.md`, an agent built on the Claude Code framework inspects the codebase and extracts executable CLI-accessible APIs.

[1] `https://github.com/HKUDS/CLI-Anything`

Table 1: Main comparison results on CodeWikiBench. We compare REPOSCOPE with DeepWiki, OpenDeepWiki, and CodeWiki across 7 core repositories. Score denotes the Quality Score, and Cov. denotes Coverage. The best results are highlighted in **bold**.

Repository	Language	LOC	DeepWiki		OpenDeepWiki		CodeWiki		REPOSCOPE	
			Score	Cov.	Score	Cov.	Score	Cov.	Score	Cov.
OpenHands	Python	229k	70.41	46/67	35.54	26/67	59.91	40/67	**88.03**	**58/67**
Svelte	JavaScript	124k	78.83	77/96	47.80	46/96	68.51	76/96	**88.19**	**80/96**
Puppeteer	TypeScript	136k	57.94	46/82	26.60	20/82	56.28	46/82	**76.60**	**58/82**
ML-Agents	C#	86k	59.43	28/46	50.10	19/46	53.51	24/46	**72.25**	**34/46**
Logstash	Java	117k	57.04	32/57	38.66	20/57	53.42	31/57	**62.98**	**38/57**
Wazuh	C	1.4M	57.09	25/46	37.90	15/46	51.61	21/46	**67.48**	**35/46**
Electron	C++	184k	44.58	43/92	20.78	17/92	42.30	48/92	**61.63**	**59/92**

Evaluation Protocol. Following CodeWikiBench, we adopt rubric-based LLM-As-Judge, rather than n-gram overlap metrics. We report **Quality Score** (%), a weighted aggregate score over repository-specific rubric trees, and **Coverage**, the fraction of satisfied leaf-level rubric requirements. For the CLI extraction task, we measure **Completeness** by the number of valid functional APIs discovered, and **Correctness** by executing the extracted APIs under expert-specified test conditions.

Baselines. We compare REPOSCOPE with three representative repository documentation systems: **DeepWiki** [38], a commercial hierarchical summarization system; **CodeWiki** [15], a state-of-the-art open-source method based on dependency-aware bottom-up aggregation; and **OpenDeepWiki** [41], a top-down system that provides documentation by selecting and summarizing important files.

Model Configuration. For CodeWikiBench, we follow CodeWiki and use K2-Instruct as the unified backbone for both generation and evaluation. For the industrial CLI extraction task, the documentation of three systems is also generated by K2-Instruct. But for the CLI extraction pipeline, we use Claude-Opus-4.6 within the Claude Code and keep the default Claude Code configuration.

4.2 Evaluation on CodeWikiBench

Table 1 presents the comparative results of REPOSCOPE against three representative techniques across seven repositories with diverse languages, sizes, and architectural styles.

Overall Performance. Overall, REPOSCOPE achieves the highest observed score and coverage on every core repository in the reported run, demonstrating the effectiveness and robustness of our repository-level documentation framework. On average, REPOSCOPE obtains a score of 73.88%, outperforming the strongest baseline, DeepWiki at 60.76%, by 13.12 percentage points. The improvement is consistent across repositories: for example, REPOSCOPE improves over DeepWiki by 17.62 percentage points on OpenHands and 18.66 percentage points on Puppeteer. In addition, REPOSCOPE achieves the highest coverage on every repository, covering 362/486 documentation targets overall, compared with 297/486 for DeepWiki and 286/486 for CodeWiki. These results indicate that REPOSCOPE improves both documentation quality and completeness.

Scalability. REPOSCOPE remains effective on large-scale repositories. Wazuh is the largest of the seven core repositories, with 1.4M LOC, making it a stress test for repository-level documentation methods. On Wazuh, REPOSCOPE achieves 67.48% and covers 35/46 documentation targets, improving over the strongest baseline DeepWiki by 10.39 percentage points and 10 additional covered targets. In contrast, CodeWiki and OpenDeepWiki obtain 51.61% and 37.90%, respectively. This shows that the advantage of REPOSCOPE persists under substantially larger codebases.

Cross-Language Generalization. The gains of REPOSCOPE are stable across languages and programming paradigms. It performs strongly on dynamic or web-oriented projects such as OpenHands and Svelte, reaching 88.03% and 88.19%, respectively. It also improves results on statically typed and desktop framework, e.g., achieving 61.63% on Electron in C++, which is 17.05 percentage points higher than DeepWiki. These results demonstrate that REPOSCOPE generalizes across heterogeneous repositories rather than being tailored to a specific language or project type.

Generalization on Extended Benchmarks. We further evaluate REPOSCOPE on all 21 repositories in CodeWikiBench. As reported in Appendix C.1, REPOSCOPE achieves state-of-the-art performance on every repository, confirming that its advantage extends beyond the seven repositories in the

Table 2: Results on the real-world CLI extraction task. Completeness is reported as the number of discovered APIs over expert-identified APIs, while correctness denotes the number of APIs passing expert-specified execution tests over extracted APIs. Comp. denotes the Completeness, and Corr. denotes the Correctness.

Setting	System 1 (97 APIs)		System 2 (135 APIs)		System 3 (50 APIs)		Overall	
	Comp.	Corr.	Comp.	Corr.	Comp.	Corr.	Comp.	Corr.
Raw Code	18/97	18/18	38/135	38/38	23/50	4/23	79/282	60/79
OpenDeepWiki	47/97	41/47	55/135	55/55	38/50	4/38	140/282	100/140
CodeWiki	58/97	52/58	92/135	92/92	40/50	5/40	190/282	149/190
REPOSCOPE	71/97	69/71	110/135	109/110	46/50	17/46	227/282	195/227

main comparison. The benchmark covers diverse software ecosystems, including data systems, frontend frameworks, developer tools, and AI applications. For example, REPOSCOPE obtains 99.05% on Trino and 91.88% on Storybook. These results indicate that REPOSCOPE generalizes across repositories with different scales, languages, and architectural styles.

Additional Analysis. To further understand the sources of these improvements, we examine the role of adaptive intent revision and the potential effect of documentation length. Disabling revision and replay while retaining intent seeding and hierarchical lifting reduces the average score from 73.88 to 58.92 (Appendix C.2.2). Meanwhile, PLP-only produces nearly the same amount of final documentation but scores 15.99 points lower, while DocAgent generates 37.0% more text yet scores 17.51 points lower (Appendix C.2.4). Additional cross-backbone experiments examine the sensitivity of these gains to model capability (Appendix C.3.1).

4.3 Evaluation on Real-World Systems

We evaluate REPOSCOPE on the three industrial systems described in Section 4.1. Table 2 compares different documentation in terms of their utility for the CLI extraction task. We exclude DeepWiki because the industrial codebases are confidential, while DeepWiki supports only public repositories.

Overall Comparison. Table 2 shows that repository-level documentation substantially improves API discovery and execution on confidential industrial systems. With only raw code, the agent discovers 79 out of 282 expert-identified APIs and successfully executes 60 of them, whereas REPOSCOPE discovers 227 APIs and executes 195 successfully, outperforming both OpenDeepWiki and CodeWiki. This indicates that the main bottleneck is not source-code access alone, but the lack of an organized understanding of the repository: raw code exposes implementation details but offers limited guidance on user-facing functionality, API grouping, and calling relationships. The stronger aggregate results of REPOSCOPE further indicate that repository-level documentation aligned with both the physical file structure and the logical organization of the codebase provides more effective guidance for downstream agentic extraction than generic automatically generated documentation.

System-Level Analysis. The gains vary across systems because their API usage patterns differ. For System 1, the API structure is less regular, and most APIs are exposed as standalone entries without heavy interdependency; hence, the dominant difficulty is locating the right functional entry rather than composing calls. REPOSCOPE increases completeness from 18/97 to 71/97, and the number of APIs that are both discovered and successfully executed grows from 18 to 69, compared with 41 for OpenDeepWiki and 52 for CodeWiki. This suggests that the documentation helps the agent identify the right entry even when the codebase lacks a uniform API style.

System 2 mainly consists of direct single-API invocations, so correctness is high across settings and the main challenge is completeness. REPOSCOPE discovers 110/135 APIs, outperforming CodeWiki's 92/135 and raw code's 38/135, showing that, for systems with direct API usage, repository-level documentation mainly improves discovery coverage rather than changing invocation difficulty.

System 3 is more challenging because tasks often require multiple APIs to be composed together. Although baselines can identify some relevant APIs, they rarely execute them successfully: for example, CodeWiki achieves 40/50 completeness but only 5 extracted APIs execute. REPOSCOPE raises completeness to 46/50 and increases the number of successfully executed APIs from 5 to 17—more than a $3\times$ improvement in the absolute count of executable APIs over the strongest

baseline. This gap shows that simply locating APIs is insufficient for systems whose functionality is exposed through multi-step logic. The improvement suggests that REPOSCOPE captures not only API locations but also the calling relationships needed for execution.

The remaining gap also reflects a limitation of our downstream evaluation: the CLI extraction agent is driven by the same `HARNESS.md`-based Claude Code across all settings, rather than by a task-specific pipeline optimized for multi-step API composition. Designing such workflows is orthogonal to our focus on repository-level documentation generation and is left for future work.

To complement the agent-based evaluation, we also report descriptive evidence from a separate industrial developer study on test-case construction, where RepoScope-assisted developers showed lower observed median completion times than the Raw Code and CodeWiki groups (Appendix D).

5 Conclusion

We introduce REPOSCOPE, a repository-level documentation generation framework that aligns architectural intent with code implementation. REPOSCOPE addresses physical–logical misalignment through Physical-Logical Partitioning, which reconciles directory structure with call-graph dependencies, and mitigates the contextual isolation of architectural intent through Seeded Architectural Intent Lift. Experiments on CodeWikiBench show state-of-the-art alignment with human-authored documentation, and validation on industrial systems demonstrates that the generated documentation can improve agent performance for CLI extraction tasks. These results suggest that repository documentation benefits from modeling repositories as structured design artifacts and intent, not only as collections of local code semantics.

Acknowledgments and Disclosure of Funding

We thank the anonymous reviewers for their constructive comments on improving our work. This work is supported by the NSFC for Young Scientists of China (No.62202400) and the RGC for Early Career Scheme (No.27210024). Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect the views of NSFC and RGC.

Apart from the funding and institutional affiliations disclosed above, the authors declare no additional relevant financial relationships or third-party support during the 36 months prior to submission.

References

[1] Jazlyn Hellman, Eunbee Jang, Christoph Treude, Chenzhun Huang, and Jin L. C. Guo. Generating github repository descriptions: A comparison of manual and automated approaches. *CoRR*, abs/2110.13283, 2021.

[2] Xuejun Zhang, Xia Hou, Xiuming Qiao, and Wenfeng Song. A review of automatic source code summarization. *Empir. Softw. Eng.*, 29(6):162, 2024.

[3] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 1882–1894. IEEE, 2025.

[4] Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. Tool documentation enables zero-shot tool-usage with large language models. *CoRR*, abs/2308.00675, 2023.

[5] Jingyi Chen, Songqiang Chen, Jialun Cao, Jiasi Shen, and Shing-Chi Cheung. When llms meet API documentation: Can retrieval augmentation aid code generation just as it helps developers? *CoRR*, abs/2503.15231, 2025.

[6] Worawalan Chatlatanagulchai, Kundjanasith Thonglek, Brittany Reid, Yutaro Kashiwa, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, and Hajimu Iida. On the use of agentic coding manifests: An empirical study of claude code. In Giuseppe Scanniello, Valentina Lenarduzzi, Simone Romano, Sira Vegas, and Rita Francese, editors, *Product-Focused Software*

Process Improvement - 26th International Conference, PROFES 2025, Salerno, Italy, December 1-3, 2025, Proceedings, volume 16361 of *Lecture Notes in Computer Science*, pages 543–551. Springer, 2025.

[7] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Trans. Softw. Eng. Methodol.*, 34(7):188:1–188:28, 2025.

[8] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sung Hun Kim. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, 35(2):58:1–58:72, 2026.

[9] Vali Tawosi, Salwa Alamir, Xiaomo Liu, and Manuela Veloso. LLM agents for automated dependency upgrades. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025 - Workshops, Seoul, Republic of Korea, November 16-20, 2025*, pages 34–37. IEEE, 2025.

[10] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. Migrating code at scale with llms at google. In Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang, editors, *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, pages 162–173. ACM, 2025.

[11] Zezhou Yang, Ting Peng, Cuiyun Gao, Chaozheng Wang, Hailiang Huang, and Yuetang Deng. A deep dive into retrieval-augmented generation for code completion: Experience on wechat. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 608–619. IEEE, 2025.

[12] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE):675–698, 2024.

