ACCEED: PERSISTENT STRUCTURED CONTEXT FOR CODE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code agents need to identify relevant modules and their interactions before modifying a repository. In large repositories, this can take extensive code exploration, which agents may repeat across tasks unless the knowledge is preserved. Documentation that preserves this knowledge for agents should state which module owns which code and how modules interact, let agents read only the detail a task needs, and stay current as code changes. Whether it meets these requirements depends on how it is organized: a tree gives each file one owner, but module interactions, such as shared dependencies, calls, and data flow, form a graph. We propose Artifact-Centric Code Exploration, Evolution & Drill-down for Hierarchical Knowledge Generation (ACCEED), built on a data structure that separates ownership from interaction, the Agent-oriented Documentation Intermediate Representation (AD-IR). Each AD-IR module owns a source scope, the files it is responsible for, nested in its parent's and disjoint from its siblings', and interactions are kept as typed relations and flows over the resulting tree. ACCEED uses these properties to build the tree recursively with one bounded scope per agent, give agents progressive access through doc-drill, and route code changes through source scopes to the nodes that own them. We also introduce ACCEED-Bench, 240 source-verified questions over six repositories, answered from documentation alone and scored by the weighted share of reference scoring points each answer covers. ACCEED's leaf pages alone outperform documentation from DeepWiki and Google Code Wiki on three shared repositories. AD-IR with doc-drill further improves coverage for every answering model, especially on cross-module and repository-wide questions. After 150 merged pull requests per repository, localized maintenance raises coverage on changed-code questions from 24.7% with frozen documentation to 67.5%, while changing about 1% of documentation lines.

1 INTRODUCTION

Repository-level issue resolution often requires locating relevant code and inspecting behavior across files (Jimenez et al., 2024; Yang et al., 2024). In large repositories, this can require many rounds of code exploration. The resulting understanding is costly to rebuild, but without an explicit mechanism for retaining it, later tasks may repeat much of the same exploration. Once retained, understanding built by a strong model can also be reused by cheaper models on later tasks.

One remedy is to record this understanding in documentation that agents can consult across tasks, including when maintaining one repository requires knowledge of another. Existing generators produce function- and class-level documentation (Luo et al., 2024; Yang et al., 2025) and repository wikis (Hoang et al., 2026; Xu et al., 2026). Their output is organized mainly as pages, and some also use module trees, dependency graphs, or documentation–code links to guide generation and selective updates. **For agents, however, the organization is central: it determines what they can inspect selectively and how code changes map to revisions.** We study how a common set of source-owned units can support construction, selective access, and maintenance while preserving cross-module interactions. Such a representation should meet three requirements:

(1) **Structuring the knowledge.** The representation should identify module responsibilities, the source each module covers, and relationships across modules. A hierarchy of pages alone does not explicitly define source ownership or cross-module relationships.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

(2) **Accessing the knowledge.** Each task typically needs only part of the documentation, and reading entire pages loads unrelated detail into an agent's limited context window. The interface should let agents choose relevant modules and the level of detail to inspect, while retaining access to module relationships.
(3) **Updating the knowledge.** Code changes should identify candidate documentation revisions through source mappings. Maintenance also needs to consider semantic effects beyond directly changed files, such as altered contracts between modules, to determine which descriptions and relationships need revision.

All three requirements rely on *ownership*, which a tree provides by giving each part of the source one responsible module at each level. Ownership splits construction into non-overlapping agent tasks, gives each piece of code one place where agents find it, and tells maintenance which pages a changed file affects. **The challenge is that module interactions are not hierarchical**: a utility can be shared by many modules, and calls, dependencies, and data flow cross module boundaries. Documentation organized along these interactions gives shared code no single owner, while a tree alone loses the interactions that cross-module tasks need. Our key idea is to *separate ownership from interaction*: a tree assigns each part of the source to one module, and interactions become relations over it.

We propose **A**rtifact-**C**entric **C**ode **E**xploration, **E**volution & **D**rill-down for Hierarchical Knowledge Generation (**ACCEED**), built on a data structure that realizes this idea, the **A**gent-oriented **D**ocumentation **I**ntermediate **R**epresentation (**AD-IR**) (Section 3.2). Every node owns a *source scope*, the files it is responsible for, nested inside its parent's and disjoint from its siblings', so a file has at most one owner at each level. A shared utility is assigned to one owner, and its uses are recorded as relations. Internal nodes are modules whose children refine their responsibilities. Leaves are single-responsibility scopes, each written as one page with anchors into the source. Over this tree, typed relations link the children of each module, and interaction flows record scenario-specific interactions between nodes in different branches, at the level of detail each scenario needs.

ACCEED derives three operations from these properties (Section 3.3). Since sibling scopes are disjoint subsets of their parent's, *recursive construction* lets a fresh agent refine each module within its own scope, dividing it into submodules with explicit scopes and relations until single-responsibility scopes are written as leaf pages. Since the tree orders knowledge from overview to detail, *progressive access* uses the *doc-drill* protocol to guide the agent from the root toward relevant branches, following relations and flows across branches, until the evidence suffices. Because a file's owners lie on one ancestor chain, *localized maintenance* maps changed files to candidate leaves through source scopes, and an update agent uses the diff to revise them and, where needed, their ancestors. Insight resumes construction sessions to validate candidate code issues (Sections 3.4 and 5.2).

To measure whether documentation supplies the knowledge a task needs, we build **ACCEED-Bench**, which contains 240 source-verified questions over six repositories. Agents answer them from documentation alone, and each answer is scored by *weighted coverage*, the weighted fraction of reference scoring points it covers. We find three things. First, ACCEED's leaf pages alone record more usable knowledge than external documentation, reaching 71.4% coverage against 40.5% for DeepWiki and 29.5% for Google Code Wiki on the three shared repositories, and full AD-IR with doc-drill exceeds iterative source retrieval at 18.6% lower answer-stage cost. Second, gains are largest on cross-module and repository-wide questions and for weaker answering models, which reuse the understanding GPT-5.6 Sol recorded in AD-IR. Third, maintenance restores answers about new behavior, raising coverage of changed behavior from 24.7% for a frozen tree to 67.5% after 150 merged pull requests per repository. Insight found two issues since confirmed upstream.

Our main contributions are summarized as follows:

- We design AD-IR, a data structure for agent-oriented repository documentation that separates source ownership, encoded as nested, sibling-disjoint source scopes, from module interaction, encoded as typed relations and interaction flows over the resulting tree.
- We develop ACCEED, which uses these properties to construct AD-IR recursively in bounded agent tasks, give agents progressive access through doc-drill, and localize updates through source scopes. Insight reuses its construction sessions to validate code issues.
- We build ACCEED-Bench, 240 source-verified questions over six repositories, and use it for three controlled experiments on structure, access, and maintenance.

2 Related Work

Representing external knowledge for agents. Retrieval-augmented generation (RAG) represents external knowledge as an indexed collection of text passages (Lewis et al., 2020). RAPTOR organizes text chunks and their recursively generated summaries into a tree (Sarthi et al., 2024), while GraphRAG derives hierarchical community summaries from an entity graph (Edge et al., 2024). For repositories, RepoGraph encodes code relationships in a graph and presents retrieved subgraphs as textual context (Ouyang et al., 2025). Codebase-Memory exposes a persistent code graph through structured query tools (Vogel et al., 2026). ACCEED organizes repository knowledge around module responsibilities and explicit source ownership, combining a module hierarchy with typed relations and interaction flows. This makes module boundaries and cross-module interactions part of the representation exposed to agents, alongside documentation at different levels of detail.

Repository documentation generators. RepoAgent generates function- and class-level documentation using a project tree and code-reference metadata, with updates triggered by changes to an object's code or incoming references (Luo et al., 2024). DocAgent generates docstrings in dependency order (Yang et al., 2025). CodeWiki uses dependency analysis and recursive grouping to construct a module tree with bounded delegation depth in its reported configuration (Hoang et al., 2026). Its current implementation also exports the tree and supports selective regeneration after code changes (FSoft-AI4Code, 2026). RepoDoc generates module and component pages and uses documentation–code links in a knowledge graph for selective updates (Xu et al., 2026). DeepWiki publishes wiki pages with diagrams and source links (The Cognition Team, 2025), and its MCP interface exposes documentation topics, content, and question answering (Cognition, n.d.). Google Code Wiki describes automatic documentation refresh, without specifying regeneration granularity in the cited announcement (Hurley et al., 2025). ACCEED makes source scopes and module relations part of the representation that downstream agents inspect and that the updater uses to locate candidate revisions.

Evaluating repository knowledge. CodeWikiBench scores generated documentation against hierarchical rubrics (Hoang et al., 2026). RepoDoc measures API mention coverage, retrieved-context answerability, and update recall (Xu et al., 2026). SWD-Bench evaluates documentation on PR-derived functionality tasks (Wang et al., 2026), while RepoProbe evaluates repository comprehension on checklist-scored questions derived from GitHub Discussions (Yang et al., 2026). SWE-QA provides repository-level code questions with reference answers (Peng et al., 2026). Our experiments measure answer completeness (RQ1, RQ2) and coverage of changed behavior (RQ3).

3 Method

3.1 Problem Setup

Let $R^{(v)}$ denote a snapshot of a repository's source files at a particular Git commit. Given $R^{(v)}$, the objective is to produce a persistent knowledge representation $\mathcal{I}^{(v)}$ grounded in the source and reusable across tasks. Given a task q targeting this snapshot and $\mathcal{I}^{(v)}$, the required output is task-relevant context that supports completing q within the agent's context window. Given $\mathcal{I}^{(v)}$ and a code change $\Delta R^{(v)}$ leading to $R^{(v+1)}$, the objective is to produce revised knowledge $\mathcal{I}^{(v+1)}$ that reflects the new source while preserving information that remains valid.

3.2 AD-IR: Recursive and Source-Grounded Representation

Let V be the set of module nodes in AD-IR at version v, with leaf nodes $L \subseteq V$. Each node n has a module description d_n and a source scope $\Sigma^{(v)}(n)$. A leaf $n \in L$ carries a detailed documentation page P_n with source references. An internal node provides a concise module overview and exposes its immediate submodules for further inspection.

