CodeNib: A Multi-View Data System for Serving Repository Context to Coding Agents

Zhongming Yu
UC San Diego
La Jolla, California, USA
zhy025@ucsd.edu

Hengjia Yu
UC San Diego
La Jolla, California, USA
hey015@ucsd.edu

Boqin Yuan
UC San Diego
La Jolla, California, USA
b4yuan@ucsd.edu

Shuting Zhao
UC San Diego
La Jolla, California, USA
shz106@ucsd.edu

Yizhao Chen
UC San Diego
La Jolla, California, USA
yic138@ucsd.edu

Aryan Dokania
UC San Diego
La Jolla, California, USA
adokania@ucsd.edu

Mihir Jagtap
UC San Diego
La Jolla, California, USA
mjagtap@ucsd.edu

Jiayu Chang
Stanford University
Stanford, California, USA
cjy1125@stanford.edu

Yitong Ma
UC San Diego
La Jolla, California, USA
yim030@ucsd.edu

Yash Jayswal
UC San Diego
La Jolla, California, USA
yjayswal@ucsd.edu

Wentao Ni
UC San Diego
La Jolla, California, USA
w2ni@ucsd.edu

Hejia Zhang
UC San Diego
La Jolla, California, USA
hez024@ucsd.edu

Zhaoling Chen
UC Riverside
Riverside, California, USA
zhaoling.chen@email.ucr.edu

Gangda Deng
University of Southern California
Los Angeles, California, USA
gangdade@usc.edu

Jishen Zhao
UC San Diego
La Jolla, California, USA
jzhao@ucsd.edu

ABSTRACT

Coding agents repeatedly search, navigate, and retain context from evolving repositories, but disconnected indexes, language servers, and task-local histories force repeated discovery and obscure lifecycle costs. CodeNib builds reusable lexical, dense, and structural views per repository commit, maps outputs to repository-relative source ranges, maintains selected views across edits, and serves ranked search, symbol navigation, and bounded context through one runtime.

Across 100 snapshots, we map quality–cost frontiers across the repository-context lifecycle. When outputs match an independent rebuild, graph and vector updates are 8.7× and 25.4× faster at the median. On the static-navigation subset matching normalized live-server locations (63% of 1,000 requests), the median per-request live/static latency ratio is 4.7×. Across five models, selected context policies preserve localization with 50–87% fewer trajectory tokens than paired grep/read. Together, these results support multi-view repository-context serving with explicit, operation-specific validity boundaries.

1 INTRODUCTION

Coding agents access evolving repositories through lexical and semantic search, symbol and dependency navigation, source reads, and bounded history [64, 70]. These operations share source state while yielding distinct evidence.

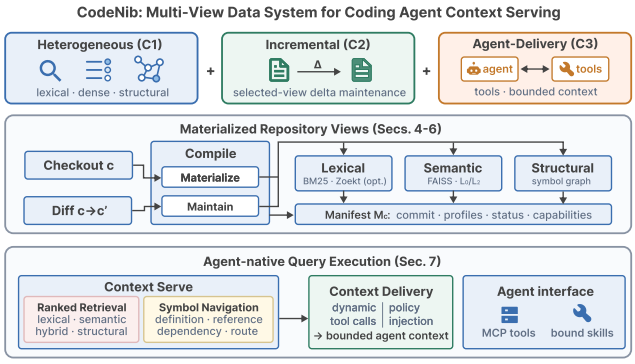


Figure 1: CodeNib compiles and maintains heterogeneous repository views, then serves them as bounded agent context.

Repository artifacts span text indexes, embedding stores, semantic graphs, language servers, and prompt state, each with distinct layouts, costs, update paths, and output contracts. Retrieval yields ranked candidates; navigation yields locations. Prior systems materialize code facts, expose live semantic tools, or construct ranked and structural repository context [5, 7, 39, 43, 48, 49]. Composing these capabilities creates a data-lifecycle coordination problem for the agent runtime.

We take a data-systems view: a commit is immutable base data; chunks, postings, embeddings, occurrences, and relationships are

Code and artifacts: https://github.com/sysevol-ai/CodeNib.