[13] Fannar Steinn Aðalsteinsson, Björn Borgar Magnússon, Mislav Milicevic, Adam Nirving Davidsson, and Chih-Hong Cheng. Rethinking code review workflows with LLM assistance: An empirical study. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2025, Honolulu, HI, USA, October 2-3, 2025*, pages 488–497. IEEE, 2025.

[14] Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. A survey on modern code review: Progresses, challenges and opportunities. *CoRR*, abs/2405.18216, 2024.

[15] Anh Nguyen Hoang, Minh Le-Anh, Bach Le, and Nghi D. Q. Bui. Codewiki: Evaluating ai's ability to generate holistic documentation for large-scale codebases. *CoRR*, abs/2510.24428, 2025.

[16] 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. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *CoRR*, abs/2402.16667, 2024.

[17] Katherine R. Dearstyne, Alberto D. Rodriguez, and Jane Cleland-Huang. Supporting software maintenance with dynamically generated document hierarchies. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*, pages 426–437. IEEE, 2024.

[18] Yifeng Zhu, Xianlin Zhao, Xutian Li, Yanzhen Zou, Haizhuo Yuan, Yue Wang, and Bing Xie. Reposummary: Feature-oriented summarization and documentation generation for code repositories. *CoRR*, abs/2510.11039, 2025.

[19] Amirkia Rafiei Oskooei, Selcan Yukcu, Mehmet Cevheri Bozoglan, and Mehmet S. Aktas. Repository-level code understanding by llms via hierarchical summarization: Improving code search and bug localization. In Osvaldo Gervasi, Beniamino Murgante, Chiara Garau, Yeliz Karaca, Maria Noelia Faginas Lago, Francesco Scorza, and Ana Cristina Braga, editors, *Computational Science and Its Applications - ICCSA 2025 Workshops - Istanbul, Turkey, June 30 - July 3, 2025, Proceedings, Part I*, volume 15886 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2025.

[20] Nilesh Dhulshette, Sapan Shah, and Vinay Kulkarni. Hierarchical repository-level code summarization for business applications using local llms. In *IEEE/ACM International Workshop on Large Language Models for Code, LLM4Code@ICSE 2025, Ottawa, ON, Canada, May 3, 2025*, pages 145–152. IEEE, 2025.

[21] Weisong Sun, Yiran Zhang, Jie Zhu, Zhihui Wang, Chunrong Fang, Yonglong Zhang, Yebo Feng, Jiangping Huang, Xingya Wang, Zhi Jin, and Yang Liu. Commenting higher-level code unit: Full code, reduced code, or hierarchical code summarization. *CoRR*, abs/2503.10737, 2025.

[22] Bingyang Wei. Requirements are all you need: From requirements to code with llms. In Grischa Liebel, Irit Hadar, and Paola Spoletini, editors, *32nd IEEE International Requirements Engineering Conference, RE 2024, Reykjavik, Iceland, June 24-28, 2024*, pages 416–422. IEEE, 2024.

[23] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Patterns of folder use and project popularity: a case study of github repositories. In Maurizio Morisio, Tore Dybå, and Marco Torchiano, editors, *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, pages 30:1–30:4. ACM, 2014.

[24] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In Foutse Khomh, Chanchal K. Roy, and Janet Siegmund, editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 200–210. ACM, 2018.

[25] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[26] Qiuyuan Chen, Zezhou Yang, Zhongxin Liu, Shanping Li, and Cuiyun Gao. Parameter description generation with the code parameter flow. In *22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022*, pages 884–895. IEEE, 2022.

[27] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael R. Lyu. Code structure-guided transformer for source code summarization. *ACM Trans. Softw. Eng. Methodol.*, 32(1):23:1–23:32, 2023.

[28] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. Automatic code summarization via chatgpt: How far are we? *CoRR*, abs/2305.12865, 2023.

[29] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. What makes good in-context demonstrations for code intelligence tasks with llms? In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 761–773. IEEE, 2023.

[30] Shangbo Yun, Shuhuai Lin, Xiaodong Gu, and Beijun Shen. Project-specific code summarization with in-context learning. *J. Syst. Softw.*, 216:112149, 2024.

[31] Yiming Wang, Zhuosheng Zhang, and Rui Wang. Element-aware summarization with large language models: Expert-aligned evaluation and chain-of-thought method. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 8640–8665. Association for Computational Linguistics, 2023.

[32] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[33] Zhuang Luo, Yichuan Li, Zexing Xu, Kyumin Lee, and S. Rasoul Etesami. Faithfulpersona: Balancing faithfulness and personalization in code explanations through self-critique. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, volume NAACL 2025 of *Findings of ACL*, pages 930–944. Association for Computational Linguistics, 2025.

[34] Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey I. Nikolenko. Proconsul: Project context for code summarization with llms. In Franck Dernoncourt, Daniel Preotiuc-Pietro, and Anastasia Shimorina, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: EMNLP 2024 - Industry Track, Miami, Florida, USA, November 12-16, 2024*, pages 866–880. Association for Computational Linguistics, 2024.

[35] Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. DocAgent: A multi-agent system for automated code documentation generation. In Pushkar Mishra, Smaranda Muresan, and Tao Yu, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, July 2025.

[36] Hongkui He, Jiexin Wang, Liuwen Cao, and Yi Cai. Clcosum: Curriculum learning-based code summarization for code language models. In *33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*, pages 1–13. IEEE, 2025.

[37] Dongming Jin, Zhi Jin, Yiran Zhang, Zheng Fang, LinYu Li, Yuanpeng He, Xiaohong Chen, and Weisong Sun. Usertrace: User-level requirements generation and traceability recovery from software project repositories. *CoRR*, abs/2509.11238, 2025.

[38] DeepWiki. Deepwiki, 2025. URL `https://deepwiki.com/`.

[39] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. On the effectiveness of llm-as-a-judge for code generation and summarization. *IEEE Trans. Software Eng.*, 51(8):2329–2345, 2025.

[40] Yang Wu, Yao Wan, Zhaoyang Chu, Wenting Zhao, Ye Liu, Hongyu Zhang, Xuanhua Shi, Hai Jin, and Philip S. Yu. Can large language models serve as evaluators for code summarization? *IEEE Trans. Software Eng.*, 51(12):3205–3217, 2025.

[41] AIDotNet. Opendeepwiki, 2025. URL `https://github.com/AIDotNet/OpenDeepWiki`.

A Method Appendix

A.1 Implementation Details of Logical Entry Grouping

A.1.1 AST Parsing and Graph Construction.

We utilize `Tree-Sitter`[2] parsers to extract Abstract Syntax Trees (ASTs) across supported languages (Python, JavaScript, TypeScript, C#, Java, C, C++). The Component Call Graph $G = (V, E)$ is constructed by resolving symbol references within the ASTs. Nodes V correspond to function and class definitions, while edges E represent function calls, class instantiations, and inheritance relationships. To handle dynamic languages (e.g., Python, JavaScript), we employ basic type inference based on import paths to resolve ambiguous calls.

A.1.2 Anchor Filtering.

We instantiate the entry-anchor criterion with a strict source-node threshold. Given the call graph $G = (V, E)$, where an edge $u \to v$ means that component u depends on or calls component v, the raw anchor set is:

$$\widetilde{\mathcal{A}} = \{v \in V \mid \deg^-(v) = 0\}. \tag{5}$$

This corresponds to the general sparse-incoming criterion with threshold $\tau_{\text{in}} = 0$ in our implementation. The raw set is deliberately high-recall: it may include test modules, benchmark drivers, parser artifacts, and fully isolated components.

Invalid Entry Filtering. We first remove raw anchors that cannot define a meaningful downstream region. Since $\widetilde{\mathcal{A}}$ is instantiated directly over call-graph components, each candidate already has a graph identity. Candidates with $\deg^+(a) = 0$ have no implementation dependency to traverse, so they are treated as orphan components for diagnostics and downstream co-location, but are excluded from logical entry grouping. For each remaining candidate, we verify that its outgoing traversal reaches at least one valid component after removing unresolved, external, generated, or otherwise invalid graph nodes. Candidates whose downstream region is empty after this validation are discarded.

Test Entry Filtering. We next filter test-related entries with a multi-signal classifier. Candidate discovery marks obvious test-like entries using path and symbol-name conventions, such as test/spec directories, test/spec file suffixes, and test-like function or class names. During graph-feature construction, this signal is refined with a rule-based likelihood score that also considers source type, test-framework tokens, existing test evidence, test-to-source associations, and whether the candidate mainly depends on test or production components. High- and medium-confidence test-like entries are removed from logical entry grouping, while low-confidence cases are conservatively retained.

LLM-Assisted Filtering. For ambiguous or high-impact cases, we optionally apply an LLM-based entry review. This review is only run on a selected subset of candidates whose rule score is ambiguous or whose test signals could affect grouping quality. The model classifies whether the candidate is test-like, production-like, standalone test code, or uncertain. Medium- or high-confidence test-like decisions exclude the candidate from logical entry grouping; explicit production-like decisions keep it; uncertain cases fall back to the rule-based classifier. Test-like entries may still be retained as test evidence for later documentation or attachment steps, but they are not used as logical entries.

A.1.3 Anchor Consolidation.

For each retained anchor $a \in \mathcal{A}$, reachability is computed from anchor a itself in the call graph. Thus the anchor receives depth 0, its direct callees receive depth 1, and so on. This convention preserves the anchor as part of the downstream profile while capturing the implementation region reached from the entry. When a logical reachability matrix is already available from graph processing, we reuse it; otherwise, we recompute the same profile with breadth-first traversal. The reachability profile can be computed by Equation (1).

To compare downstream regions, a reachable component receives a depth- and sharing-aware weight:

$$w(a, v) = \frac{1}{1 + d(a, v)} \cdot \frac{1}{|\{a' \in \mathcal{A} : v \in \mathcal{R}(a')\}|}. \tag{6}$$

[2] `https://tree-sitter.github.io/tree-sitter/`

The first factor emphasizes nearby implementation components, while the second discounts shared utilities and infrastructure reached by many anchors.

We use weighted reachability containment for entry anchor consolidation. Given two entry anchors a_i and a_j, the containment of a_i in a_j is:

$$C(a_i \to a_j) = \frac{\sum_{v \in \mathcal{R}(a_i) \cap \mathcal{R}(a_j)} w(a_i, v)}{\sum_{v \in \mathcal{R}(a_i)} w(a_i, v)}. \tag{7}$$

If this score is sufficiently high, a_i is absorbed into a_j; otherwise, the two anchors are kept separate. The threshold is relation-aware and adaptive: anchors from closer source locations can be merged with weaker evidence, while distant anchors require stronger containment. The implementation further raises the threshold for semantically dissimilar public-facing anchors to avoid merging unrelated entries that only share common infrastructure. As shown in Equation (2), after merging, each logical entry group aggregates the profiles of its member anchors by keeping the shortest depth to each reachable component. The implementation validates that consolidation does not create duplicate strong identities across groups and preserves the component coverage of the retained anchors.

A.2 Implementation Details of Path-Guided Bucket Initialization

Shared Infrastructure Filtering. Before building path-guided buckets, we separate broadly shared infrastructure components from entry-specific implementation footprints. Let V denote the set of implementation components. For each non-entry component v, let $\rho(v)$ denote the number of logical entry groups that reach v, and let $\deg^-(v)$ denote its graph in-degree. We define the candidate set:

$$V_{\text{cand}} = \{v \in V \mid v \text{ is not an entry component and } \rho(v) > 0\}. \tag{8}$$

Given a repository-level sharing percentile threshold q_{share}, let

$$r_{\text{share}} = Q_{q_{\text{share}}}(\{\rho(x) : x \in V_{\text{cand}}\}), \qquad d_{\text{share}} = Q_{q_{\text{share}}}(\{\deg^-(x) : x \in V_{\text{cand}}\}). \tag{9}$$

We define the global-common set as

$$V_{\text{shared}} = \{v \in V_{\text{cand}} \mid \rho(v) \geq r_{\text{share}} \wedge \deg^-(v) \geq d_{\text{share}}\}. \tag{10}$$

Components in V_{shared} are treated as shared context rather than evidence that unrelated entry groups should belong to the same bucket. They are therefore excluded from ordinary initialization footprints and handled separately in the final bucket output.

Entry Footprints. For each logical entry group $s \in \mathcal{S}$, let $\mathcal{R}(s)$ be the set of implementation components reachable from s in the logical call graph. Let $V_{\text{anchor}}(s)$ denote the protected anchor/member components that define the consolidated entry group. After removing globally shared infrastructure components V_{shared}, the initializer uses the effective entry footprint:

$$\Phi(s) = (\mathcal{R}(s) \setminus V_{\text{shared}}) \cup V_{\text{anchor}}(s). \tag{11}$$

This footprint provides the logical evidence used to initialize path-rooted buckets.