Let $T = (V \cup \{r\}, E_T)$ be the ownership tree with virtual repository root r, where $(n, u) \in E_T$ means that u is a child of n. For each internal node n, including r, let $C(n) = \{u : (n, u) \in E_T\}$ denote its immediate children. Their interactions form a local graph $g_n = (C(n), E_n)$, with

$$E_n \subseteq C(n) \times \mathcal{T} \times C(n),$$

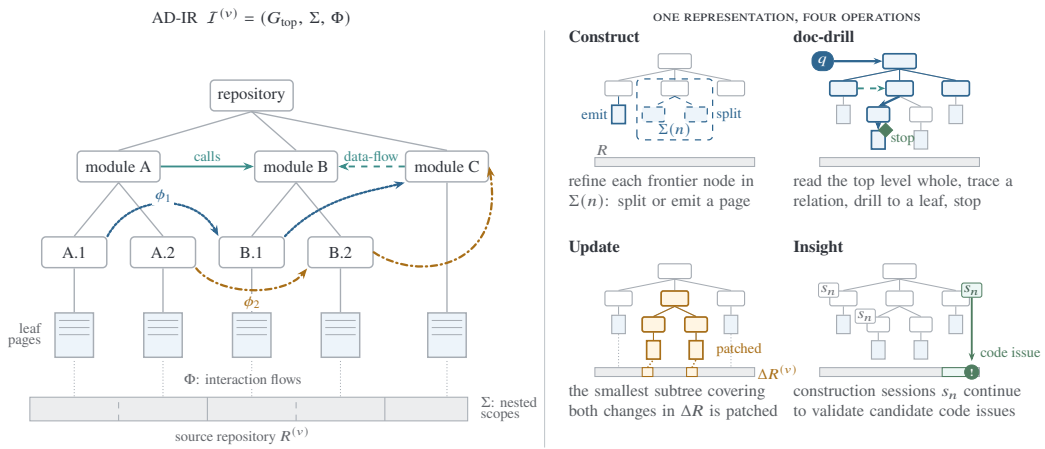


Figure 1: ACCEED overview. **Left:** AD-IR combines a recursive module hierarchy, typed edges within each local subgraph, source-grounded leaf pages, source scopes Σ, and a collection of interaction flows Φ. Two illustrative flow fragments, ϕ_1 and ϕ_2, connect participants in different subgraphs. **Right:** four operations on AD-IR; color marks the parts involved in each operation.

where $\mathcal{T}$ contains calls, dependency, data-flow, event, inheritance, and composition relations. Nesting these local graphs according to T forms $G^{(v)}_{\text{top}}$.

Source scopes are serialized in each node's `codeScope` field, with $\Sigma^{(v)}(r) = \bigcup_{n \in V} \Sigma^{(v)}(n)$ for the virtual root. For distinct $u, w \in C(n)$, AD-IR requires

$$\Sigma^{(v)}(u) \subseteq \Sigma^{(v)}(n), \qquad \Sigma^{(v)}(w) \subseteq \Sigma^{(v)}(n), \qquad \Sigma^{(v)}(u) \cap \Sigma^{(v)}(w) = \varnothing.$$

A child can own only files within its parent's source scope, and no file is assigned to two children of the same parent. Under these constraints, each covered file f has a unique deepest owner $o(f)$. Its owners are exactly the module nodes on the path from the root to $o(f)$:

$$\mathcal{O}(f) = \{n \in V \mid f \in \Sigma^{(v)}(n)\} = \text{Path}_T(r, o(f)) \setminus \{r\},$$

where $\text{Path}_T(r, o(f))$ denotes the nodes on that tree path. Proposition 1 in Appendix A.1 establishes this property.

For siblings $u, w \in C(n)$, a relation $(u, \tau, w) \in E_n$ records an interaction from u to w, labeled by its type τ and a textual description. If u uses shared code owned by w, the relation links the modules while the code remains in $\Sigma^{(v)}(w)$.

Interactions across local graphs are recorded in scenario-specific flows $\Phi^{(v)} = \{\phi_1, \ldots, \phi_m\}$. Each flow identifies participants by their documentation paths (`docPath`) and records an ordered sequence of steps. Each step specifies its source and target participants, relation type, action, explanatory detail, and source reference. Participants may lie in different branches and at different depths.

Together, these components form AD-IR (Figure 1, left):

$$\mathcal{I}^{(v)} = \left(G^{(v)}_{\text{top}}, \Sigma^{(v)}, \Phi^{(v)}\right).$$

Internal nodes support orientation, relations and flows support cross-module tracing, leaf pages provide implementation evidence, and $\Sigma^{(v)}$ grounds all of them in source code.

3.3 ACCEED: Construction, Access, and Maintenance

Recursive construction. ACCEED compiles repository source into AD-IR through a top-down tree of bounded agent tasks. A first pass identifies top-level modules and their source scopes. Each node is handled in a fresh agent session that receives the node's description, its source scope $\Sigma(n)$, the summaries and relations of the levels above it, and optional human-supplied notes. The agent thus knows where its scope sits in the repository without holding any other part of the analysis in context.

Refining node n yields its children, their scopes, and the typed relations among them, and marks each child either for further refinement or for emission as a leaf. The agent adds another level only when the code reveals several separable responsibilities within a scope. Each step keeps children's scopes within the parent's and disjoint from one another. Emitted leaves are written by separate sessions with the same bounded input. Recursive construction distributes analysis across module scopes, giving complex modules finer-grained documentation (Appendix D.1).

An optional checker reviews generated subgraphs for structural consistency. Accepted subgraphs and completed leaf pages are persisted, allowing independent tasks to run concurrently and interrupted runs to resume.

After the hierarchy and leaf pages are complete, a flow-analysis agent selects representative end-to-end scenarios from the generated documentation. It traces each scenario along module relations and verifies key interactions against the source. Participants are chosen at the level needed to explain the scenario, retaining high-level modules for pass-through behavior and selecting finer-grained submodules for critical branching or state changes. Algorithm 1 in Appendix A.2 gives the construction procedure.

Progressive access. Doc-drill guides agents from module overviews toward finer detail through subgraph expansion, relation and flow inspection, and leaf-page reads (Figure 1, right, and Figure 3 in Appendix A.3). The agent first selects relevant branches from node descriptions and source scopes. Within a subgraph, typed edges identify related nodes. For a cross-subgraph scenario, the agent follows a relevant flow's steps and uses participants' documentation paths to locate modules or pages. Source anchors in leaf pages lead back to the implementation.

At each step, the base agent uses the task and its observations so far to decide whether the current evidence supports the task or whether another expansion is needed, as in ReAct (Yao et al., 2023). An architectural question may stop at the top level, a local implementation question may reach a leaf, and a cross-module question may follow interaction-flow steps across branches. Cross-level search provides a jump only when the visible structure cannot locate a candidate node. Doc-drill (Appendix A.3) fixes the available actions and their priority, while the base agent chooses concrete ones at each step, so different tasks and base models produce different trajectories over the same AD-IR. Algorithm 2 gives the consumption loop. Appendix C.5 traces a command-approval question through the agent's document reads to its answer, with the visited tree shown in Figure 4.

Localized maintenance. Given the existing AD-IR $\mathcal{I}^{(v)}$ and the diff $\Delta R^{(v)}$ to $R^{(v+1)}$, the update agent identifies candidate leaves using the changed paths $D = \mathrm{paths}(\Delta R^{(v)})$:

$$\mathcal{L}_\Delta = \left\{ \ell \in \mathrm{Leaves}(G^{(v)}_{\mathrm{top}}) \;\middle|\; \Sigma^{(v)}(\ell) \cap D \neq \varnothing \right\}.$$

These leaves provide the initial update locations. For changes without a matching leaf scope, such as newly added files, the update agent navigates the hierarchy using source scopes and module responsibilities to locate an existing node or a suitable insertion point.

At each location, the agent updates documentation, scopes, and local graphs, adding, removing, or refining tree nodes as needed. Changes propagate upward while they affect parent descriptions or structure. After each upward pass, the proposed extension reanalyzes related flows against the updated AD-IR and current source, revising them only when needed and checking predecessor and successor participants for further effects. Newly affected participants enter the same update process. Algorithm 3 in Appendix A.4 gives the procedure.

3.4 Reusing Construction Sessions

Documentation construction requires agents to inspect implementations and trace module interactions. We observe that this exploration can also surface potential code issues, whose supporting context remains in the agent sessions even when it is not included in the generated documentation.

Insight therefore resumes a construction session s_n for source scope $\Sigma(n)$ with its original prompt and exploration history intact. It shifts the objective to validating these candidate issues through callers, tests, configuration, and adjacent modules. Supported findings are reported with source locations, impact, suggested fixes, and severity (Section 5.2).

4 EXPERIMENTS

4.1 EVALUATION QUESTIONS

We evaluate ACCEED through three questions, one for each requirement of Section 1. **RQ1 (structure)**: does complete AD-IR with doc-drill improve answer coverage over the same leaf pages alone? **RQ2 (access)**: how does doc-drill over the persistent tree compare with iterative source retrieval in answer quality and answer-stage cost? **RQ3 (update)**: after the code evolves, does localized update keep the tree useful, compared with the version frozen before the changes?

4.2 ACCEED-BENCH

In cross-repository development, an agent maintaining repository B may need knowledge of another repository A, such as a dependency. We use repository QA as a proxy for this knowledge-acquisition step, asking whether the agent can obtain the required understanding of A's behavior and module interactions from its documentation. Documentation conditions therefore provide documentation only, without access to the repository's source code. The evaluation measures knowledge acquisition and does not assess downstream editing success.

ACCEED-Bench contains 240 source-verified questions. The main split has 30 questions each for Codex, vLLM, KataGo, qBittorrent, SGLang, and Ollama, spanning local, cross-module, and repository-wide scopes. The update split has 30 questions each for Codex and vLLM about behavior changed by merged pull requests (Section 4.6). The primary metric, weighted scoring-point coverage, is the fraction of a question's total scoring-point weight awarded to an answer. Overall results average over questions and five answering configurations. We also report answer-stage cost and paired bootstrap 95% CIs for coverage differences. Appendix B gives the full setup.

Appendix C gives question–answer examples at all three scopes, their scoring criteria, and a step-by-step case study showing what the agent reads from AD-IR.

4.3 RQ1: DOES EXPLICIT STRUCTURE HELP?

RQ1 evaluates AD-IR's structural layer and doc-drill as a unit, with leaf-page text held fixed. Internal nodes give concise module overviews listing their children's responsibilities, source scopes, and relations; leaf pages give detailed documentation anchored in the source. These summaries, typed relations, interaction flows, and source scopes convey repository knowledge and guide doc-drill's navigation. We compare complete AD-IR with doc-drill (Structured) against the same leaf pages presented alone (Unstructured), measuring the joint contribution of this layer and its access protocol to answer coverage. An intermediate Hierarchy-only condition retains module descriptions, source scopes, tree navigation, and the same leaf pages, but removes typed relations and interaction flows.

Structured outperforms Unstructured for all five answering configurations, raising overall weighted coverage from 69.9% to 76.9% (Table 1). The gain is largest for the weakest configuration, Claude Haiku 4.5, and no confidence interval includes zero. Hierarchy-only falls between the two conditions, and adding typed relations and interaction flows accounts for most of the gain (Appendix D.3), supporting the added value of explicit interactions and their access.

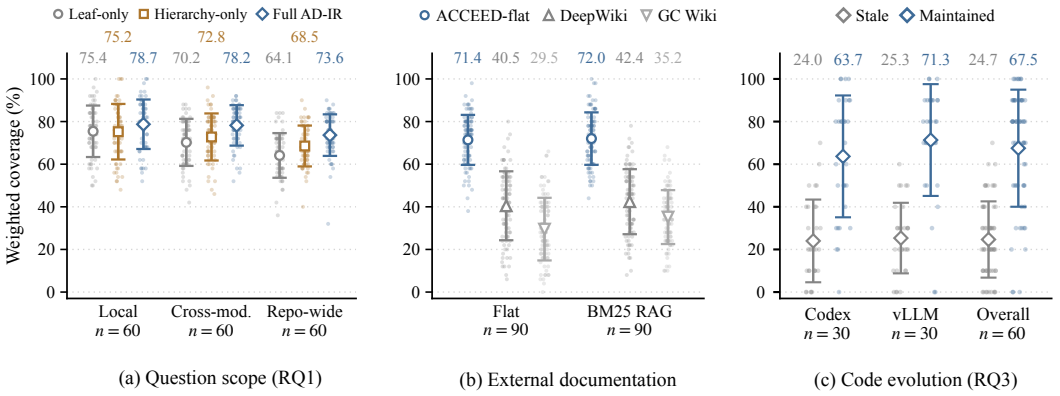
As Figure 2a shows, the gain grows from 3.3 points on local questions to 9.5 on repository-wide questions. A local question concerns a single module, so keyword search over flat pages can often reach the relevant page directly. Cross-module and repository-wide questions draw on three or more top-level modules, and the overviews, relations, and flows tell the agent which modules take part. Both conditions expose the same shell tools without a call limit. Structured makes more tool calls and costs more per question than Unstructured (Appendix D.3), so RQ1 shows a gain in answer quality at a higher answer-stage cost.

4.4 LEAF CONTENT AGAINST EXTERNAL DOCUMENTATION

RQ1 holds leaf content fixed. This experiment asks the complementary question of whether ACCEED's leaf pages themselves carry more usable knowledge than the pages of existing documentation systems.

Table 1: Weighted scoring-point coverage (%) for Unstructured and Structured context. Brackets give paired bootstrap 95% CIs over questions.

Answering configuration	Unstructured	Structured	Difference	95% CI
Claude Haiku 4.5	54.7	**66.6**	+11.9	[9.1, 14.7]
Claude Sonnet 5	70.3	**76.7**	+6.4	[4.6, 8.4]
GPT-5.6 Luna	73.4	**79.0**	+5.6	[3.9, 7.2]
GPT-5.6 Sol	79.0	**84.1**	+5.1	[3.4, 6.7]
GPT-5.6 Terra	72.2	**77.9**	+5.7	[3.7, 7.8]
Mean	69.9	**76.9**	**+6.9**	[5.9, 8.2]



(a) Question scope (RQ1) (b) External documentation (c) Code evolution (RQ3)

Figure 2: Weighted coverage (%). In (a,c), faint dots show per-question scores, outlined markers and numeric labels show means, and error bars show ±1 sample standard deviation across questions. (a) Unstructured (Leaf-only), Hierarchy-only, and Structured (Full AD-IR), with per-question scores averaged over five answering configurations. (b) Bars show mean coverage on the three shared repositories under Flat and BM25 RAG access. (c) Changed-code questions under stale and maintained documentation; Overall pools both repositories.

We compare ACCEED-flat (ACCEED's generated leaf Markdown only, which is the Unstructured condition restricted to the shared repositories), DeepWiki (The Cognition Team, 2025), and Google Code Wiki (Hurley et al., 2025) on the 90 shared questions from Codex, Ollama, and vLLM. Each source is accessed through two aligned interfaces: **Flat** supports page listing, reading, and keyword search, while **RAG** (Lewis et al., 2020) uses the same BM25 retriever (Robertson & Zaragoza, 2009). Questions, answering configurations, prompts, and judge remain fixed. The comparison evaluates exported corpora under these shared interfaces. The systems' native navigation and question-answering services are outside its scope.

As Figure 2b shows, ACCEED-flat has the highest coverage under both interfaces. With Flat access it averages 71.4%, against 40.5% for DeepWiki and 29.5% for Google Code Wiki, and RAG access gives the same ordering. Because the gap persists when all three corpora use the same BM25 retriever, it comes from what the pages contain. Table 7 in Appendix D.2 gives per-configuration results. Corpus size and organization are part of each system's output, so these results measure corpus utility under the shared access interfaces.

4.5 RQ2: Progressive Access vs. Source Retrieval

RQ2 compares Structured with Source Retrieval, which provides no pregenerated documentation and exposes source code through iterative BM25 retrieval (Zhang et al., 2023).

Averaged over all configurations and questions, Structured is 6.4 coverage points higher than iterative source retrieval and lowers mean answer-stage cost by 18.6% (Table 2). Both conditions make a similar number of tool calls, so the difference comes mainly from what each observation contains. The gain is largest for the weakest configuration, Claude Haiku 4.5, and nearly vanishes for the strongest, GPT-5.6 Sol, which reaches similar coverage at lower cost with AD-IR. Because AD-IR

Table 2: Weighted coverage (%) and answer-stage cost for Source Retrieval and Structured context. Brackets give paired bootstrap 95% CIs over questions.

Configuration	Source	Structured	Diff.	95% CI	Cost (Source)	Cost (Structured)
Claude Haiku 4.5	49.3	**66.6**	+17.3	[14.6, 20.1]	$0.204	$0.224
Claude Sonnet 5	69.2	**76.7**	+7.6	[5.4, 10.0]	$0.829	$0.735
GPT-5.6 Luna	74.3	**79.0**	+4.7	[2.4, 7.0]	$0.231	$0.224
GPT-5.6 Sol	83.8	**84.1**	+0.3	[−1.7, 2.4]	$1.579	$1.084
GPT-5.6 Terra	75.6	**77.9**	+2.3	[−0.1, 4.8]	$0.396	$0.369
Mean	70.4	**76.9**	**+6.4**	[4.7, 8.1]	**$0.648**	**$0.527**

is built with GPT-5.6 Sol, Haiku with AD-IR comes close to Sonnet 5 with source retrieval at 27% of its cost. Structured costs less for four of the five configurations and slightly more for Haiku (Figure 5 in Appendix D.4).

Table 8 in Appendix D.4 breaks down these results by question scope. For GPT-5.6 Sol, source retrieval does better on local questions, where the evidence sits in one module and can be read directly, while AD-IR does better on cross-module and repository-wide questions. Averaged over the five configurations, the two conditions are close on local questions, and AD-IR leads by a wider margin on the two distributed scopes. AD-IR therefore records cross-module semantics that a strong model does not fully recover from retrieved snippets, and gives weaker models understanding they would otherwise rebuild at every scope.

This comparison changes representation and access together. Isolating AD-IR from doc-drill would need iterative retrieval over the AD-IR corpus as a further condition.

On 336 questions from seven SWE-QA repositories, Structured scores higher on average than a function-chunk RAG baseline adapted from SWE-QA (Peng et al., 2026), and its advantage is larger on the two largest repositories by source-chunk count, SymPy and Django, than on the other five. Appendix E and Figure 6 give the setup and per-repository results.

4.6 RQ3: Does Localized Update Keep the Layer Current?

We construct real code-evolution trajectories for Codex and vLLM. From snapshot $R^{(0)}$, each repository advances through 150 consecutive merged pull requests in 15 windows of 10. The updater rolls the documentation forward window by window, and the update split's 30 questions per repository concern behavior that changed within the windows (Appendix B.4). Claude Sonnet 5 answers these 60 questions from documentation alone under two states, the *stale* documentation frozen at $R^{(0)}$ and the incrementally *maintained* documentation, and the same isolated GPT-5.6 Sol judge scores the answers. The frozen tree retains its pre-change documentation, so the score difference measures the benefit of maintenance on these changed-code questions.

As Figure 2c shows, maintenance raises mean coverage from 24.7% to 67.5%, and the gain holds on both repositories. The edits stay local. Maintenance modified fewer than one in ten documentation files in each repository, and for vLLM the added and deleted lines total 0.86% of the initial line count. These results support AD-IR's design for local maintenance as code evolves. Appendices B.4 and D.5 give the counting method, confidence interval, update costs, and file counts. The experiment measures the downstream utility of the updated context, not the semantic correctness of each edit, and it has no regenerated-from-scratch reference.