To avoid over-counting reusable components, each component is weighted by inverse sharing frequency:

$$\omega(v) = \frac{1}{\max(1, \rho(v))}. \tag{12}$$

For any component set A, its sharing-aware complexity is

$$\Omega(A) = \sum_{v \in A} \omega(v). \tag{13}$$

Thus, components reached by many entry groups contribute less than components that are specific to a small number of entry groups.

Path-Tree Aggregation. We build a repository path tree over directory prefixes of entry source files. Each entry group s is inserted into the directory node containing its anchor source file. For a path-tree node u, let $\mathcal{S}(u)$ be the entry groups inserted in the subtree rooted at u. The node footprint can be computed by Equation (3), and the node complexity used for initialization is $\Omega(M(u))$.

Adaptive Complexity Budget. The split and merge budget is derived from the empirical distribution of per-entry footprint complexities. Let

$$a_s = \Omega(\Phi(s)), \qquad s \in \mathcal{S}. \tag{14}$$

We compute the median footprint scale and a dispersion factor:

$$\mu_{\text{med}} = \text{median}_{s\in\mathcal{S}} a_s, \qquad \eta = 1 + \text{clip}(\text{CV}(\{a_s\}_{s\in\mathcal{S}})), \tag{15}$$

where CV is the coefficient of variation and clipping prevents degenerate ranges. The adaptive lower and upper bounds are:

$$\Omega_{\min} = \max(5, \mu_{\text{med}}/\eta), \qquad \Omega_{\max} = \max(\Omega_{\min} + 5, \mu_{\text{med}}\eta). \tag{16}$$

This gives repositories with more heterogeneous entry footprints wider admissible bucket ranges while keeping the budget data-driven.

Recursive Initialization. Starting from the path-tree root, the initializer recursively partitions path nodes according to $\Omega(M(u))$. A node becomes an initial bucket candidate when its complexity lies within the adaptive range $[\Omega_{\min}, \Omega_{\max}]$. The candidate carries the entry groups $\mathcal{S}(u)$ together with the accumulated footprint $M(u)$. If its complexity exceeds $\Omega_{\max}$ and the node has child path regions, the initializer descends into those children. If no further path split is available, the node is retained as an oversized initial candidate and is penalized by J_{over} during subsequent refinement rather than being split artificially. If its complexity falls below $\Omega_{\min}$, it is deferred for local merging with nearby candidates under the same parent region, provided that the merged footprint remains within the budget. Components in V_{shared} and orphan components excluded from entry grouping are assigned to auxiliary buckets and are not used as merge evidence during ordinary initialization.

This procedure returns an initial bucket set $\mathcal{B}^0$ whose regions are rooted in repository paths but whose granularity is determined by logical reachability. Subsequent cross-structure refinement may adjust both bucket boundaries and component memberships.

A.3 Details of Physical-Logical Affinity Refinement

Bucket-Level Logical Affinity. For a bucket B_i, let V_i denote its represented components. We summarize B_i by two logical signatures derived from canonical entry-group reachability: an entry-group set $\mathcal{S}_i$ and an average depth profile $\bar{d}_i$ over those groups. The logical affinity is:

$$\ell(B_i, B_j) = \frac{1}{2}\text{Jac}(\mathcal{S}_i, \mathcal{S}_j) + \frac{1}{2}D_{\text{depth}}(\bar{d}_i, \bar{d}_j), \tag{17}$$

where Jac is the Jaccard overlap between the two entry-group sets, and D_{depth} is a normalized consistency score over their shared entry groups, assigning higher values when the shared groups reach both buckets at similar average depths. Direct cross-bucket dependency support is not included in the definition of $\ell(B_i, B_j)$; it is used only as a secondary merge heuristic when two candidate edits have comparable affinity and objective values. Auxiliary buckets are excluded from ordinary affinity refinement and appended unchanged to the final output.

Bucket-Level Physical Affinity. The physical view measures whether two buckets occupy nearby regions of the repository path tree. For each bucket, we aggregate path features from its components, including directory distance, longest common prefix ratio, same-directory evidence, same-file evidence, and path-depth gap. Let δ_{ij} denote the effective path distance between B_i and B_j. We convert this distance into a path-tree kernel:

$$p(B_i, B_j) = \exp(-\delta_{ij}/\sigma_{ij}), \tag{18}$$

where σ_{ij} is derived from the average path depth of the two buckets. Here δ_{ij} summarizes both the tree distance between bucket path regions and the dispersion of component-level path features within the two buckets. The score is high when the buckets are physically close and low when their components are dispersed across unrelated paths.

Local Confidence and Arbitration. The two views are calibrated before arbitration:

$$\hat{\ell}_{ij} = \text{sigmoid}\left(\frac{\ell(B_i, B_j) - 0.5}{T_\ell}\right), \qquad \hat{p}_{ij} = \text{sigmoid}\left(\frac{p(B_i, B_j) - 0.5}{T_p}\right), \tag{19}$$

where T_ℓ and T_p are view-specific temperatures estimated from signal dispersion. Each view receives a local reliability-weighted confidence:

$$q_{ij}^{\ell} = \rho_{ij}^{\ell} |2\hat{\ell}_{ij} - 1|, \qquad q_{ij}^{p} = \rho_{ij}^{p} |2\hat{p}_{ij} - 1|. \tag{20}$$

Here ρ_{ij}^{ℓ} depends on reachability coverage, shared-entry consistency, and depth consistency, while ρ_{ij}^{p} depends on path-feature density, directory evidence, and physical coverage. The local arbitration weights are:

$$\tau_{ij}^{\ell} = \frac{q_{ij}^{\ell} + \epsilon}{q_{ij}^{\ell} + q_{ij}^{p} + 2\epsilon}, \qquad \tau_{ij}^{p} = \frac{q_{ij}^{p} + \epsilon}{q_{ij}^{\ell} + q_{ij}^{p} + 2\epsilon}. \tag{21}$$

In practice, the weights are clipped to avoid degenerate decisions when one view is sparse. The calibrated scores are used only to estimate view confidence and arbitration weights; the final affinity combines the original normalized logical and physical affinities.

Bucket-Level Refinement. The bucket-level affinity is:

$$A(B_i, B_j) = \tau_{ij}^{\ell} \ell(B_i, B_j) + \tau_{ij}^{p} p(B_i, B_j). \tag{22}$$

For a candidate partition $\mathcal{B}$, we evaluate a normalized objective defined as:

$$\mathcal{J}(\mathcal{B}) = \lambda_{\text{dep}} J_{\text{dep}} + \lambda_{\text{path}} J_{\text{path}} + \lambda_{\text{over}} J_{\text{over}} + \lambda_{\text{bal}} J_{\text{bal}} + \lambda_{\text{agree}} J_{\text{agree}} + \lambda_{\text{coup}} J_{\text{coup}}, \tag{23}$$

where each term lies in $[0, 1]$ and lower is better. J_{dep} is the mean fraction of dependency edges in each bucket that point to components outside the bucket. J_{path} is the mean path dispersion, computed as one minus the path similarity between bucket entries and the bucket path region. J_{over} penalizes buckets whose sharing-aware complexity exceeds $\Omega_{\max}$, and J_{bal} penalizes high variance in bucket sizes. J_{agree} penalizes local disagreement between calibrated logical and physical supports, especially when both views are reliable but conflict. J_{coup} penalizes separating directory regions with known coupling evidence. Candidate edits are accepted only when they do not increase $\mathcal{J}$.

Component-Level Reassignment. After bucket-level refinement, the method examines boundary components whose current bucket fit is weak. For a component v currently in source bucket B_i, and a candidate destination bucket B_j, we define component-to-bucket fit scores. The logical fit $\ell(v, B_j)$ measures the overlap between the canonical entry-group reachability profile of v and the entry-group profile represented by components in B_j, with a depth discount. The physical fit $p(v, B_j)$ measures the proximity between v's file path and the path region of B_j.

Using the source–destination arbitration weights, we define a direction-specific component affinity:

$$A_{i \to j}(v, B_k) = \tau_{ij}^{\ell} \ell(v, B_k) + \tau_{ij}^{p} p(v, B_k), \quad k \in \{i, j\}. \tag{24}$$

The same arbitration weights are used to evaluate both the current bucket B_i and the candidate destination bucket B_j for this migration. A migration is proposed only when the best destination improves this joint affinity over the current bucket by a sufficient margin:

$$\max_{j \neq i} \left[A_{i \to j}(v, B_j) - A_{i \to j}(v, B_i) \right] \geq \gamma. \tag{25}$$

The migration is applied only if the updated partition does not increase $\mathcal{J}(\mathcal{B})$. Components that do not satisfy the boundary or low-fit condition are left unchanged, keeping the refinement focused on ambiguous assignments rather than rewriting stable bucket interiors.

A.4 Repository artifacts and tests.

PLP first partitions implementation components and then enriches the resulting buckets with repository artifacts and tests. This post attachment does not change the component partition.

Artifact attachment. Let $\mathcal{F}$ denote non-code repository artifacts, such as configuration files, documentation files, schemas, build files, deployment metadata, and assets. PLP defines an artifact-to-bucket support relation as:

$$\Gamma_F \subseteq \mathcal{F} \times \mathcal{B}, \tag{26}$$

where $(f, B_k) \in \Gamma_F$ means that artifact f provides contextual evidence for interpreting bucket B_k. This relation is many-to-many at the evidence level: a bucket may receive multiple artifacts, and an

artifact may support multiple buckets when it constrains a shared interface, build target, schema, or repository-wide behavior.

PLP estimates artifact support from path locality, artifact type, explicit file or directory references, and lightweight naming cues. Artifacts located inside a bucket's path region are attached to that bucket when the locality signal is unambiguous. Remaining artifacts are matched using category and path support. Artifacts whose scope is repository-wide or whose owner is ambiguous are not forced into a single implementation bucket; instead, they are retained as auxiliary repository-asset evidence and may be exposed to related buckets during intent construction. Content-level summaries of these artifacts are produced later for SAIL, but they do not change the component partition.

Test attachment. Let $\mathcal{T}$ denote test-related repository entries. Tests are handled separately because they may either specify product behavior or implement testing infrastructure. PLP first identifies test-like entries using path, naming, dependency, and test-framework signals, with model-based review for ambiguous or high-impact cases. Product-behavior tests are then represented by a test-to-bucket support relation: $\Gamma_T \subseteq \mathcal{T} \times \mathcal{B}$, where $(t, B_k) \in \Gamma_T$ indicates that test t exercises or specifies behavior owned by bucket B_k. This relation is many-to-many at the evidence level: a bucket may receive multiple tests, and a test may support multiple buckets when it exercises cross-bucket or integration behavior. In the materialized PLP output, strongly local tests are stored under their best-supported bucket, while broad or ambiguous tests are preserved as cross-bucket or auxiliary test evidence.

Test support is estimated from explicit target components or target source files when available, and otherwise from dependency overlap, path locality, naming similarity, and test-coupling signals. Tests with clear single-bucket support are attached to the corresponding implementation bucket as behavioral evidence. Tests with multi-bucket support are retained as cross-bucket or integration evidence rather than being forced into a single owner. Test-like regions that are better interpreted as reusable frameworks or standalone testing infrastructure are retained as auxiliary test modules.

The enriched PLP output can therefore be viewed as the original implementation bucket together with attached evidence links:

$$\bar{B}_k = (V_k, F_k, T_k), \quad F_k = \{f \in \mathcal{F} \mid (f, B_k) \in \Gamma_F\}, \quad T_k = \{t \in \mathcal{T} \mid (t, B_k) \in \Gamma_T\}. \tag{27}$$

Here V_k is the implementation-component set of the original bucket B_k, while F_k and T_k are evidence sets induced by the attachment relations. SAIL consumes these attached artifacts and tests when selecting context for seed construction and later intent revision.

A.5 Details of Seeded Intent Construction

Algorithm 1 formalizes how SAIL constructs initial bucket intent seeds. The bucket partition is fixed throughout this stage. We use the enriched bucket representation $\bar{B}_k = (V_k, F_k, T_k)$ from Appendix A.4, where V_k contains implementation components and F_k, T_k are artifact and test evidence attached by PLP. Context expansion is used only to decide which related buckets and attached evidence are visible when interpreting a target bucket; it does not merge, split, or reassign buckets. We use Γ_{repo} to denote repository-level contextual summaries and evidence that are not captured by a single bucket-local attachment.

Context construction. The auxiliary graph H captures local relations that may clarify bucket interpretation. Its nodes are the fixed PLP buckets. For each candidate bucket pair (B_i, B_j) supported by PLP metadata, SAIL accumulates relation signals from several sources: cross-bucket dependency edges and their exposed interfaces, physical-logical boundary uncertainty, shared or mediating hub components, and files whose assignment evidence is split across the pair. Artifact and test links are used as additional signals when a configuration file, documentation artifact, or behavioral test provides evidence relevant to both buckets.