5 Real-World Evidence

5.1 Operational Deployment

ACCEED is deployed as the documentation layer for internal codebases, where one AD-IR per repository serves three audiences. Engineers browse the same representation as a site, module graph, drill-down tree, leaf pages, and interaction flows, to orient themselves before opening the code. Coding agents consult it through doc-drill as the knowledge base for development, and because the layer follows merged pull requests, it stays current as the code iterates. A business line registers the repositories it owns on one server, and every MCP tool (Anthropic, 2024) takes the project as

a parameter, so an agent working on one repository reaches the documentation of the services and libraries it depends on without cloning or reading their source. Because this deployment differs from the controlled evaluation, we draw no quantitative productivity conclusion from it.

5.2 Insight Findings

Insight produced 1,463 candidate findings across five open-source repositories: 60 on DeepEP, 49 on Codex, 70 on Ollama, 837 on SGLang, and 447 on vLLM. The Clash Verge case described below comes from a separate deployment. Of these, 938 concern correctness, 372 reliability, 47 security, and 106 performance, maintainability, or other issues. The counts describe candidate volume, not human-confirmed defects. Follow-up validation reproduced multiple bugs.

Two findings have been confirmed upstream. In Clash Verge, Insight flagged that saving the home-page card layout rewrote the global-hotkey setting with an unset patch value, silently re-enabling a hotkey the user had disabled. The maintainers confirmed the defect and merged a fix that removes the redundant write. In vLLM, a security finding was accepted into a private GitHub Security Advisory, with details withheld until disclosure.

6 Limitations

ACCEED incurs a higher cold-start token cost than approaches that retrieve source on demand. Initial construction costs roughly USD 10–200 per repository at API rates, with larger builds exceeding USD 200. The resulting AD-IR is reused across tasks and maintained through local updates as the repository evolves.

Multiple reproduced bugs and two upstream-confirmed cases show that Insight produces valuable findings, but we cannot adjudicate all 1,463 candidates, because reproduction needs the model weights those repositories serve, GPUs, and manual analysis. Unreviewed outputs therefore remain candidates, and we do not estimate precision. The continued agent also tends to report minor or moderate issues as high or critical, so every finding needs human triage before action.

7 Future Work

AD-IR could be extended to other external knowledge sources, such as technical documentation and issue collections, using its hierarchy, scopes, relations, and pages for progressive access and local maintenance. Such an extension would require domain-specific relation types, a source-ownership scheme, and update rules that route new or changed material to the appropriate nodes. AD-IR could also support a persistent memory system for agents, organizing task experience and verified findings for progressive retrieval and local revision as new evidence becomes available.

8 Conclusion

ACCEED represents repository knowledge as persistent structured context that agents construct, access, and maintain through AD-IR. Across six repositories and five answering configurations, complete AD-IR with doc-drill improves coverage over its leaf pages presented alone as flat documentation, with larger gains on cross-module and repository-wide questions. Against iterative source retrieval, it improves mean coverage at 18.6% lower mean answer-stage cost, with the largest gain for weaker answering models. After code evolution, localized maintenance restores coverage on changed-code questions from 24.7% to 67.5%. These results support using source scopes, module relations, and progressive access together for reusable repository context.

Reproducibility Statement

The supplementary material provides the ACCEED implementation, AD-IR query tools, agent and evaluation prompts, and scripts for ablations, evaluation, and result aggregation. ACCEED-Bench includes 180 main questions and 60 update questions, with reference answers, weighted scoring points, source evidence references, repository commit metadata, and update windows. The code

README gives setup and evaluation commands; Appendices A and B describe the algorithms and experimental settings. Generated AD-IR instances, external documentation snapshots, and original answer and judge outputs are not included.

Ethics Statement

The controlled experiments use public repositories and no human participants or private user data. Insight outputs remain candidate findings because they may include false positives or undisclosed vulnerabilities. We privately report only reproduced, high-impact issues through upstream channels, avoid bulk reports that burden maintainers, and withhold exploitable details until remediation or public disclosure.

AI Use Statement

The authors used LLM assistants for experiment scripts, result checking, paper structuring, and language editing. All AI-assisted content, numbers, citations, and claims were reviewed by the authors, who take responsibility for the final work. Models used as system components, answerers, and judges are documented in the paper and supplementary material.

References

Anthropic. Introducing the Model Context Protocol. Anthropic News, November 2024. URL https://www.anthropic.com/news/model-context-protocol. Accessed: 2026-09-26.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/25f7be9694d7b32d5cc670927b8091e1-Abstract-Conference.html.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Shankar Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE): 675–698, 2024. doi: 10.1145/3643757.

Yoshua Bengio and Yann LeCun. Scaling learning algorithms toward AI. In *Large-Scale Kernel Machines*, pp. 321–360. MIT Press, 2007. doi: 10.7551/mitpress/7496.003.0016.

Cognition. DeepWiki MCP. Devin Docs, n.d. URL https://docs.devin.ai/work-with-devin/deepwiki-mcp. Accessed: 2026-09-20.

Katherine R. Dearstyne, Alberto D. Rodriguez, and Jane Cleland-Huang. Supporting software maintenance with dynamically generated document hierarchies. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 426–437. IEEE, 2024. doi: 10.1109/ICSME58944.2024.00046.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024. URL https://arxiv.org/abs/2404.16130.

FSoft-AI4Code. CodeWiki source code. GitHub repository, 2026. URL https://github.com/FSoft-AI4Code/CodeWiki/tree/9ed15bad90170bb4db6b708210e7bc90a130a953. Revision 9ed15ba (2026-09-11). Accessed: 2026-09-20.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. doi: 10.1162/neco.2006.18.7.1527.

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. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 5812–5827. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.288.

Fergus Hurley, Pedro Rodriguez, Rafael Marques, and Omar Shams. Introducing Code Wiki: Accelerating your code understanding. Google Developers Blog, November 2025. URL https://developers.googleblog.com/introducing-code-wiki-accelerating-your-code-understanding/. Accessed: 2026-09-20.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638.

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. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 436–464. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-demo.46.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/4a4a3c197deac042461c677219efd36c-Abstract-Conference.html.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Zhongqiang Pan, Chuanyi Li, Wenkang Zhong, Yi Feng, Bin Luo, and Vincent Ng. RepoRepair: Leveraging code documentation for repository-level automated program repair. *arXiv preprint arXiv:2603.01048*, 2026. URL https://arxiv.org/abs/2603.01048.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions? In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8230–8245. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.402.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009. doi: 10.1561/1500000019.

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. RAPTOR: Recursive abstractive processing for tree-organized retrieval. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/8a2acd174940dbca361a6398a4f9df91-Abstract-Conference.html.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31210–31227. PMLR, 2023.

The Cognition Team. DeepWiki: AI docs for any repo. Cognition, May 2025. URL https://cognition.com/blog/deepwiki. Accessed: 2026-09-20.

Martin Vogel, Falk Meyer-Eschenbach, Severin Kohler, Elias Grünewald, and Felix Balzer. Codebase-Memory: Tree-Sitter-based knowledge graphs for LLM code exploration via MCP. *arXiv preprint arXiv:2603.27277*, 2026. URL https://arxiv.org/abs/2603.27277.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Pengfei Gao, and Chao Peng. Evaluating repository-level software documentation via question answering and feature-driven development. *arXiv preprint arXiv:2604.06793*, 2026. URL https://arxiv.org/abs/2604.06793.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi: 10.1145/3715754.

Dong Xu, Mingwei Liu, Xiwen Wang, Jianfeng Zhong, and Zibin Zheng. RepoDoc: A knowledge graph-based framework to automatic documentation generation and incremental updates. *arXiv preprint arXiv:2604.26523*, 2026. URL https://arxiv.org/abs/2604.26523.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 17577–17604, 2025. doi: 10.52202/085713-0593.

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 *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 460–471. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-demo.44.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601.

Yuexi Yang, Alyssa Wu, Ji Luo, Richeng Xuan, Zhichao Hu, Yuhong Liu, and Zhen Qin. RepoProbe: Benchmarking architecture-aware repository comprehension with checklists. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2026. URL https://arxiv.org/abs/2608.04783. Accepted for publication.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://iclr.cc/virtual/2023/oral/12647.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025. URL https://arxiv.org/abs/2512.24601.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.151.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2024. doi: 10.1145/3650212.3680384.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46595–46623, 2023.

A METHOD DETAILS

A.1 STRUCTURAL PROPERTIES OF AD-IR

We consider the source selected for representation at a fixed repository version and suppress the version superscript. Let V be the actual module nodes, including leaves, and interpret each $\Sigma(n)$ as the set of repository files denoted by its `codeScope`. Write $U = \bigcup_{n \in V} \Sigma(n)$. For the proof, attach the top-level modules to a virtual root r with $\Sigma(r) = U$, giving a finite rooted tree T. Assume each child scope is contained in its parent's scope and distinct sibling scopes are disjoint, including the top-level modules. Typed relations and interaction flows are separate from the tree edges. Let $\mathrm{Anc}_V^*(n)$ contain n and its ancestors in V, excluding r.

Children need not exhaust their parent's source scope: $\bigcup_{c \in \mathrm{Children}(n)} \Sigma(c) \subseteq \Sigma(n)$ may be a strict inclusion, and U need not contain every repository file. At each level, refinement selects responsibilities that warrant further documentation. For example, a TypeScript package's lockfile may remain within the package scope without a dedicated child node or page when its contents are unnecessary to explain the documented functionality. The representation permits such omissions, so a file's deepest owner can be an internal node. Changes to omitted files are still inspected for effects on documented behavior (Appendix A.4).

Proposition 1 (Unique source-ownership chain)**.** *For each file $f \in U$, its owners $\mathcal{O}(f) = \{n \in V \mid f \in \Sigma(n)\}$ lie on one ancestor chain. Consequently, f has at most one owner at each depth and a unique deepest owner $o(f)$, with $\mathcal{O}(f) = \mathrm{Anc}_V^*(o(f))$.*

Proof. Suppose two owners a, b are incomparable in the ancestor order. Their lowest common ancestor has distinct children u, w on the paths to a, b. Repeated scope containment gives $f \in \Sigma(u) \cap \Sigma(w)$, contradicting sibling disjointness. Thus all owners form a chain. This chain is nonempty because $f \in U$ and finite because T is finite, so its deepest node $o(f)$ exists and is unique. Every other owner is an ancestor of $o(f)$; conversely, every actual ancestor contains f by scope nesting. This proves the equality and the per-depth uniqueness. □

Proposition 2 (Ancestor-path form of source overlap)**.** *For a set $D \subseteq U$ of changed files within the represented source, the nodes whose source scopes intersect D satisfy*

$$\mathcal{C}(D) = \{n \in V \mid \Sigma(n) \cap D \neq \varnothing\} = \bigcup_{f \in D} \mathrm{Anc}_V^*(o(f)).$$

Proof. If $n \in \mathcal{C}(D)$, choose $f \in \Sigma(n) \cap D$. Then $n \in \mathcal{O}(f) = \mathrm{Anc}_V^*(o(f))$ by Proposition 1, so n belongs to the union. Conversely, if $n \in \mathrm{Anc}_V^*(o(f))$ for some $f \in D$, scope nesting gives $f \in \Sigma(n)$, hence $n \in \mathcal{C}(D)$. The two inclusions establish the equality. □

These propositions describe source ownership under the stated scope constraints. Configurations that allow overlapping sibling scopes or out-of-parent scopes fall outside these assumptions. Source overlap identifies candidate ancestor paths. Maintenance may stop earlier when a node needs no revision, and new subtrees are handled separately (Appendix A.4). Neither proposition certifies generated descriptions or revision decisions.

A.2 RECURSIVE CONSTRUCTION

Algorithm 1 gives the checker-enabled construction loop. Each agent receives the node description, source scope, ancestor summaries and relations, and applicable domain knowledge through CONTEXT. GROUND writes each leaf page in a separate session.

GENERATECHECKREPAIR combines typed-schema parsing with a checker session that reviews node identifiers, relation targets, source paths, scope overlap, and description completeness. Feedback goes back to the generating session for repair. This agent review does not establish the scope assumptions in Appendix A.1.

Each checker cycle permits up to five repairs after the initial candidate, then raises an error if review still fails. Task-level retries may start another attempt; resumed attempts can restore stored generator

Algorithm 1 Recursive AD-IR Construction

Require: Repository snapshot R; optional domain knowledge K
Ensure: AD-IR $(G_{\text{top}}, \Sigma, \Phi)$ with leaf pages attached, or a reported failure

$G_{\text{top}} \leftarrow$ GENERATECHECKREPAIR(PARTITION, (R, K))
Persist G_{top}; $\Sigma \leftarrow$ SCOPES(G_{top}); $F \leftarrow$ TOPNODES(G_{top})
while $F \neq \varnothing$ **do**
    $n \leftarrow$ POP(F)
    $c_n \leftarrow$ CONTEXT(n, G_{top}, K)
    $(C_n, E_n, z) \leftarrow$ GENERATECHECKREPAIR(REFINE, (R, c_n))
    Attach fresh children C_n and their relations E_n beneath n
    $\Sigma \leftarrow$ SCOPES(G_{top}); persist the refinement of n
    for each $v \in C_n$ **do**
        if $z_v = \text{split}$ **then**
            $F \leftarrow F \cup \{v\}$
        else
            $c_v \leftarrow$ CONTEXT(v, G_{top}, K)
            $P_v \leftarrow$ GROUND(R, c_v)
            Attach P_v to v and persist the page
        end if
    end for
end while
$\Phi \leftarrow$ TRACE(R, G_{top}); persist Φ
return $(G_{\text{top}}, \Sigma, \Phi)$

and checker sessions. Exhausted retries mark the task as failed while preserving completed artifacts for resumption.

Accepted subgraphs are saved before their child tasks become eligible; leaf pages are saved when writing finishes. The frontier F records these dependencies, while independent graph and page tasks run concurrently with persisted status. After the tree and pages are complete, TRACE invokes the FlowAnalyzer to derive Φ from the source and documentation, recording representative scenarios, participants, and interaction steps. Optional human review and checker-free operation are not shown in the algorithm.

A.3 THE DOC-DRILL ACCESS PROTOCOL

The protocol instructs agents to begin with the top-level subgraph G_{top}, including every top-level module's description and `codeScope`. Each subsequent READGRAPH(n) returns all immediate child-node entries of n together; their descendant subgraphs and leaf pages are read only when selected. Figure 3 shows one numbered access sequence, and Algorithm 2 gives the action priorities. Relations are fetched on demand: local edges identify related siblings, while interaction flows use participants' `docPath` values to locate nodes across branches and depths, without reading each ancestor subgraph.

For precise modification or evidence checks, source anchors lead to the matching snapshot $R^{(v)}$ when access is permitted; documentation-only conditions omit source reads. If the visible structure cannot locate a node, cross-level search supplies a path for further navigation. The base agent selects actions (Yao et al., 2023) and stops at any depth once evidence is sufficient (cf. Asai et al., 2024) or no useful permitted read remains, reporting unresolved gaps. These priorities are prompted, without a separate retrieval model or an enforced scheduler. O denotes the observation sequence, and $\|$ appends an observation.

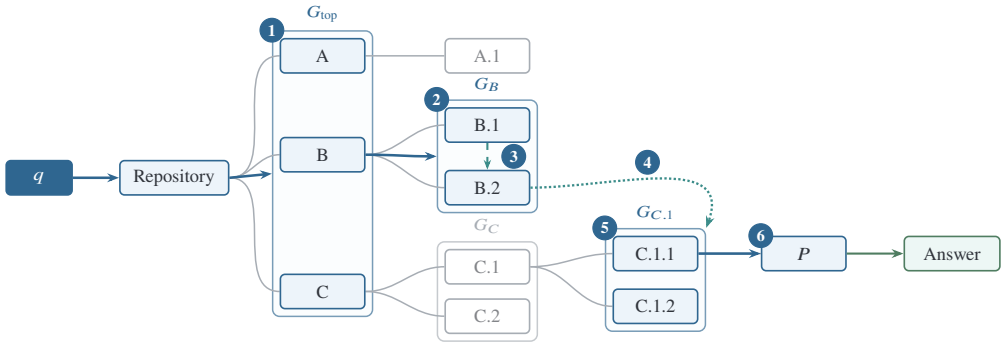


Figure 3: One doc-drill trajectory: (1) G_{top}, (2) G_B, (3) local edges, (4) interaction flows, (5) $G_{C.1}$, and (6) page P. The flow locates the third-level subgraph directly, without reading G_C. Each blue enclosure groups the nodes returned by one graph read.

Algorithm 2 Task-Guided AD-IR Consumption (doc-drill)

Require: Task q; AD-IR $\mathcal{I}^{(v)}$; permitted read tools $\mathcal{A}$
Ensure: Observations O for answering or acting, and any unresolved evidence gaps

$O \leftarrow [\textsc{ReadTop}(\mathcal{I}^{(v)})]$
while $\neg\textsc{Sufficient}(q, O)$ **do**
  Consider only useful, untried reads permitted by $\mathcal{A}$
  if a relevant interaction still needs tracing **then**
    $a \leftarrow \textsc{SelectTrace}(q, O)$
  else if a visible internal node needs finer detail **then**
    $n \leftarrow \textsc{SelectModule}(q, O)$; $a \leftarrow \textsc{ReadGraph}(n)$
  else if a relevant leaf page remains unread **then**
    $n \leftarrow \textsc{SelectLeaf}(q, O)$; $a \leftarrow \textsc{ReadPage}(n)$
  else if source access is permitted and a known anchor needs verification **then**
    $a \leftarrow \textsc{ReadSource}(R^{(v)}, \textsc{SelectAnchor}(q, O))$
  else if the visible structure cannot locate a needed node **then**
    $a \leftarrow \textsc{SearchNodes}(\mathcal{I}^{(v)}, \textsc{Keywords}(q, O))$
  else
    break
  end if
  $O \leftarrow O \parallel [\textsc{Execute}(a)]$
end while
return $(O, \textsc{Unresolved}(q, O))$

A.4 Localized Maintenance

Algorithm 3 maintains a queue of update locations and impact evidence. For each item, the updater completes one upward pass before checking related flows, which may enqueue affected participants from other branches. Each item (n, e) identifies a node or insertion location n and the code changes or interaction effects e to inspect. For a directly affected leaf $n \in \mathcal{L}_\Delta$, e_n is its relevant code diff. Decisions use this evidence, the target source, and the existing AD-IR. The flow checks form the proposed extension to the evaluated upward pass.

Changed paths D include both source versions so that deleted or renamed files can be mapped to existing nodes. When no leaf scope matches, the update agent navigates the hierarchy using source scopes and module responsibilities. It selects an existing node that can cover the change or an insertion point in the smallest suitable subgraph, using the top-level graph if no module fits. An insertion point is represented in Q by its enclosing module. During local maintenance, new responsibilities become leaves or subtrees as needed, while an existing leaf can be extended when its responsibility fits the new source. Changes to source omitted under selective refinement are inspected at the enclosing subgraph and require no edit when its documentation and structure remain accurate.