These signals are converted into a context strength ψ_{ij}, which measures whether the two buckets should be read together during seed construction. The resulting graph is used only to form bounded reading contexts. Strongly related buckets may be placed in the same context group, while buckets without strong relation signals are processed independently or batched by path locality for efficiency. Thus, H defines local visibility for interpretation rather than a new bucket partition.

Role-revealing material selection. For a target enriched bucket $\bar{B}_k$ in context group G_m, SAIL constructs the candidate material universe $\mathcal{U}_k(G_m)$ from the target bucket's own implementation components, attached artifacts, and attached tests, together with the corresponding evidence from

Algorithm 1 Seeded Intent Construction

Require: Fixed enriched buckets $\bar{\mathcal{B}} = \{\bar{B}_k = (V_k, F_k, T_k)\}_{k=1}^{K}$, PLP relation metadata $\mathcal{M}_{\mathrm{PLP}}$, repository context Γ_{repo}, and seed material budget τ_{seed}

Ensure: Initial bucket intent seeds $\{\tilde{z}_k\}_{k=1}^{K}$

Construct an auxiliary context graph $H = (\bar{\mathcal{B}}, \mathcal{E}_H)$.
for each candidate bucket pair $(\bar{B}_i, \bar{B}_j)$ supported by $\mathcal{M}_{\mathrm{PLP}}$ **do**
    Estimate a context strength ψ_{ij} from cross-bucket dependencies, boundary uncertainty, hub connections, competing file assignments, and shared artifact/test evidence.
    if ψ_{ij} is informative **then**
        Add $(\bar{B}_i, \bar{B}_j, \psi_{ij})$ to $\mathcal{E}_H$.
    end if
end for
Form bounded context groups $\mathcal{G} = \{G_m\}_{m=1}^{M}$ from H, assigning buckets with no strong relation signals to locality-based context batches.
Ensure that every bucket appears as a target bucket in exactly one context group.
for each context group $G_m \in \mathcal{G}$ **do**
    Infer a compact group context summary ξ_m from the relation signals and shared evidence visible in G_m.
    for each target bucket $\bar{B}_k \in G_m$ **do**
        Define the candidate material universe

$$\mathcal{U}_k(G_m) = V_k \cup F_k \cup T_k \cup \bigcup_{\bar{B}_j \in G_m,\, j \neq k} (V_j \cup F_j \cup T_j). \tag{28}$$

        Select bounded role-revealing materials $X_k \subseteq \mathcal{U}_k(G_m)$ under budget τ_{seed}.
        Generate an initial seed $\tilde{z}_k$ from $(\bar{B}_k, X_k, \xi_m, \Gamma_{\mathrm{repo}})$.
    end for
end for
Validate that $\{\tilde{z}_k\}_{k=1}^{K}$ contains exactly one seed for each fixed bucket and that no seed changes its bucket assignment.
return $\{\tilde{z}_k\}_{k=1}^{K}$

related buckets in the same context group. The related buckets provide interpretive context, but the target bucket remains the unit for which the seed is generated.

SAIL then selects a bounded set of role-revealing materials $X_k \subseteq \mathcal{U}_k(G_m)$. Rather than covering files uniformly, the selection prioritizes materials that expose the bucket's architectural role: entry behavior, boundary-facing interfaces, cross-bucket interactions, behavioral contracts encoded by tests, and non-code constraints such as configuration, packaging, or deployment artifacts. For each context group, SAIL also records a compact group context summary ξ_m describing why the buckets are read together; this summary is used as context for seed generation but does not replace the bucket seed.

Seed format and validity. For each fixed enriched bucket $\bar{B}_k = (V_k, F_k, T_k)$, its structured hypothesis is:

$$\tilde{z}_k = \langle r_k, \mathcal{R}_k, \mathcal{I}_k, \mathcal{E}_k, \mathcal{Q}_k, c_k \rangle,$$

where the fields record the provisional role, responsibilities and behaviors, interfaces and boundary interactions, supporting evidence, unresolved questions, and confidence, respectively.

From $(\bar{B}_k, X_k, \xi_m, \Gamma_{\mathrm{repo}})$, SAIL generates a structured initial seed $\tilde{z}_k$. Each seed contains a provisional architectural role, expected responsibilities and salient behaviors, expected interfaces or APIs, supporting evidence references, unresolved questions for later cell-level validation, and a confidence or prior-strength estimate. The seed is therefore an evidence-traceable hypothesis about the bucket before full bottom-up cell observation.

The construction is valid only if every fixed bucket receives exactly one seed and each seed remains attached to its assigned bucket. Context groups may make neighboring buckets and shared evidence visible during interpretation, but they do not authorize merging buckets, splitting buckets, or assigning a seed to a different bucket. This preserves the PLP bucket partition while producing initial intent hypotheses that can later be validated and revised by grounded cell-level evidence.

Algorithm 2 Grounded Intent Revision

Require: Fixed enriched buckets $\bar{\mathcal{B}} = \{\bar{B}_k = (V_k, F_k, T_k)\}_{k=1}^{K}$, initial seeds $\{\tilde{z}_k\}_{k=1}^{K}$, component dependency metadata $\mathcal{M}_{\text{dep}}$, repository context Γ_{repo}, and cell context budget τ_{cell}

Ensure: Revised bucket intents $\{\hat{z}_k\}_{k=1}^{K}$ and finalized cell documents $\{d_j\}_{j=1}^{N}$

Decompose bucket contents and relevant repository evidence into documentable cells $\mathcal{C} = \{C_j\}_{j=1}^{N}$ under dependency-coherence and context-feasibility constraints.
Order cells according to dependency structure so prerequisite or shared cells precede dependent cells.
Initialize the proposal set $\mathcal{P} \leftarrow \emptyset$.
for each cell $C_j \in \mathcal{C}$ in dependency order **do**
    Let $b(j)$ denote the target bucket associated with C_j.
    Generate a cell document d_j from C_j using $\tilde{z}_{b(j)}$ as a provisional semantic scaffold.
    Add any optional evidence-backed revision proposals emitted by d_j to $\mathcal{P}$.
end for
for each bucket $\bar{B}_k \in \bar{\mathcal{B}}$ **do**
    Let $\mathcal{D}_k = \{d_j : b(j) = k\}$.
    Let $\mathcal{P}_k = \{p \in \mathcal{P} : p \text{ targets } B_k \text{ and } p \text{ cites concrete code evidence}\}$.
    Jointly adjudicate $\mathcal{P}_k$ against $\mathcal{D}_k$ and the cited code observations.
    Assign each proposal $p \in \mathcal{P}_k$ a decision $\delta(p) \in \{\text{ACCEPT}, \text{REJECT}, \text{DEFER}\}$.
    Let $\mathcal{P}_k^{\text{acc}} = \{p \in \mathcal{P}_k : \delta(p) = \text{ACCEPT}\}$.
    if $\mathcal{P}_k^{\text{acc}} \neq \emptyset$ **then**
        Merge $\mathcal{P}_k^{\text{acc}}$ with $\tilde{z}_k$ to obtain $\hat{z}_k$.
        Identify the replay cell set $\mathcal{C}_k^{\text{rep}} \subseteq \{C_j : b(j) = k\}$ affected by the accepted revisions.
    else
        Set $\hat{z}_k \leftarrow \tilde{z}_k$ and $\mathcal{C}_k^{\text{rep}} \leftarrow \emptyset$.
    end if
end for
for each bucket $\bar{B}_k$ with $\mathcal{C}_k^{\text{rep}} \neq \emptyset$ **do**
    for each cell $C_j \in \mathcal{C}_k^{\text{rep}}$ **do**
        Regenerate d_j from C_j under the revised intent $\hat{z}_k$.
    end for
end for
return $\{\hat{z}_k\}_{k=1}^{K}$ and $\{d_j\}_{j=1}^{N}$

A.6 Details of Grounded Intent Revision

Algorithm 2 formalizes the grounded revision stage. The stage treats the initial bucket intent seeds $\{\tilde{z}_k\}$ as provisional hypotheses and validates them through bottom-up cell-level documentation. Although cells are constructed after bucket-level seeds are available in the implementation, we present cell construction here because cells are the units used for documentation, evidence collection, revision proposal generation, adjudication, and targeted replay.

Cell decomposition. A cell is the finest unit used for documentation generation and intent observation. Cells are constructed from bucket contents subject to two constraints. First, a cell should be dependency-coherent: entry-centered behavior is grouped with its private supporting components when feasible, whereas reusable or prerequisite components are separated into shared cells so that they can be documented before their consumers. Second, a cell should be context-feasible: its code content, supporting evidence, and relevant intent context must fit within the LLM context budget. High-confidence test modules and selected standalone repository assets may also be represented as cells when they provide behavioral, configuration, or usage evidence. The resulting cells are ordered by dependency structure, allowing later cell documentation to refer to already documented prerequisite behavior rather than rediscovering it.

Cell documentation and proposal emission. During cell-level generation, the seed $\tilde{z}_{b(j)}$ serves as a provisional semantic scaffold rather than a hard constraint. The generator documents the concrete behavior observed in C_j, including functionality, interfaces, dependencies, configuration or asset evidence, and other code-grounded observations. Intent revision is only a secondary output: a cell document may emit a candidate proposal when its observations indicate that the current bucket

interpretation is incomplete, overly broad, or inaccurate. Each proposal records the target bucket, the proposed update, the revision type, a short rationale, a confidence estimate, and a code-level citation. Proposals without concrete code evidence are filtered out before adjudication, so a cell document can flag a candidate change but cannot directly revise a bucket intent.

Adjudication. For each bucket B_k, SAIL aggregates all evidence-backed proposals targeting its seed and evaluates them jointly rather than applying them independently. The adjudicator compares the proposed changes with the generated documents and their cited code observations, then assigns each proposal a decision in $\{\text{ACCEPT}, \text{REJECT}, \text{DEFER}\}$. This joint decision step prevents a single local observation from immediately changing the bucket-level interpretation, while still allowing consistent or high-quality evidence across cells to refine the initial seed. Accepted proposals are merged with $\tilde{z}_k$ to form the revised intent $\hat{z}_k$; if no proposal is accepted, the bucket retains its initial seed.

Targeted replay. Accepted revisions may change the semantic frame under which some cell documents were originally generated. To maintain consistency between revised bucket intents and cell documentation, SAIL performs targeted replay: it selectively reruns documentation generation for the proposing cell and any affected cells identified during adjudication. Cells outside the replay scope are preserved. This yields a finalized set of revised bucket intents and cell documents that are mutually consistent, while avoiding the cost and instability of regenerating all cell documents after every accepted revision.

B Experiment Settings

B.1 Evaluation on CodeWikiBench

Benchmark protocol. CodeWikiBench evaluates repository-level documentation using repository-specific rubric trees. Each rubric decomposes documentation quality into hierarchical requirements, covering both high-level architectural understanding and fine-grained implementation details. Following the official protocol, we use LLM-based judges to assess whether the generated documentation satisfies each leaf-level requirement. This rubric-based protocol is more suitable for repository documentation than surface-form metrics such as BLEU or ROUGE, since correct documentation may use different wording while preserving the same system semantics.

For all methods, we use the same benchmark split, rubric definitions, judge prompts, and model backbone. Unless otherwise stated, we use K2-Instruct[3] as the default backbone model for documentation generation and LLM-based evaluation, following the setup of CodeWiki. To examine whether REPOSCOPE generalizes across different generation backbones, we further conduct additional CodeWikiBench experiments by replacing the documentation generation model with GLM-4.7[4] and MiniMax-M2.1[5]. In these generalization experiments, the benchmark split, rubric definitions, judge prompts, and judge model remain unchanged.

All outputs are converted into the same `docs_tree.json`/`structured_docs.json` schema and exposed through the same document-access interface. Parsed artifacts contain no explicit method/model identifiers. Judges are blind to explicit method identity, but can see hierarchy, prose, and content volume; length and style are not normalized.

Metrics. We report two metrics: **Quality Score** and **Coverage**. Let $\mathcal{L}$ denote the set of leaf-level requirements in a rubric tree. For each leaf requirement $l \in \mathcal{L}$, we obtain m independent LLM-judge assessments, where each assessment gives a binary satisfaction label $s_i(l) \in \{0, 1\}$. The leaf-level score is computed as:

$$\mathcal{S}(l) = \frac{1}{m} \sum_{i=1}^{m} s_i(l). \tag{29}$$

The Quality Score is computed by propagating scores upward through the rubric tree. For an internal node n with children $\mathcal{C}(n)$, its score is:

$$\mathcal{S}(n) = \frac{\sum_{c \in \mathcal{C}(n)} w_c \mathcal{S}(c)}{\sum_{c \in \mathcal{C}(n)} w_c}, \tag{30}$$

[3] `https://huggingface.co/moonshotai/Kimi-K2-Instruct-0905`

[4] `https://huggingface.co/zai-org/GLM-4.7`

[5] `https://huggingface.co/MiniMaxAI/MiniMax-M2.1`

Table 3: Statistics of the anonymized industrial systems used in the real-world evaluation.