Algorithm 3 Bottom-Up AD-IR Maintenance with a Flow-Guided Extension

Require: AD-IR $\mathcal{I}^{(v)}$; diff $\Delta R^{(v)}$ from $R^{(v)}$ to $R^{(v+1)}$; target snapshot $R^{(v+1)}$
Ensure: Updated hierarchy, scopes, relations, leaf pages, and flows, with an update report

$(G', \Sigma', \Phi') \leftarrow (G^{(v)}_{\text{top}}, \Sigma^{(v)}, \Phi^{(v)})$
$D \leftarrow \text{paths}(\Delta R^{(v)})$
Initialize queue Q with (n, e_n) for each leaf $n \in \mathcal{L}_\Delta$
Open one updater session with the diff, target source, and existing AD-IR
for each changed region without a matching leaf scope **do**
    Navigate the hierarchy by source scope and responsibility to locate a node or insertion point n
    Enqueue (n, e) with the region's relevant diff e
end for
while $Q \neq \varnothing$ **do**
    Remove an item (n, e) with new, unchecked impact evidence from Q
    $A \leftarrow \{(n, e)\}$; $u \leftarrow n$
    while u exists **do**
        Save u's parent p and current documentation path
        Update documentation, scopes, and local graphs (nodes and typed relations) as needed
        Add, remove, or refine tree nodes as module responsibilities require
        Record behavior and structural effects, including affected paths, in A
        if p is absent or p's description, scope, and local graph remain unaffected **then**
            break
        end if
        $(u, e) \leftarrow (p, \text{effects relevant to } p)$
    end while
    for each flow $\phi \in \textsc{RelatedFlows}(\Phi', A)$ with new impact evidence **do**
        Reanalyze ϕ's interactions using updated (G', Σ') and the target source
        if ϕ's steps, descriptions, or participant references need revision **then**
            Update ϕ in Φ'
        end if
        Check predecessor/successor participants for further effects
        Enqueue newly affected participants with their impact evidence
    end for
end while
Verify that all flow participant `docPath` references resolve
Persist edited artifacts, retaining unaffected subtrees
$\mathcal{I}' \leftarrow (G', \Sigma', \Phi')$
return $(\mathcal{I}', \text{update report})$

Updating a local graph $g_p = (C(p), E_p)$ revises its child-node entries and typed relations. Adding or removing modules changes $C(p)$ and the corresponding ownership-tree edges. Refining a leaf creates a child graph and turns the leaf into an internal node.

The upward pass checks the current node using its impact evidence e, the diff, and the target source. It stops when the parent's description, scope, and local graph remain unaffected. Saving the parent before an edit allows propagation to continue after node deletion. Structural edits preserve a single parent per node and nested, disjoint scopes. Deleting a source file need not remove its node if the node still has a valid responsibility. All edits use tools in the same updater session, assuming additions in an individual pull request are small enough to handle without recursive construction tasks.

In the flow-guided extension, A retains the initial source-change evidence and the effects found during the upward pass, even when a node's documentation needs no edit. RELATEDFLOWS finds flows involving these affected nodes, including participants representing their enclosing modules and references affected by deletion or relocation. For each selected flow, the agent retraces its scenario using the updated hierarchy, documentation, and target source. It compares the stored steps, descriptions, and participant references with this analysis and revises the flow only where needed. Predecessors and successors are determined by the flow's interaction steps. The agent checks whether changed interactions affect their behavior or documentation, even when their own source files are unchanged. These participant checks also run when the flow itself needs no revision. Newly affected participants enter Q for the same upward maintenance and flow checks. Queue insertion merges pending evidence for the same node and skips evidence already checked. Nodes and flows are revisited only when new impact evidence arrives. Paths saved before structural edits identify `docPath` references to repair or remove. The report records revisions and unresolved checks.

RQ3 evaluates only the upward maintenance pass. The update costs in Section 4.6 exclude flow reanalysis. The flow-guided extension has not been evaluated in that experiment.

B ACCEED-BENCH EVALUATION SETUP

B.1 QUESTION CONSTRUCTION

We wrote 30 questions per repository (180 in total) in developer terms, without file names or code identifiers. Local questions concern one module's contract, boundary behavior, configuration effects, or failure handling. Cross-module questions require evidence from at least three top-level modules. Repository-wide questions cover every subsystem involved in a scenario, with scoring points assigned per module so that omissions lose credit.

Each item has a mechanism-level reference answer and 3–10 source-supported scoring points with weights summing to 10. Supporting source files are retained for verification and shown to neither the answering agent nor the judge. Each question draws on a median of 7 files (range 2–21).

B.2 ANSWERING AND SCORING

Unless noted otherwise, ACCEED runs GPT-5.6 Sol through Codex in all roles. The five answering configurations are Claude Haiku 4.5 and Claude Sonnet 5 through Claude Code, and GPT-5.6 Luna, Sol, and Terra through Codex. Instructions and response format stay fixed across conditions; only the knowledge source and access interface change. GPT-5.6 Sol judges each answer against the scoring points in an isolated session without tools, following the LLM-as-a-judge setup (Zheng et al., 2023).

Weighted scoring-point coverage is the awarded weight divided by the total weight for each question. Configuration-level results average the 180 questions; overall results average the five configurations. Differences are computed per question before rounding. Paired bootstrap 95% confidence intervals use 1,000 resamples over questions.

We record tool calls, answer-stage tokens, and answer-stage costs. Claude costs are SDK-reported; Codex costs use token counts and per-token prices fixed during the experiment. These costs exclude document construction, incremental maintenance, and offline index construction.

B.3 ACCESS CONDITIONS AND RETRIEVAL

Table 3 lists the context available in the main conditions. Documentation conditions exclude source code.

The Unstructured export copies leaf-page text unchanged into a single directory and omits the JSON graphs and interaction flows. Internal-node descriptions provide higher-level summaries of child-module responsibilities, while leaf pages provide detailed source-grounded accounts. The export therefore preserves all leaf-level explanations. Module descriptions and relation text belong to the structural layer because they encode responsibilities, module boundaries, and interactions, including

Table 3: Context available to the answering agent in each main condition. Documentation conditions exclude source code; Source Retrieval exposes it through iterative retrieval.

Condition	Context available to the agent
Structured	Complete AD-IR: recursive hierarchy, typed relations, interaction flows, `codeScope`, and leaf pages, accessed through doc-drill.
Hierarchy-only	The same leaf pages, recursive hierarchy, module descriptions, and `codeScope`, with tree-based doc-drill access. Typed relations and interaction flows are removed.
Unstructured	The same leaf-page text in a flat directory, with the JSON structural layer and doc-drill removed.
Source Retrieval	No pregenerated documentation; the agent iteratively queries source code through a BM25 search tool.

when rendered as prose. The full-versus-Unstructured comparison removes this layer together with its navigation protocol. For Hierarchy-only, we keep node descriptions and source scopes, empty local edge lists, remove `flows.json`, and omit relation and flow commands from the doc-drill interface.

Measured as stored UTF-8 bytes, JSON accounts for 8.99% of the combined Markdown and JSON documentation across the six RQ1 corpora: 9.37% for KataGo, 11.15% for Codex, 11.22% for Ollama, 11.90% for qBittorrent, 7.57% for SGLang, and 9.30% for vLLM. The denominator includes leaf pages, graph files, and interaction flows; it excludes tool scripts, skill instructions, and runtime logs. Codex and vLLM use the retained RQ1 snapshots preceding incremental maintenance.

For documentation RAG, pages are split at headings and paragraphs into chunks of at most roughly 2,400 characters. ACCEED-flat corpora range from 3,010 chunks (qBittorrent) to 38,091 (SGLang). The agent can query the BM25 retriever repeatedly, receiving the top 10 chunks per call. Flat access provides page listing, page reading, and keyword search over the same corpus. The supplementary material records the exact chunking and BM25 settings, which stay fixed across documentation sources within RAG access.

Google Code Wiki documentation was unavailable for KataGo and qBittorrent at collection time. The three-way comparison therefore uses 90 shared questions from Codex, Ollama, and vLLM; the five-repository comparison with DeepWiki uses 150 questions, adding KataGo and qBittorrent. Questions, answering configurations, prompts, judge, and access interfaces stay fixed across sources within each comparison. In Table 7, each configuration cell aggregates 90 answers and each mean aggregates 450; each condition in the five-repository comparison aggregates 750 answers.

B.4 RQ3 Evaluation Protocol

Codex advances from commit `768848ab` (2026-05-26) to `3b7334d0` (2026-05-31), and vLLM from `c621af16` (2026-06-14) to `731fb332` (2026-06-18). Each trajectory contains 150 merged pull requests in 15 windows of 10. After the updater processes a window, a Claude Fable 5 session with access to its diffs and source proposes up to two questions about the changed code's current behavior, which the authors review. Questions mention neither pre-change behavior nor specific pull requests, and every question's evidence files intersect the window's changed files.

RQ3 file counts include Markdown pages and JSON structure files. Textual change is the sum of added and deleted lines between the initial and final AD-IR, divided by the initial line count, using Git's Myers line-diff algorithm. The count includes added and removed files; replacing a line counts as one deletion and one addition. Appendix D.5 reports the maintenance statistics.

C Benchmark Examples and Agent Access Case Study

The following examples cover local, cross-module, and repository-wide evidence scopes. Each includes a question, a condensed reference answer, and weighted scoring criteria. Questions are translated from the original Chinese items; weights sum to 10 per question.

C.1 Local Example: Locating Edits in a Text Patch

Repository and item. Codex, `baseline-01`.

Question. The agent updates existing files using a custom text-patch format. Each edit block may include a context line beginning with two at signs, old and replacement lines, and an end-of-file marker. How does the patch engine locate these edits? Explain the search-order constraints across blocks, the successive matching rules when exact matching fails, the effect of the end-of-file marker, and the retry when the old lines end with an empty line. When does matching fail completely, what information is reported, and can such a failure leave the target file partially modified?

Reference answer, condensed. The engine processes blocks with a forward cursor. A context-line match advances the cursor past that line; matching the old lines advances it past the matched region. Matching progressively permits trailing whitespace differences, surrounding whitespace differences, and normalized Unicode punctuation and spaces after first attempting exact equality. An end-of-file marker anchors matching to the file suffix rather than scanning forward from the cursor. If matching fails and the old lines end with an empty element, the engine removes that element, and the corresponding trailing empty element in the replacement when present, before retrying. Missing context or unmatched old lines produce an error identifying the file and missing content. Replacement ranges are computed before writing, so a location failure causes no partial write to that file.

This local question requires boundary behavior and failure semantics in addition to the component's main purpose.

C.2 Cross-Module Example: A Command-Approval Round Trip

Repository and item. Codex, `edges-01`.