System	Repository scale				File composition				
	Dirs	Files	Depth	Size (GB)	Code	Non-code	C/C++ Src.	Headers	Proto
System 1	4,806	24,706	13	1.06	24,581	125	6,825	16,035	1,721
System 2	4,568	24,237	11	1.02	24,106	131	6,645	15,716	1,745
System 3	4,414	23,300	10	0.91	23,174	126	6,661	15,217	1,296

where w_c is the importance weight of child node c. The final Quality Score is the root-node score, reported as a percentage.

Coverage measures the fraction of leaf-level requirements satisfied by the generated documentation. Following the benchmark protocol, a leaf requirement is considered satisfied if $\mathcal{S}(l) \geq \tau$, where $\tau = 0.5$. Coverage is defined as:

$$\text{Coverage} = \frac{|\{l \in \mathcal{L} : \mathcal{S}(l) \geq \tau\}|}{|\mathcal{L}|}. \tag{31}$$

Implementation details. For each method and repository, we evaluate the generated documentation with the same rubric tree and the same set of judge prompts. We keep the decoding and prompting settings consistent with CodeWiki to ensure a fair comparison. The use of multiple judge assessments (three times in our experiments) reduces the variance of individual LLM judgments and follows the official CodeWikiBench evaluation.

B.2 Industrial API Extraction Evaluation

Task setup. We evaluate REPOSCOPE on three anonymized industrial C++ systems provided by our collaborators. Since these systems are confidential and do not have public benchmark annotations, internal experts construct an agent-oriented CLI extraction task for each system. Each system has an expert-written `HARNESS.md` specifying operational requirements: execution environment, CLI format, invocation conventions, constraints, and validation. It does not enumerate target APIs, reveal implementation locations or the components realizing target functionality, or specify successful call compositions. The agent must recover that solution information from source and, when available, documentation. Given the same harness, an agent built on the Claude Code framework inspects the repository and extracts CLI-accessible APIs.

Within each system, the identical harness is reused unchanged across Raw Code, OpenDeepWiki, CodeWiki, and REPOSCOPE. Agent configuration, prompt structure, tools, complete source access, environment, and expert tests are fixed. We run five independent clean-environment, fresh-session trials per system-condition, totaling $3 \times 4 \times 5 = 60$ runs, with no history or intermediate artifacts carried over. We deduplicate the five-run API union and validate every unique API with the same execution tests. This measures cumulative utility within the fixed workflow rather than single-run expected performance or agent-independent documentation quality.

Industrial systems. Table 3 reports the repository-scale statistics of the three systems. All systems are large production codebases with more than 23K files, over 4.4K directories, and substantial C/C++ source, header, and protocol definition files. These systems therefore require the agent to reason over non-trivial repository structure, interface boundaries, and cross-module dependencies.

Beyond repository scale, the three systems also differ in their API usage patterns, as summarized in Table 4. System 1 contains a mixture of simple APIs that can be invoked independently and more complex APIs that require additional logic or auxiliary code. System 2 is dominated by direct single-API invocations, so the main challenge is discovering the right interfaces. System 3 contains fewer expert APIs, but most of them require deeper semantic understanding or multi-step invocation logic, making correctness substantially harder even when relevant APIs are found.

Metrics. We evaluate extracted APIs using **Completeness** and **Correctness**. Completeness measures how many expert-identified functional APIs are recovered by the agent in the required CLI format. Correctness measures how many extracted APIs can be successfully executed under expert-specified test conditions and produce the intended behavior. In the result table, completeness is reported as discovered APIs over expert APIs, while correctness is reported as passing APIs over extracted APIs.

Table 4: API characteristics of the industrial systems.

System	# Expert APIs	API usage pattern
System 1	97	Mixed APIs: some can be invoked directly, while others require additional logic or helper code.
System 2	135	Mostly direct single-API invocations; the main challenge is finding the correct interfaces.
System 3	50	Mostly logic-dependent APIs; successful execution often requires semantic understanding or multi-step API composition.

B.3 LLM and Agent Settings

LLM inference settings. For CodeWikiBench experiments, we use K2-Instruct as the default backbone model. To evaluate the generalization ability of REPOSCOPE, we additionally run documentation generation with GLM-4.7 and MiniMax-M2.1 on CodeWikiBench. Unless otherwise specified, all generation runs use the same decoding configuration: temperature 0.2, top-p 0.95, maximum new tokens 8,192, and repetition penalty 1.0. For LLM-as-Judge evaluation, we use K2-Instruct with temperature 0 to reduce evaluation variance.

Agent configuration for industrial systems. All documentation of the three systems is generated by K2-Instruct. For the industrial API extraction experiments, we use the Claude Code framework with Claude-Opus-4.6 as the underlying agent model. We keep the default Claude Code configuration, including its standard tool-use behavior and execution protocol. All industrial systems are evaluated under the same agent framework and the same `HARNESS.md` specification format.

B.4 Compute Resources

Repository preprocessing, graph construction, documentation orchestration, and evaluation scripts are executed on a Linux server with 32 Intel Xeon Gold 6148 vCPUs, 62 GiB RAM, and 2 TB local storage. These CPU-side stages do not require local GPU computation.

LLM inference is served by collaborator-provided model backends deployed on NVIDIA H20 GPUs with 96GB VRAM per GPU. This backend is used for K2-Instruct generation and LLM-as-Judge evaluation, as well as the additional GLM-4.7 and MiniMax-M2.1 generation experiments. For the industrial API extraction task, the agent uses the Claude Code framework with Claude-Opus-4.6; the local machine handles repository access, tool execution, and harness evaluation, while model inference is provided by the external agent backend.

The dominant compute cost of the experiments comes from LLM inference. We therefore report token consumption and estimated monetary cost in Section C.4, as token usage more directly reflects the computational demand of the documentation generation framework. In contrast, wall-clock time is strongly affected by model-service factors such as concurrency limits, request scheduling, and the number of API calls. Moreover, larger repositories still require more preprocessing and inference because they contain more files, larger call graphs, and deeper documentation hierarchies.

C Evaluation on CodeWikiBench

C.1 Complete Experimental Results on CodeWikiBench

Table 5 reports the complete evaluation results on all 21 repositories in CodeWikiBench. The benchmark covers repositories from diverse software ecosystems, including AI applications, frontend frameworks, developer tools, data systems, firmware, and low-level systems software. It also spans multiple programming languages and a wide range of repository sizes, from 16k to 1.7M lines of code (LOC). This full evaluation complements the main comparison by testing whether the performance trend observed on the core repositories also holds under broader repository distributions.

Overall results. Across all 21 repositories, REPOSCOPE achieves the highest Quality Score among the compared systems on every repository. Averaged over all repositories, REPOSCOPE obtains a Quality Score of 81.41, compared with 67.69 for DeepWiki and 59.57 for CodeWiki. When

Table 5: Full evaluation results on 21 CodeWikiBench repositories categorized by Language. The Core set uses the latest evaluation results.

Repository	Language	LOC	DeepWiki		CodeWiki		REPOSCOPE	
			Score	Cov.	Score	Cov.	Score	Cov.
Core Evaluation Set								
OpenHands	Python	229k	70.41	46/67	59.91	40/67	**88.03**	**58/67**
Svelte	JavaScript	124k	78.83	77/96	68.51	76/96	**88.19**	**80/96**
Puppeteer	TypeScript	136k	57.94	46/82	56.28	46/82	**76.60**	**58/82**
ML-Agents	C#	86k	59.43	28/46	53.51	24/46	**72.25**	**34/46**
Logstash	Java	117k	57.04	32/57	53.42	31/57	**62.98**	**38/57**
Wazuh	C	1.4M	57.09	25/46	51.61	21/46	**67.48**	**35/46**
Electron	C++	184k	44.58	43/92	42.30	48/92	**61.63**	**59/92**
Extended Evaluation Set								
GraphRAG	Python	30k	77.67	45/56	54.22	29/56	**78.01**	**45/56**
Rasa	Python	148k	76.42	52/70	71.96	53/70	**77.62**	**55/70**
Chart.js	JavaScript	91k	91.68	57/61	57.97	38/61	**96.05**	**58/61**
MarkText	JavaScript	62k	77.64	41/57	68.90	31/57	**85.30**	**42/57**
Mermaid	TypeScript	157k	68.91	50/67	56.48	38/67	**91.64**	**62/67**
Storybook	TypeScript	365k	77.85	60/78	74.95	55/78	**91.88**	**71/78**
FluentValidation	C#	16k	57.93	29/48	43.15	26/48	**79.56**	**41/48**
Git-Cred-Manager	C#	41k	77.98	56/70	61.97	43/70	**84.09**	**57/70**
Material-Comp	Java	244k	54.70	60/96	55.60	64/96	**74.90**	**75/96**
Trino	Java	1.5M	49.56	34/68	77.75	52/68	**99.05**	**67/68**
QMK Firmware	C	1.1M	64.82	65/110	45.80	49/110	**89.06**	**96/110**
sumatrapdf	C	1.7M	74.94	31/40	73.03	27/40	**78.24**	**32/40**
JSON	C++	120k	73.19	43/57	63.14	36/57	**81.06**	**46/57**
x64dbg	C++	208k	72.81	72/107	60.58	64/107	**86.00**	**85/107**

compared against the strongest baseline on each repository, REPOSCOPE improves the Quality Score by an average of 12.34 absolute points. These results show that the advantage of REPOSCOPE is not limited to the repositories selected for the main comparison, but is consistently observed across the full CodeWikiBench evaluation, which uses human-written documentation as the reference.

On the Core Evaluation Set, REPOSCOPE achieves an average score of 73.88, outperforming DeepWiki by 13.12 points and CodeWiki by 18.80 points. On the Extended Evaluation Set, REPOSCOPE achieves an average score of 85.18, compared with 71.15 for DeepWiki and 61.82 for CodeWiki. The larger evaluation set therefore preserves the same overall ranking as the core comparison while covering a broader range of languages, repository sizes, and application domains.

Coverage. The score improvements are accompanied by higher coverage of reference points. Over the full benchmark, REPOSCOPE covers $1194/1471$ reference points, corresponding to an aggregate coverage of 81.17%. DeepWiki covers $992/1471$ points (67.44%), and CodeWiki covers $891/1471$ points (60.57%). At the repository level, REPOSCOPE achieves the highest coverage on 20 out of 21 repositories and ties for the highest coverage on the remaining repository, GraphRAG.

The coverage results suggest that the higher Quality Score is not obtained by producing narrower or more selective documentation. Instead, REPOSCOPE generally recovers more benchmark-relevant information while also achieving higher aggregate quality. For example, on Trino, REPOSCOPE covers $67/68$ reference points and obtains a score of 99.05, whereas the strongest baseline covers $52/68$ points and scores 77.75. On QMK Firmware, REPOSCOPE covers $96/110$ points and obtains a score of 89.06, compared with $65/110$ and 64.82 for DeepWiki. On Electron, REPOSCOPE covers $59/92$ points and scores 61.63, while DeepWiki and CodeWiki obtain scores of 44.58 and 42.30.

Performance across languages. CodeWikiBench includes repositories written in Python, JavaScript, TypeScript, Java, C#, C, and C++. REPOSCOPE obtains the best score in every listed language group. For Python repositories, it scores 88.03 on OpenHands, 78.01 on GraphRAG, and 77.62 on Rasa. For JavaScript and TypeScript repositories, it scores 88.19 on Svelte, 76.60 on Puppeteer, 96.05 on Chart.js, 85.30 on MarkText, 91.64 on Mermaid, and 91.88 on Storybook. For statically typed and

systems-oriented repositories, it scores 72.25 on ML-Agents, 79.56 on FluentValidation, 84.09 on Git-Credential-Manager, 74.90 on Material-Comp, 99.05 on Trino, 67.48 on Wazuh, 89.06 on QMK Firmware, 78.24 on sumatrapdf, 81.06 on JSON, and 86.00 on x64dbg.

These results indicate that the observed improvements are not confined to a single programming language or software domain. However, we emphasize that this table provides empirical evidence of cross-language performance rather than isolating the exact source of this behavior. The component-level contribution of the proposed design is analyzed separately in the ablation studies.

Performance across repository scales. The evaluated repositories vary substantially in size. Small and medium-sized projects include FluentValidation (16k LOC), GraphRAG (30k LOC), Git-Credential-Manager (41k LOC), MarkText (62k LOC), and ML-Agents (86k LOC). Large-scale repositories include Wazuh (1.4M LOC), Trino (1.5M LOC), QMK Firmware (1.1M LOC), and sumatrapdf (1.7M LOC). REPOSCOPE remains the best-performing system in both regimes.