Question. While using the agent's full-screen terminal interface, the model requests a command that cannot be approved automatically, and a confirmation prompt appears. Describe the complete approval round trip. How does the core runtime suspend the tool call and deliver the request across process boundaries to the interface? How does the user's decision return and resume the waiting call? Why does selecting "allow for this session" suppress subsequent prompts for the same command? If an approved command is then blocked by the sandbox, what happens before it is retried?

Reference answer, condensed. The core registers a one-shot response channel under the approval identifier before emitting an approval event, then awaits the decision. The app-server converts the event into a server-initiated RPC request. The terminal interface records the correspondence between the approval identifier and RPC request identifier and uses it to return the user's choice. The service maps that response to a core approval operation, which resolves the pending channel; cancellation aborts the task. Malformed or failed client responses are treated as denial. Session approval is cached using the normalized command, working directory, and permissions, allowing subsequent matching calls to bypass another approval exchange. If sandbox execution is denied and escalation is permitted, the orchestrator prepares a retry reason and requests approval again unless an exemption applies, then performs one permitted retry.

A complete answer connects state maintained by the core runtime, the service, and the terminal interface. Describing the confirmation prompt alone leaves most scoring points uncovered.

C.3 Repository-Wide Example: Loading a LoRA Adapter at Runtime

Repository and item. vLLM, `structure-01`.

Question. An online inference service uses vLLM. We want to load a new LoRA adapter through an administrative API without restarting the service, so subsequent generation requests can select it by name. Which subsystems participate in this capability, and what does each own? Trace the loading operation from the HTTP request through the device processes to the model runtime where the adapter weights are loaded.

Reference answer, condensed. Four subsystems participate. The API layer validates the adapter name and path, serializes same-name operations, handles duplicate loads, assigns an integer identifier, and calls the engine client. It registers successful loads for subsequent name-based selection and maps failures to HTTP errors. The engine control plane forwards loading as a utility call to the engine-core process, which delegates directly to the executor without scheduling it as a generation request. The executor broadcasts the operation to all workers and requires success from every rank.

Scoring criterion (Codex `baseline-01`)	Weight
Forward cursor and ordering of context and old-line searches	2
Four matching levels: exact, trailing whitespace, surrounding whitespace, and Unicode normalization	3
End-of-file anchoring without fallback to a forward scan	2
Retry after removing trailing empty elements	2
Error information and absence of partial writes on location failure	1

Scoring criterion (Codex `edges-01`)	Weight
Register the pending channel before emitting the event; await its decision and handle channel removal as abort	2
Convert the event to a server-initiated RPC request; handle malformed or failed responses as denial	2
Correlate interface and RPC identifiers, return the decision to the core, resume the waiter, and distinguish task cancellation	3
Cache session approval using the command, working directory, and permissions	1
Explain sandbox denial, escalation eligibility, renewed approval, and the retry	2

Each worker delegates to its model runner's LoRA support, which also constructs adapter mappings and activates adapters for subsequent batches. The worker-side adapter manager loads checkpoint weights, manages adapter slots, and performs LRU eviction. The adapter request object carries the shared contract across these layers.

Here, repository-wide coverage means accounting for every subsystem involved in the requested capability. The API endpoint alone cannot explain how the adapter becomes available on all devices.

C.4 What the Agent Sees through doc-drill

In the evaluated interface, the agent accesses AD-IR through the `browse.mjs` command-line tool. A graph read returns formatted text containing module descriptions, source scopes, and commands for opening children. Child entries distinguish further graphs from leaf pages. Relations are requested separately with `--edges`; interaction flows are requested with `--flows`; leaf content is read with `--read`.

In the command-approval case, step 9 returns the following leaf entry (translated; `DOC_ROOT` denotes the documentation directory):

> `### ToolOrchestratorFlow _(page)_`
>
> The approval, sandbox-selection, and retry orchestrator for generic ToolRuntime implementations. It determines whether execution is allowed using approval policy, guardian/strict auto review, permission hooks, and tool-specific requirements; selects the first sandbox attempt; manages the network-approval lifecycle; and requests an unsandboxed retry after sandbox denial when permitted by policy.
>
> `- Next: node browse.mjs DOC_ROOT codex AgentRuntime/ToolPlanningAndDispatch/ApprovalSandboxNetworkEvents/ToolOrchestratorFlow --read`
>
> `- Code scope: codex-rs/core/src/tools/orchestrator.rs`
>
> `- Edges (2): node browse.mjs DOC_ROOT codex AgentRuntime/ToolPlanningAndDispatch/ApprovalSandboxNetworkEvents --edges ToolOrchestratorFlow`

This entry exposes a component's responsibility, its source scope, and the next available reads before loading its detailed page. Reading the parent graph returns all five immediate children together: `SandboxApprovalContracts`, `ToolOrchestratorFlow`, `ManagedNetworkApproval`, `ToolEventBridge`, and `TurnDiffTracking`. Their page bodies remain outside the observation until requested.

C.5 Following the Command-Approval Question

Claude Sonnet 5 answers the command-approval question using five graph reads, one flow read, one relation read, three searches, and five leaf-page reads. Table 4 follows the complete sequence, beginning with the doc-drill skill. Figure 4 locates the graph and page reads in the documentation tree.

Scoring criterion (vLLM `structure-01`)	Weight
API validation, locking, duplicate handling, identifier allocation, error mapping, and successful registration	3
Utility-call forwarding through the engine control plane, bypassing generation scheduling	2
Broadcast to every worker, all-rank success, and batch-specific mapping and activation	3
Checkpoint loading, adapter-slot management, LRU eviction, and the shared request contract	2

Table 4: Recorded access sequence for Codex `edges-01`. Paths are shortened to node names. Steps 12–13 pipe search output through `head -80`.

Step	Call	Returned information
1	Skill: doc-drill	Documentation model and navigation commands.
2	Read top	Ten top-level modules, descriptions, source scopes, and next commands.
3	`--flows`	Seven flows, including tool approval and sandbox execution.
4	Graph: AgentRuntime	Runtime submodules, including tool dispatch and session lifecycle.
5	`--search approval`	70 matches; the tool returns a preview and a saved-output path.
6	Graph: ApprovalReviewAndSafety	Guardian review, patch safety, network policy, and review-related children.
7	ToolPlanningAndDispatch `--edges`	22 relations among tool-planning and dispatch components.
8	Graph: ToolPlanningAndDispatch	Nine children, including ApprovalSandboxNetworkEvents.
9	Graph: ApprovalSandboxNetworkEvents	Five children covering contracts, orchestration, network approval, events, and diffs.
10	Page: ToolOrchestratorFlow	Initial approval, sandbox selection, and escalation/retry conditions.
11	Page: SandboxApprovalContracts	Runtime contracts and session approval-cache behavior.
12	`--search request_approval`	No matches.
13	`--search ExecApprovalRequest`	Three matches in the core, app-server, and terminal interface.
14	Page: ApprovalRequestWaiters	Pending approval channels, event emission, response notification, and cleanup.
15	Page: CommandExecutionApprovalBridge	Server-initiated RPC requests, decision mapping, and error handling.
16	Page: ApprovalEventModels	Terminal-side approval event fields and effective approval identifiers.

From the overview to candidate modules. At step 2, the overview identifies approvals as a responsibility of `AgentRuntime`. Step 3 exposes the following flow fragment:

> **Model tool calls through approval and sandbox execution**
>
> `ToolPlanningAndDispatch` → `ApprovalReviewAndSafety` [calls] — request approval and safety checks.
>
> Command, patch, network, and privilege-escalation calls first pass through approval policy, guardian, or hooks.
>
> code: `codex-rs/core/src/tools/orchestrator.rs`

The description of `ApprovalSandboxNetworkEvents` at step 8 names the approval cache and retry workflow. The agent expands it at step 9 and reads two leaves at steps 10–11.

Crossing branches through search results. The search at step 12 returns no matches. Searching for `ExecApprovalRequest` finds three pages under `AgentRuntime`, `AppServerPlatform`, and `UserSurfaces`; steps 14–16 read them without expanding their ancestor graphs.

From a page to the answer. The core waiter page at step 14 gives the following ordering (translated excerpts):

> Create `oneshot::channel<ReviewDecision>`.

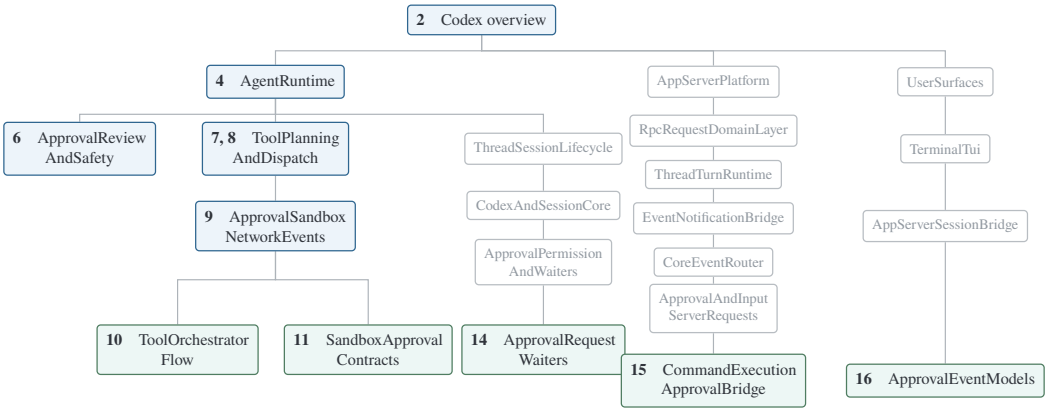


Figure 4: Graph and leaf-page reads in the command-approval case. Numbers match Table 4. Blue boxes mark graph reads; green boxes mark page reads. Gray paths connect ancestors whose child graphs were not read. Step 7 reads relations at the same node expanded in step 8.

> [...]
> Before sending the event, insert the sender into `active_turn.turn_state.pending_approvals`, to avoid a fast host response arriving before the waiter exists.
> [...]
> Wait for the receiver; if the sender is cleared or dropped, return `ReviewDecision::Abort`.