For repositories above one million LOC, REPOSCOPE obtains scores of 67.48 on Wazuh, 99.05 on Trino, 89.06 on QMK Firmware, and 78.24 on sumatrapdf. The corresponding strongest baseline scores are 57.09, 77.75, 64.82, and 74.94. For smaller repositories below 50k LOC, REPOSCOPE scores 78.01 on GraphRAG, 79.56 on FluentValidation, and 84.09 on Git-Credential-Manager. Thus, the full benchmark does not show a degradation of REPOSCOPE that is specific to either very large or relatively compact repositories.

Close cases and baseline competitiveness. Although REPOSCOPE achieves the highest score on all repositories, several cases show narrow margins. On GraphRAG, REPOSCOPE scores 78.01, while DeepWiki scores 77.67, and both systems cover $45/56$ reference points. On Rasa, REPOSCOPE scores 77.62, compared with 76.42 for DeepWiki. On sumatrapdf, REPOSCOPE scores 78.24, compared with 74.94 for DeepWiki and 73.03 for CodeWiki. These cases indicate that existing systems can be competitive on some repositories.

At the same time, the performance gap is larger on several repositories with many reference points or larger codebases. For example, REPOSCOPE improves over the strongest baseline by 22.73 points on Mermaid, 21.63 points on FluentValidation, 21.30 points on Trino, 24.24 points on QMK Firmware, and 19.30 points on Material-Comp. These results contribute substantially to the aggregate improvement observed over the full benchmark.

In summary, the complete CodeWikiBench evaluation shows that REPOSCOPE consistently outperforms DeepWiki and CodeWiki across all 21 repositories in terms of Quality Score, while also achieving the highest or tied-highest coverage on every repository. The results hold across different languages, software domains, and repository scales. The following ablation studies further examine which components of REPOSCOPE contribute to these improvements.

C.2 Ablation and Control Analysis

C.2.1 Incremental Ablation of PLP and SAIL

Ablation Variants. We evaluate three incremental variants to isolate the contributions of structural partitioning and intent-guided synthesis: (1) **Naive Bottom-Up**, which follows the physical directory hierarchy and aggregates documentation from files or directories without logical restructuring; (2) **REPOSCOPE w/ PLP**, which applies PLP to construct responsibility-oriented buckets, but generates documentation without the full SAIL procedure; and (3) **REPOSCOPE (Full)**, which further incorporates SAIL to seed, ground, revise, and hierarchically lift architectural intent.

Table 6 shows a consistent monotonic trend across all seven repositories. The average score increases from 36.21 for Naive Bottom-Up to 57.89 with PLP, and further to 73.88 with the full REPOSCOPE pipeline. Coverage exhibits the same pattern, rising from 153/486 covered units to 280/486 after introducing PLP, and to 362/486 after adding SAIL. These results suggest that the two components contribute complementary benefits: PLP improves the construction of documentation contexts, while SAIL further improves the use of those contexts through intent-guided, evidence-grounded synthesis.

Effect of PLP. The improvement from Naive Bottom-Up to REPOSCOPE w/ PLP suggests that physical directory structure alone is often an imperfect proxy for repository-level documentation structure. Directory-based aggregation preserves local file proximity, but it may separate components that participate in the same logical flow or repository responsibility. By reconciling physical organization with

Table 6: Ablation study results on CodeWikiBench. PLP stands for Physical-Logical Partitioning. We compare the full REPOSCOPE system with its variants to evaluate the effectiveness of each component. The best results are highlighted in **bold**.

Repository	Language	LOC	Naive Bottom-Up		REPOSCOPE w/ PLP		REPOSCOPE	
			Score	Cov.	Score	Cov.	Score	Cov.
OpenHands	Python	229k	25.78	19/67	78.02	53/67	**88.03**	**58/67**
Svelte	JavaScript	124k	21.12	15/96	80.77	73/96	**88.19**	**80/96**
Puppeteer	TypeScript	136k	27.27	21/82	33.50	29/82	**76.60**	**58/82**
ML-Agents	C#	86k	42.22	18/46	59.11	26/46	**72.25**	**34/46**
Logstash	Java	117k	39.18	22/57	50.22	27/57	**62.98**	**38/57**
Wazuh	C	1.4M	54.37	21/46	57.45	30/46	**67.48**	**35/46**
Electron	C++	184k	43.53	37/92	46.13	42/92	**61.63**	**59/92**

logical reachability, PLP forms documentation units that better align with functional implementation boundaries. This effect is most visible in OpenHands and Svelte, where scores increase by 52.24 and 59.65 points, respectively, and coverage improves from 19/67 to 53/67 and from 15/96 to 73/96. These gains indicate that PLP helps the system cover substantially more annotated evaluation units, rather than only improving surface-level generation quality.

The PLP-only variant also reveals the limits of structural partitioning. In Puppeteer, Wazuh, and Electron, PLP brings only modest score improvements over Naive Bottom-Up, with gains of 6.23, 3.08, and 2.60 points, respectively. This pattern suggests that coherent structural grouping is helpful but not sufficient in all cases. For repositories where architectural roles are distributed across abstractions, build conventions, or low-level implementation details, a single-pass generation process may still miss or mischaracterize higher-level intent.

Effect of SAIL. Adding SAIL further improves every repository, increasing the average score by 15.99 points over the PLP-only variant and adding 82 covered evaluation units. The largest gain occurs in Puppeteer, where the score rises from 33.50 to 76.60 and coverage doubles from 29/82 to 58/82. This result is consistent with SAIL's design: instead of treating the initial bucket interpretation as final, SAIL constructs intent seeds, grounds them against cell-level evidence, adjudicates evidence-backed revisions, and lifts the revised intent state into higher-level documentation. This process allows implementation evidence to correct or refine the initial architectural interpretation.

The gains on ML-Agents, Logstash, Wazuh, and Electron further suggest that SAIL is useful beyond cases where PLP alone already produces large improvements. Overall, the ablation supports a cautious conclusion: PLP improves the alignment between documentation units and repository structure, while SAIL further improves quality and coverage by grounding architectural intent in code evidence.

C.2.2 Internal Control of SAIL: Fixed Initial Intent

The preceding incremental ablation evaluates SAIL as a whole. To examine its adaptive component, we introduce **Fixed Initial Intent**, which uses the same PLP buckets, seed construction, and hierarchical lifting as full SAIL, but freezes the initial intent and disables evidence-grounded revision and targeted replay. This control preserves the inputs needed by seeding and lifting while isolating the combined contribution of revision and replay; it does not separately estimate their individual effects.

As shown in Table 7, Fixed Initial Intent improves the average score over PLP-only by 1.03 points and adds 17 covered requirements, but lowers the score on OpenHands, ML-Agents, and Logstash. Freezing the initial intent therefore does not reproduce the gains of full SAIL. Full SAIL exceeds Fixed Initial Intent on all seven repositories, raising the average score from 58.92 to 73.88 and coverage from 297/486 to 362/486. On Puppeteer, Fixed Initial Intent scores 12.56 points above PLP-only, while full SAIL scores a further 30.54 points higher. These results support the combined contribution of evidence-grounded revision and targeted replay; they do not distinguish the individual effects of the two components.

Table 7: Internal SAIL ablation on the seven core repositories. PLP-only uses no architectural intent; Fixed Initial Intent retains seeding and lifting but disables revision and replay. Bold metric values denote observed maxima.

Repository	PLP-only		Fixed Initial Intent		Full SAIL	
	Score	Cov.	Score	Cov.	Score	Cov.
OpenHands	78.02	53/67	69.23	49/67	**88.03**	**58/67**
Svelte	80.77	73/96	84.30	76/96	**88.19**	**80/96**
Puppeteer	33.50	29/82	46.06	38/82	**76.60**	**58/82**
ML-Agents	59.11	26/46	49.92	19/46	**72.25**	**34/46**
Logstash	50.22	27/57	45.05	28/57	**62.98**	**38/57**
Wazuh	57.45	30/46	58.65	31/46	**67.48**	**35/46**
Electron	46.13	42/92	59.23	56/92	**61.63**	**59/92**
Average / Total	57.89	280/486	58.92	297/486	**73.88**	**362/486**

Table 8: Buckets receiving accepted intent revisions in full SAIL. The total rate is computed over all buckets, rather than averaging repository-level percentages.

Repository	Total buckets	Revised buckets	Revision rate
OpenHands	86	18	20.93%
Svelte	111	11	9.91%
Puppeteer	34	11	32.35%
ML-Agents	54	13	24.07%
Logstash	29	6	20.69%
Wazuh	58	9	15.52%
Electron	42	3	7.14%
Total	414	71	17.15%

C.2.3 Frequency of Accepted Intent Revisions

To characterize how often SAIL accepts intent revisions, Table 8 reports the fraction of buckets receiving at least one accepted revision. The unit is a bucket, not a revision proposal or replayed cell. Accepted revisions occur in 71/414 buckets (17.15%). Lower revision rates accompany smaller Full-over-Fixed score gains in Electron (7.14%, +2.40) and Svelte (9.91%, +3.89), while ML-Agents (24.07%, +22.33) and Puppeteer (32.35%, +30.54) show larger rates and gains. This descriptive pattern is consistent with selective adaptation of the initial intent. It does not establish a causal relationship between revision frequency and score improvement across repositories, and the absence of an accepted revision does not prove that an initial intent is correct.

C.2.4 Final Documentation Length and Detail Controls

Table 9 reports the length of the final documentation evaluated by CodeWikiBench. We tokenize only the final text retained after CodeWikiBench parsing, using the same K2 tokenizer for all methods and excluding intermediate generation artifacts. The following comparisons examine whether greater final documentation length is sufficient to explain the observed performance differences.

Longer Function-Level Documentation. We run DocAgent [35] with K2-Instruct as the generation backbone and otherwise retain its default configuration. As shown in Table 9, it produces 37.0% more evaluated text than REPOSCOPE, but scores 17.51 points lower and covers 80 fewer requirements. It underperforms REPOSCOPE on every core repository (Table 10). This comparison shows that longer documentation does not necessarily yield higher scores or greater coverage on CodeWikiBench.

Nearly Matched Internal Length. PLP-only produces 976,842 final tokens, only 0.8% fewer than full REPOSCOPE, but scores 15.99 points lower and covers 82 fewer requirements. The nearly matched final lengths suggest that a substantial increase in text volume is not required for the observed SAIL gains. Together with the Fixed Initial Intent control in Appendix C.2.2, these results are consistent with a benefit from adaptive, evidence-grounded synthesis.

Table 9: Final evaluated documentation length and CodeWikiBench performance over seven core repositories. Length ratios are normalized by full REPOSCOPE; averages refer to tokens per repository. Score is averaged across repositories, while coverage is summed.

Method	Final tokens	Average	Length ratio	Score	Coverage
CodeWiki	495,383	70,769	0.50×	55.08	286/486
DeepWiki	865,082	123,583	0.88×	60.76	297/486
REPOSCOPE w/ PLP	976,842	139,549	0.99×	57.89	280/486
REPOSCOPE (Full)	984,715	140,674	1.00×	73.88	362/486
OpenDeepWiki	1,069,946	152,849	1.09×	36.77	163/486
DocAgent	1,348,915	192,702	1.37×	56.37	282/486

Table 10: Comparison of documentation quality and coverage between DocAgent and full REPOSCOPE on the seven core CodeWikiBench repositories.

Repository	DocAgent		REPOSCOPE	
	Score	Coverage	Score	Coverage
OpenHands	72.62	50/67	88.03	58/67
Svelte	49.03	49/96	88.19	80/96
Puppeteer	60.94	51/82	76.60	58/82
ML-Agents	60.34	30/46	72.25	34/46
Logstash	55.56	34/57	62.98	38/57
Wazuh	52.72	25/46	67.48	35/46
Electron	43.37	43/92	61.63	59/92
Average / Total	56.37	282/486	73.88	362/486

Scope of the Controls. REPOSCOPE remains longer than CodeWiki and DeepWiki, and the methods are not forced to identical length or style. These comparisons do not isolate the causal effect of documentation length.