The model's answer uses the registration-before-emission detail and connects it to the return path:

> "It registers the Sender into the active turn's `TurnState.pending_approvals` [...] before sending anything, specifically to avoid a race where a fast host response arrives before the waiter exists."
> [...]
> "It then submits `Op::ExecApproval` [...] back into the core submission queue."
> [...]
> "[...] sends the `ReviewDecision` into the oneshot — this wakes the `.await` inside `request_command_approval` [...]."

The answer omits channel-cleanup aborts and RPC error-to-denial handling, although both appear in observations 14–15. The terminal's approval-ID/RPC-request-ID mapping is also missing; its dedicated `PendingRequestRegistry` page was not read.

Access patterns across the three questions. The local question goes directly from search to two pages; the LoRA question explores four top-level branches before reading eight pages (Table 5). The local run skips the overview: doc-drill's ordering is prompted, not enforced.

D Supplementary Results

D.1 AD-IR Size and Construction Cost

Table 6 summarizes each benchmark repository's AD-IR. Nodes count subgraph entries, edges count typed relations across all levels, and depth is the deepest subgraph below the root.

For SGLang, the largest repository, the usage ledger recorded 741 sessions: one scaffold, 739 refinement sessions, and one flow analysis. These consumed 130.8M tokens (2.26M output; the remainder input, including cache reads) over roughly 34 hours at a concurrency of eight. The ledger omits leaf-writing sessions, making this a lower bound on construction cost. The other five repositories were built before usage recording was added, so their construction costs are unavailable.

Table 5: Documentation reads by Claude Sonnet 5 for the three examples. Counts are browse-command invocations, including repeated searches; graph reads include the top level.

Question	Graph	Page	Flow	Edges	Search
Codex: patch matching	0	2	0	0	4
Codex: command approval	5	5	1	1	3
vLLM: dynamic LoRA loading	11	8	1	0	1

Table 6: AD-IR size per benchmark repository.

Repository	Subgraphs	Nodes	Edges	Flows	Leaf pages	Depth	Leaf text (MB)
KataGo	95	557	847	6	463	4	5.8
Codex	287	1,728	2,781	7	1,449	6	15.6
Ollama	105	616	882	6	512	6	5.3
qBittorrent	65	372	584	5	308	7	3.6
SGLang	739	4,006	3,866	6	3,269	8	47.5
vLLM	353	2,254	2,898	6	1,922	6	22.7

D.2 External Documentation Baselines

Table 7 gives per-configuration coverage for the three-way comparison in Section 4.4, using the shared access conditions in Appendix B. In the five-repository, 150-question comparison, ACCEED-flat and DeepWiki average 70.7% versus 38.5% with Flat access, and 70.9% versus 40.3% with RAG.

Table 7: Weighted coverage (%) on the three repositories shared by all documentation systems. Column maxima are bold.

Documentation	Access	Haiku 4.5	Sonnet 5	Luna	Sol	Terra	Mean
ACCEED-flat	Flat	54.6	**72.6**	**74.7**	**82.0**	73.2	71.4
ACCEED-flat	RAG	**61.1**	69.4	73.9	81.2	**74.3**	**72.0**
DeepWiki	Flat	27.8	33.0	43.4	54.2	44.0	40.5
DeepWiki	RAG	29.9	34.0	46.8	56.0	45.3	42.4
Google Code Wiki	Flat	15.9	19.3	35.8	49.9	26.8	29.5
Google Code Wiki	RAG	18.2	20.1	43.9	52.9	40.8	35.2

D.3 RQ1 Ablation and Cost

Hierarchy-only reaches 72.2% mean coverage, 2.3 points above Unstructured (paired 95% CI [1.3, 3.2]), and full AD-IR adds a further 4.7 points (95% CI [3.8, 5.6]). By question scope, Structured gains 3.3 points over Unstructured on local questions, 8.0 on cross-module questions, and 9.5 on repository-wide questions. Structured makes 20.1 tool calls per question on average against 11.9 for Unstructured, and spends $0.527 per question against $0.446.

D.4 RQ2 Coverage and Cost

Figure 5 plots the cost and coverage results of Table 2: Structured has higher coverage for all five answering configurations and lower cost for four. Structured and Source Retrieval make 20.1 and 20.3 tool calls per question on average. Table 8 gives coverage by question scope for Claude Haiku 4.5, GPT-5.6 Sol, and the mean of all five configurations.

D.5 RQ3 Maintenance Statistics

The paired bootstrap 95% CI of the mean gain over the 60 questions is [35.2, 49.7] points. Coverage rises from 24.0% to 63.7% on Codex and from 25.3% to 71.3% on vLLM. Over 150 pull requests per repository, maintenance ran 66 and 68 update tasks using 1.69M and 1.76M tokens, and modified 169 of 1,734 and 160 of 2,272 documentation files, in Codex and vLLM respectively.

Update tasks classified 49 and 51 changes as minor, 7 and 9 as structural, and 9 and 7 as requiring no edit for Codex and vLLM, respectively; a few changes lacked a parsable classification. Cumulative token usage was approximately linear in the number of processed pull requests.

Maintenance added 5 and 7 pages and removed 2 and 3 in Codex and vLLM, respectively. Every modified file lies within a node whose source scope intersects the changed files. Under the counting method in Appendix B.4, vLLM's textual change is $(1{,}726 + 1{,}787)/408{,}985 \approx 0.86\%$.

E SWE-QA Evaluation

Setup. We evaluate 336 questions from seven SWE-QA (Peng et al., 2026) repositories at the benchmark's pinned commits, using its questions and reference answers. We first selected the five repositories with the fewest tracked files (Requests, Flask, xarray, Reflex, and pytest), then added SymPy and Django to cover larger repositories. Each contributes 48 questions. AD-IR uses the same construction pipeline, with Claude Opus 5 in all roles.

Conditions and scoring. All three conditions use Claude Sonnet 5 for answering. **Direct** receives only the repository name and question. **Function-chunk RAG** retrieves up to ten function/class source chunks with BM25 and answers in a single turn. These two baselines are adapted from the benchmark's direct-prompting and function-chunking settings (Peng et al., 2026). **Structured** accesses complete AD-IR through doc-drill without source access. Using the benchmark's judge prompt, Claude Sonnet 4.5 scores correctness, completeness, relevance, clarity, and reasoning from 1 to 20 each, for a total out of 100.

Results. Structured averages 71.4 points, compared with 64.2 for Function-chunk RAG and 55.2 for Direct (Figure 6). SymPy and Django contain 38,795 and 41,249 source chunks, respectively; the mean gain over Function-chunk RAG is 10.7 points on these two repositories, versus 5.8 on

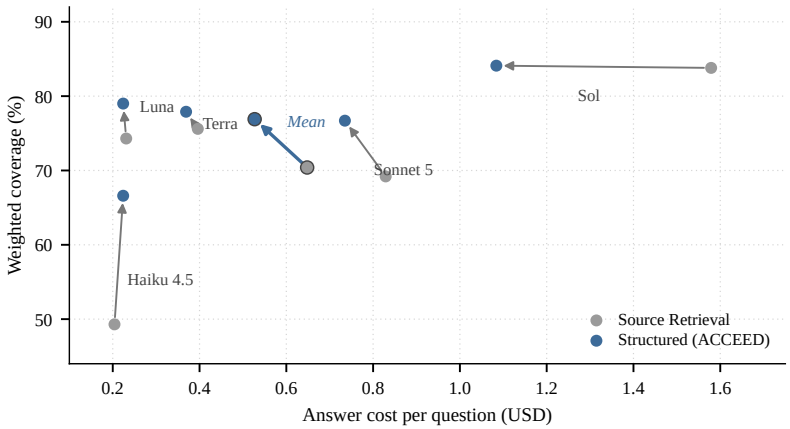


Figure 5: RQ2 answer-stage cost and weighted coverage. Arrows run from Source Retrieval to Structured for each answering configuration; the mean movement is highlighted.

Table 8: RQ2 by question scope: weighted coverage (%) under Source Retrieval (S) and Structured (A) context.

	Local		Cross-module		Repository-wide	
Answering configuration	S	A	S	A	S	A
Claude Haiku 4.5	56.2	**72.5**	45.5	**65.5**	46.2	**61.7**
GPT-5.6 Sol	**88.2**	84.5	83.0	**85.7**	80.2	**82.0**
Mean of five	77.0	**78.7**	69.2	**78.2**	65.1	**73.6**

the other five (755–8,885 chunks). The average advantage is larger on the two largest evaluated repositories, although individual gains do not increase monotonically with size.

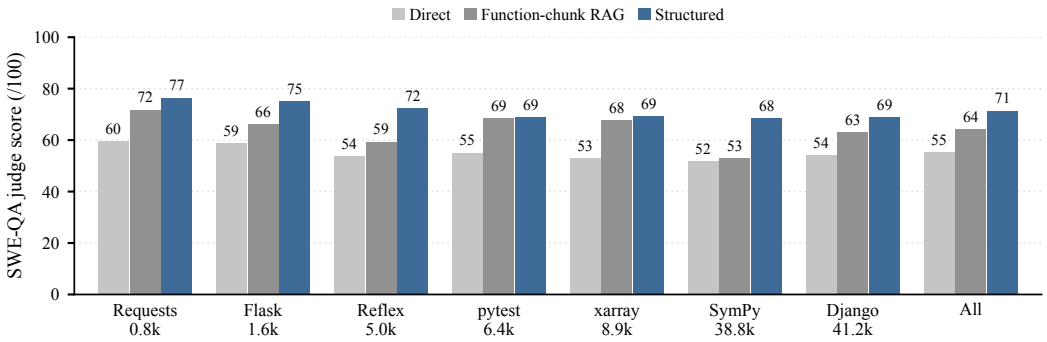


Figure 6: SWE-QA scores (out of 100), ordered by function/class source-chunk count (below each name). Each repository contributes 48 questions; All averages all 336. Bar labels are rounded to integers.