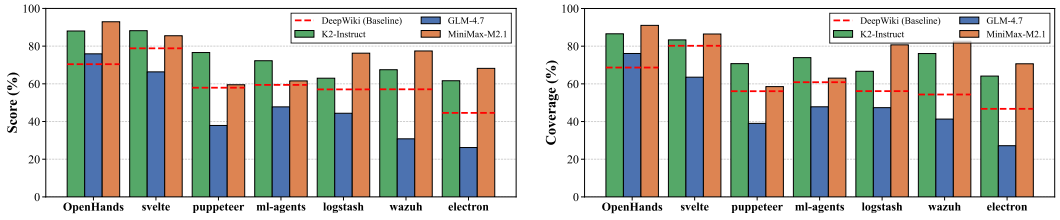
C.3 Model Generalization

C.3.1 Cross-Backbone Evaluation and Sensitivity

To examine whether the effectiveness of REPOSCOPE depends on a specific LLM backbone, we further instantiate the framework with GLM-4.7 and MiniMax-M2.1, and compare them with K2-Instruct and the DeepWiki baseline. As shown in Figure 2, REPOSCOPE generalizes well when paired with sufficiently capable models, but its performance is still bounded by the reasoning and instruction-following capacity of the underlying LLM.

Effect of Backbone Capacity. The results indicate that REPOSCOPE does not remove the dependence on model capability. With stronger backbones such as K2-Instruct and MiniMax-M2.1, the framework generally outperforms DeepWiki in both quality score and coverage across repositories. This suggests that PLP and SAIL provide useful structural and intent-level scaffolding for repository documentation. However, GLM-4.7 does not consistently surpass the baseline, and its performance drops substantially on several repositories. This gap is expected: although REPOSCOPE improves the organization of context and supplies evidence-grounded intent signals, the model still needs to perform non-trivial reasoning, evidence selection, and hierarchical synthesis. When the backbone is less capable, these steps can become the limiting factor.

Repository-Specific Model Behavior. The figure also shows that different strong backbones benefit from REPOSCOPE in different ways. MiniMax-M2.1 performs particularly well on several large or structurally complex repositories, such as Logstash, Wazuh, and Electron, where it achieves higher scores than K2-Instruct. In contrast, K2-Instruct remains stronger on repositories such as Puppeteer and ML-Agents. This suggests that repository characteristics, including programming language, architectural style, abstraction level, and build/runtime conventions, interact with the strengths of each model. Therefore, the best-performing backbone is not uniform across all repositories. MiniMax-M2.1 scores higher than K2-Instruct on Logstash, Wazuh, and Electron under the fixed K2 judge.



(a) The Quality Score (%) of Three LLMs.

(b) The Coverage (%) of Three LLMs.

Figure 2: Experimental results of different LLMs on CodeWikiBench.

Table 11: Pairwise semantic similarity of intent hypotheses, averaged over the seven core repositories. Values are reported in the upper triangle; diagonal and lower-triangular entries are omitted.

	K2-Instruct	K2-Thinking	K2.6	GLM-4.7	GLM-5.2	MiniMax-M2.1	MiniMax-M2.5
K2-Instruct	—	0.797	0.788	0.771	0.782	0.803	0.784
K2-Thinking	—	—	0.782	0.753	0.781	0.775	0.761
K2.6	—	—	—	0.784	0.801	0.795	0.779
GLM-4.7	—	—	—	—	0.781	0.762	0.775
GLM-5.2	—	—	—	—	—	0.792	0.791
MiniMax-M2.1	—	—	—	—	—	—	0.803
MiniMax-M2.5	—	—	—	—	—	—	—

This observation argues against a uniform preference for K2-generated documentation, but does not rule out evaluator bias or establish agreement across independent judges.

Implications. Overall, the generalization results support a cautious conclusion: REPOSCOPE is not tied to a single LLM, and its PLP+SAIL design can improve documentation generation across multiple strong backbones. At the same time, the framework should not be interpreted as fully model-agnostic. Its structured pipeline raises the quality and coverage attainable by capable models, but the final performance ceiling is still determined by the backbone's ability to follow evidence, revise intent, and synthesize multi-level documentation.

C.3.2 Cross-Model Consistency of Intent Hypotheses

The backbone evaluation measures downstream documentation performance. We separately examine whether different models produce semantically similar intent hypotheses for corresponding repository regions. For the same seven core repositories, we generate intent hypotheses using seven configurations from three model families and embed each intent segment with `all-MiniLM-L6-v2`. For each model pair, we compute semantic similarity on the seven corresponding repositories and average the repository-level similarities to obtain the corresponding entry in Table 11.

Across the 21 distinct model pairs, excluding self-comparisons, mean similarity is 0.783 (range 0.753–0.803). Cross-family similarity averages 0.781, close to the within-family mean of 0.790. A negative-control comparison with non-corresponding implementation regions yields an average of 0.488, compared with 0.783 for corresponding regions, giving a margin of 0.295. The higher similarity for corresponding regions is consistent with agreement across models. However, shared errors remain possible, and semantic agreement alone does not validate the inferred architectural roles. The evidence traceability and revisability criteria in Section 3.2, together with downstream tests, provide complementary checks.

C.4 Token Cost Analysis

We further analyze the token consumption and estimated monetary cost of different documentation pipelines. For a fair comparison, all token counts are computed using the K2 tokenizer, since K2-Instruct is the generation backbone used in our main experiments.

Unlike the final documentation lengths reported in Appendix C.2.4, pipeline token counts aggregate all generation calls, including intermediate intent seeds, cell drafts, revision proposals, adjudication,

Table 12: Token consumption and estimated cost of different documentation methods over seven repositories. All token counts use the K2 tokenizer. Cost is estimated with K2 official pricing under the uncached-input assumption. Token totals, ratios, and costs are computed from unrounded counts and rounded independently for display.

Method	Input	Output	Total	Input Ratio	Output Ratio	Est. Cost (USD)	Cost Ratio
OpenDeepWiki	7.60M	3.33M	10.93M	0.96×	0.83×	$11.63	0.87×
Bottom-Up	12.25M	1.82M	14.07M	1.55×	0.45×	$10.86	0.81×
CodeWiki	25.84M	1.52M	27.36M	3.28×	0.38×	$17.73	1.33×
REPOSCOPE w/o SAIL	7.59M	2.92M	10.50M	0.96×	0.72×	$10.69	0.80×
REPOSCOPE (Full)	7.89M	4.04M	11.92M	1.00×	1.00×	$13.35	1.00×

targeted replay, and hierarchical synthesis. Full REPOSCOPE produces 4.04M pipeline output tokens, whereas its final documentation evaluated by CodeWikiBench contains 984,715 tokens.

We estimate cost using the official K2 pricing[6], converted to USD with an exchange rate of 1 USD = 7.2 RMB: RMB 4.00 per 1M uncached input tokens and RMB 16.00 per 1M output tokens. Since our logs do not record cache-hit ratios, we report a conservative estimate that charges all input tokens at the uncached rate:

$$\text{Cost} = \frac{4 \cdot T_{\text{in}}/10^6 + 16 \cdot T_{\text{out}}/10^6}{7.2}.$$

Table 12 reports the aggregated statistics over all seven CodeWikiBench repositories. The input/output ratios are normalized by the full REPOSCOPE pipeline.

Cost Patterns Across Methods. The results show that token usage reflects the operational design of each method. OpenDeepWiki consumes a similar amount of input tokens to REPOSCOPE, but it also produces a relatively large output budget. Because output tokens are four times more expensive than uncached input tokens under the K2 pricing scheme, its monetary cost remains close to the full REPOSCOPE pipeline despite using fewer total tokens.

Bottom-Up has the opposite profile. It consumes more total tokens than REPOSCOPE, mainly because a bottom-up traversal repeatedly reads and aggregates low-level files and directories. However, its output budget is much smaller, so its estimated cost is lower. This indicates that Bottom-Up is not necessarily token-light; rather, much of its consumption lies on the cheaper input side.

CodeWiki has the largest token consumption and the highest estimated cost. This is consistent with its implementation strategy: it uses LLM calls over directory trees and file trees to support partitioning and component extraction. These prompts introduce large structural contexts, which substantially increase input-side token usage. Although CodeWiki produces compact outputs, its large input budget dominates the final cost.

Cost of Intent Guidance. The comparison between REPOSCOPE w/o SAIL and the full pipeline quantifies the additional token consumption associated with SAIL. Input consumption increases from 7.59M to 7.89M tokens, while output consumption increases from 2.92M to 4.04M tokens. The additional consumption is therefore concentrated on the output side, despite only a 0.8% increase in final evaluated documentation length. This distinction indicates that the larger pipeline output count primarily reflects intermediate generation rather than a comparable expansion of the final documentation.

Overall, SAIL adds 1.42M total tokens and about $2.65 over seven repositories under the conservative pricing estimate. The full pipeline remains less expensive than CodeWiki under the same pricing assumptions. Interpreted together with the ablation results, this suggests that REPOSCOPE's gains are not simply due to spending substantially more input tokens, but to allocating a bounded additional budget toward structured context construction and evidence-grounded intent refinement.

D Developer Evaluation in an Industrial Testing Workflow

To complement rubric-based evaluation and executable CLI extraction, we analyze a pre-existing internal developer evaluation conducted for one month on the same three industrial systems. The generating method was concealed in the conditions that supplied documentation.

[6] `https://platform.kimi.com/docs/pricing/chat-k2`

Table 13: Descriptive results of the internal developer evaluation. The 179 executable test cases are aggregated across conditions; condition-specific counts are unavailable. Document use counts assignments in which the supplied documentation was consulted.

Support condition	Developers	Assignments	Median time (min)	Document use
Raw Code	7	19	148	Not applicable
CodeWiki	7	19	126	5/19 (26.3%)
REPOSCOPE	7	19	103	12/19 (63.2%)

D.1 Participants and Task Protocol

The evaluation is conducted internally with company permission. Twenty-one developers are randomly assigned to three support conditions, with seven developers per condition: raw code only, raw code augmented with anonymized CodeWiki documentation, or raw code augmented with anonymized REPOSCOPE documentation. The documentation is anonymized with respect to its generating method. The testing workflow and expert-defined acceptance criteria are fixed across conditions. Assignments are balanced across the three systems and matched by functionality type and estimated difficulty; every completed assignment is required to satisfy the same expert-defined acceptance criteria. Each assignment targeted one newly added or modified backend functionality and typically required multiple test cases for normal, boundary, and failure conditions. Developers complete 57 functionality-level test-construction assignments, with 19 per condition, producing 179 executable test cases in total. Completion time is reported per functionality-level assignment, rather than per individual test case. Document use is measured through access logs.

D.2 Completion Time and Document Use

As shown in Table 13, the REPOSCOPE condition is associated with a 30.4% lower median completion time than Raw Code and an 18.3% lower median than CodeWiki. Documentation is consulted in 5/19 CodeWiki assignments and 12/19 REPOSCOPE assignments. These observations complement rubric scores with descriptive evidence from a developer workflow whose completed outputs satisfy expert-defined acceptance criteria.

E Case Study: Qualitative Analysis

We provide two representative case studies to complement the ablation results in Table 6. These examples are not intended as separate causal evidence, but help illustrate concrete documentation failure modes observed in practice. The OpenHands case highlights how PLP can surface functionally coherent modules that are not prominent in a directory-level view, while the Electron case illustrates how SAIL can make architectural roles more explicit after relevant code contexts have been formed.

E.1 Case Study 1: Module Discovery in OpenHands

Figure 3 shows a structural omission in the documentation of `OpenHands`. The DeepWiki output presents a directory-oriented hierarchy, including folders such as `openhands/`, `frontend/`, `runtime/`, and `controller/`. However, the `resolver` functionality is not surfaced as a distinct module in this high-level view, even though it is relevant to autonomous issue-resolution workflows.

This example reflects a common limitation of relying primarily on physical organization. A directory tree is useful for navigation, but it does not always expose functional responsibility. In `OpenHands`, the files related to issue resolution are organized under the `resolver` path and associated implementation files, but their role can be underemphasized if documentation follows only the visible directory layout.

In contrast, REPOSCOPE presents this functionality as an "Autonomous Issue Resolution" module. This behavior is consistent with PLP, which combines physical locality with logical flow information to construct responsibility-oriented documentation buckets. Rather than only reproducing the directory hierarchy, the generated documentation groups the relevant `resolver` files into a functional unit and describes their role in issue resolution.

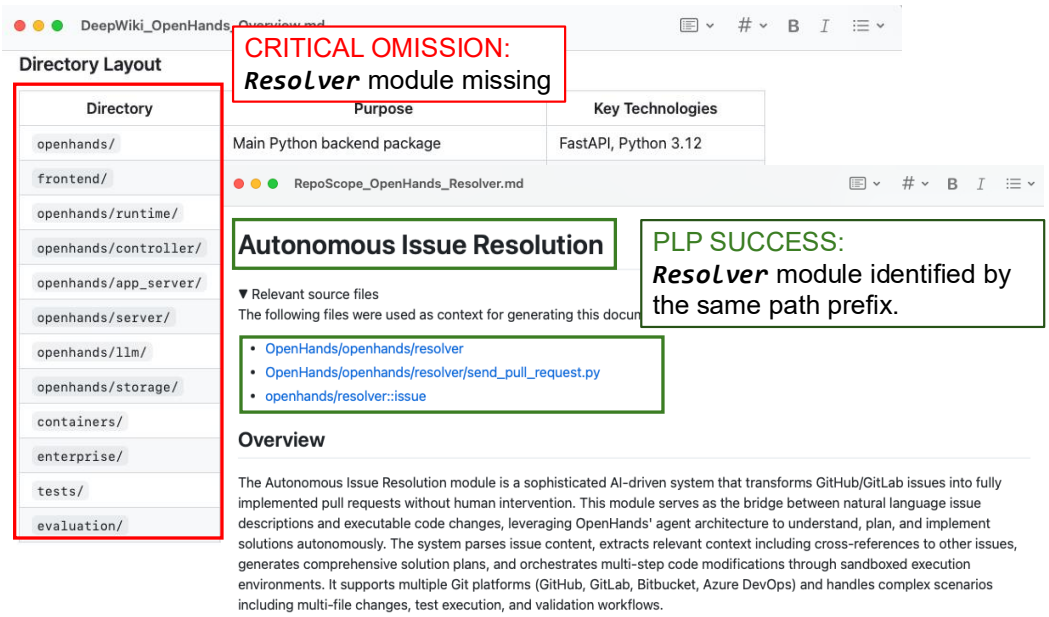


Figure 3: Comparison of module discovery in OpenHands. DeepWiki (left) does not surface the `resolver` module in the displayed high-level directory-oriented view, while REPOSCOPE (right) identifies it as an autonomous issue-resolution module through PLP.

This case is consistent with the ablation results on `OpenHands`: adding PLP improves the score from 25.78 to 78.02 and increases coverage from 19/67 to 53/67. The example illustrates one concrete form such improvement can take: functionally important modules may become visible when documentation units are constructed from both physical and logical signals.

E.2 Case Study 2: Architectural Role Articulation in Electron

Figure 4 illustrates a different failure mode in the `Electron` repository. The DeepWiki output describes several WebContents-related components, but the diagram does not clearly identify the architectural control point that coordinates `<webview>` operations. The output is therefore locally informative, but less explicit about the subsystem-level role of the coordinating module.

REPOSCOPE highlights the `IPC Guest Manager` as the single entry point for `<webview>` operations. This distinction matters because understanding Electron's `<webview>` subsystem requires more than listing the participating components; it also requires identifying which module mediates requests, ownership checks, creation, attachment, and teardown across process boundaries.

This example illustrates the role of SAIL in architectural documentation. After PLP forms a relevant code context, SAIL seeds architectural intent, grounds it against implementation evidence, and lifts the resulting intent into higher-level documentation. In this case, the final output emphasizes the coordinating role of `IPC Guest Manager`, rather than treating it as only another implementation component in the WebContents subsystem.

The ablation results show a compatible pattern. On `Electron`, PLP alone yields only a modest improvement, from 43.53 to 46.13 in score and from 37/92 to 42/92 in coverage. With SAIL, the score further increases to 61.63 and coverage to 59/92. This suggests that, for repositories such as `Electron`, structural grouping alone may be insufficient; grounding and lifting architectural intent can provide additional benefit.

Together, these cases illustrate the complementary roles suggested by the ablation study. The OpenHands example shows how PLP can help surface functional modules beyond a directory-level view. The Electron example shows how SAIL can help make subsystem-level architectural roles explicit once the relevant code context has been constructed.

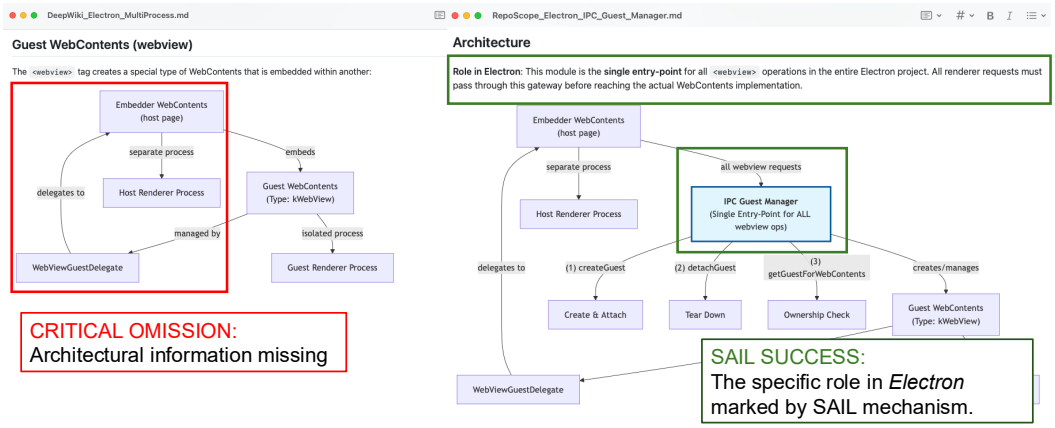


Figure 4: Architectural role articulation in Electron. DeepWiki (left) describes related WebContents components in isolation, while REPOSCOPE (right) articulates the `IPC Guest Manager` as the single entry point for `<webview>` operations through SAIL-guided documentation.

F Limitations and Threats to Validity

Evaluation validity. Repository-level documentation quality is difficult to assess directly, and the CodeWikiBench evaluation relies on rubric-based LLM-as-Judge, which may introduce evaluator variance. We mitigate this threat by using repository-specific rubric trees, setting the evaluator temperature to 0, and repeating each evaluation three times before reporting the average score. We further complement this benchmark with an industrial CLI extraction task, where documentation quality is reflected by deterministic downstream metrics: valid APIs discovered and extracted APIs passing expert-specified execution tests.

Method generality. The effectiveness of REPOSCOPE depends on the availability of useful structural and semantic signals, including call reachability, repository paths, artifacts, tests, and code-level observations. Repositories with highly dynamic dispatch, heavy generated code, incomplete dependency metadata, or weak call-graph coverage may provide noisier evidence. REPOSCOPE mitigates this limitation by combining physical and logical signals with local confidence arbitration, and by using evidence-backed intent revision in SAIL rather than relying on a single structural view. Our results across CodeWikiBench and three industrial systems suggest that this design is robust across diverse repositories, though broader evaluation remains future work.

Downstream agent workflow. Our industrial evaluation uses a fixed Claude Code workflow driven by expert-written `HARNESS.md` files to isolate the effect of different documentation sources. This setting does not optimize the agent pipeline itself, and systems requiring multi-step API composition may benefit from specialized planning, verification, or execution-repair mechanisms. We leave the design of documentation-aware agent workflows to future work.

Reproducibility. The industrial systems used in our real-world evaluation cannot be fully released because they contain confidential production code. To support reproducibility, we report detailed repository statistics and use expert-specified executable tests for deterministic downstream evaluation. The project repository is `https://github.com/watreyoung/RepoScope`.

G Broader Impacts

REPOSCOPE aims to improve repository-level documentation, which can benefit software maintenance, onboarding, code review, and autonomous software-engineering agents by making large codebases easier to understand. Better documentation may also reduce duplicated engineering effort and improve the accessibility of complex software systems.

At the same time, automatically generated documentation may introduce risks if it is inaccurate, incomplete, or used without verification, potentially misleading developers or agents in downstream tasks. The method may also make proprietary or security-sensitive code easier to inspect when

applied without appropriate authorization. We mitigate these risks by grounding documentation in code evidence, evaluating correctness in downstream tasks, and treating confidential industrial systems under collaborator-approved anonymization and non-release constraints.

H Existing Assets and Licenses

We use existing assets only under their applicable licenses or terms of use. CodeWikiBench is used as the public repository-documentation benchmark, and we cite its original source, version and URL. Baseline systems, including CodeWiki and OpenDeepWiki, are credited with their original repositories and licenses; DeepWiki is used according to its service terms where applicable. The LLM backbones and agent frameworks, including Kimi-K2-Instruct, GLM-4.7, MiniMax-M2.1, Claude-Opus-4.6, and Claude Code, are used through their respective providers or collaborator-provided deployments under the corresponding usage terms. The industrial systems are proprietary assets provided by collaborators for research evaluation. They are used with permission, anonymized in the paper, and are not publicly redistributed due to confidentiality constraints.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and Section 1 clearly state the paper's main contributions, including Physical-Logical Partitioning, SAIL, and the evaluation on CodeWikiBench and industrial systems. The claims are scoped to repository-level documentation generation and downstream CLI extraction, matching the method in Section 3 and the experiments in Section 4.1.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: The paper includes a dedicated Limitations and Threats to Validity section in Appendix F, discussing evaluation validity, method generality, reproducibility constraints for industrial systems, and robustness concerns under noisy structural signals. It also reports a Token Cost Analysis in Section C.4, quantifying the computational and monetary overhead of REPOSCOPE and SAIL relative to baselines.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A].

   Justification: The paper does not present theoretical results such as theorems, lemmas, or formal guarantees that require assumptions and proofs. The equations and algorithms in Section 3 and Appendix A are used to specify the proposed framework and implementation procedures rather than to establish theoretical claims.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes].

   Justification: The method, evaluation protocols, model settings, and implementation details are described in Sections 3 and 4.1 and Appendix B. Industrial repository statistics and execution-based evaluation are reported in Appendix B.2. Code access and the limits on reproducing confidential industrial results are discussed in Appendix F.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No].

   Justification: The project repository and industrial-data access restrictions are described in Appendix F. CodeWikiBench uses accessible public repositories, but the confidential industrial production code cannot be openly released. Thus, not all experimental data are openly accessible; industrial system statistics are provided in Appendix B.2.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes].

   Justification: The paper specifies the benchmarks and tasks, evaluation protocols, metrics, baselines, model backbones, and real-world system settings in Section 4.1. Additional metric definitions, implementation details, repository statistics, and experimental settings are provided in Appendix B and Appendix B.2.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No].

Justification: The paper reports descriptive experimental results without formal error bars, confidence intervals, or significance tests. CodeWikiBench evaluations are repeated three times with evaluator temperature set to 0 (Appendix B); industrial CLI extraction reports deduplicated API unions over five trials per condition, validated by expert-specified execution tests. The developer evaluation reports median completion times and documentation-use counts (Appendix D).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes].

   Justification: The paper reports the local CPU-side environment used for preprocessing and orchestration, the collaborator-provided H20 GPU inference backend used for LLM generation and evaluation, and the token consumption and estimated monetary cost that dominate the experimental compute in Appendix B.4 and Appendix C.4.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes].

Justification: The research conforms to the NeurIPS Code of Ethics. The industrial systems are used with collaborator permission, anonymized in the paper, and not publicly released due to confidentiality constraints, as described in Appendix B.2, Appendix F, and Appendix H; public benchmark experiments use accessible repositories and evaluation materials.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes].

Justification: The paper discusses positive impacts such as improving repository comprehension, maintenance, onboarding, and agentic software-engineering workflows in Appendix G. It also discusses potential negative impacts, including risks from inaccurate generated documentation and unauthorized use on proprietary or security-sensitive code, together with mitigation through evidence grounding, evaluation, and confidentiality constraints.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A].

Justification: The paper does not release high-risk models, pre-trained generative models, image generators, or newly scraped datasets that require controlled release safeguards. As described in Appendix H and Appendix F, confidential industrial systems are not publicly released, and only anonymized evaluation materials are provided where permitted.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes].

    Justification: The paper credits the existing benchmarks, baselines, model backbones, and agent framework used in the experiments, and reports their versions, URLs, licenses, or terms of use where applicable in Appendix H. The industrial systems are used with collaborator permission under confidentiality constraints and are not publicly redistributed.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes].

    Justification: The paper releases new assets including the REPOSCOPE implementation, experimental scripts, and anonymized evaluation materials where permitted, accompanied by documentation on setup, usage, evaluation protocols, and limitations, as described in Appendix B and Appendix H. Confidential industrial code is not released, and only collaborator-approved anonymized materials are provided.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [No].

    Justification: The paper includes a small internal developer evaluation involving 21 company developers, as described in Appendix D. The appendix reports the three documentation-support conditions, task allocation, test-construction protocol, completion times, and documentation use. No study-specific compensation was provided for participation.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [No].

    Justification: The developer evaluation in Appendix D was a small internal company study conducted with company permission. Participants performed software test-construction tasks, and only aggregate results are reported. No formal IRB or equivalent ethics approval was obtained.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes].

    Justification: The paper explicitly describes the use of LLMs as part of the proposed documentation generation framework and evaluation pipeline in Section 3, Section 4.1, and Appendix B, including Kimi-K2-Instruct for documentation generation and LLM-as-Judge evaluation, additional generation backbones for robustness experiments, and Claude-Opus-4.6 within Claude Code for the industrial CLI extraction task. Any LLM use for writing or grammar polishing does not affect the core methodology or results.

    Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.