Table 1: System positioning by reused state and agent-facing result. Entries show primary emphasis, not exhaustive features.

System family	Reused state	Agent-facing result
Code stores / dataflows [9, 43, 46, 57]	Materialized facts or incremental transforms; system-specific updates	Search, navigation, developer or agent data
Retrieval graphs [8, 39, 49, 62]	Repository- or task-scoped retrieval structures	Ranked and structural context
Live analysis and agent tools [1, 24, 48]	Workspace state and live language/tool services	On-demand semantic context, file/search, edit, and command results
CodeNib	Repository manifest plus lexical, dense, and structural views; incremental maintenance and runtime view loading	Repository retrieval, static navigation, bounded context serving

derived views; agent requests are view-specific queries; and prompt context is a bounded delivery result. The key design is to reuse repository-derived state across tasks without collapsing ranked candidates, source locations, and prompt history into one abstraction. Figure 1 summarizes three coupled challenges. **C1: Heterogeneous views.** Lexical, dense, and structural data need different physical layouts, but their results must map back to the same commit and repository-relative source ranges. **C2: Incremental freshness.** An edit changes each view differently, so graph repair, embedding reuse, and rebuild paths must be selected and evaluated separately. **C3: Agent delivery.** Precomputed evidence must reach the model through tools or bounded context while build, runtime loading, query, and history costs remain visible. Otherwise, each issue repeats model-directed grep/read discovery and carries its task-specific observations through later turns.

We present CodeNib, a multi-view data system spanning this lifecycle. The mechanism is simple yet effective: build several repository views once, map every result back to source ranges, update each view through its own path, and load the views needed by each agent operation. Its repository view compiler normalizes files, scopes, and callables; independent view builders materialize lexical, dense, and structural artifacts, while view-specific maintainers use Git/LSP-assisted graph repair and content-addressed vector reuse. A manifest records each artifact's path, status, commit, configuration, and supported operations; repository-relative addresses align their outputs. Independent rebuilds are used only after timing to determine which update outputs match; this comparison does not execute on the maintainer path.

The runtime loads required views, lowers ranked requests to physical routes, and composes source-linked ranked code blocks. It exposes static or live symbol providers through one location interface. Bound skills and a stdio MCP adapter [2] connect these operations to the agent loop, where policies govern grep/read, eager, or eager-plus-compact context delivery. Traces retain the view, provider, and token usage across construction, maintenance, queries, and delivery. Table 1 positions this boundary.

This paper makes three contributions corresponding to C1–C3.

(1) **A repository view compiler (C1).** CodeNib builds lexical, dense, and structural artifacts independently, records which are available for a commit, and maps their outputs to repository-relative source ranges.
(2) **View-specific incremental maintenance (C2).** Git/LSP-assisted graph repair and content-addressed vector reuse update affected state. We compare timed updates with independent rebuilds offline and report update speedups only for matching outputs.
(3) **A cost-visible agent runtime and extensive Pareto evaluation (C3).** Ranked plans, static/live navigation providers, and bounded context policies connect reusable views to agent tools. Across the lifecycle, our stage-separated Pareto and quality–cost analyses cover retrieval and reranking, dense-index construction and search, static/live navigation, incremental maintenance, and bounded context delivery. They distinguish quality, compatibility, update fidelity, latency, and token usage rather than collapsing them into one score.

We evaluate repository localization and context serving; patch generation and concurrent production updates remain outside this study. The measurements expose three system-level gains. Static navigation reproduces the live server's normalized path/start-line set on 632 of 1,000 requests and has a 4.72× median per-request live/static latency ratio on that subset. Graph and vector updates match independent rebuilds on 15/33 and 28/31 source-changing transitions, with median 8.67× and 25.44× speedups on those cases. For each of five agent models, the lowest-token core arm that meets a common localization margin uses 50–87% fewer trajectory tokens than paired grep/read. Together, CodeNib turns repository context from repeated per-task exploration into reusable views that can be built, updated, and served independently. Shared commit and source addresses connect their outputs, while concrete per-operation measurements distinguish exact substitutions from quality–cost tradeoffs.

2 BACKGROUND AND POSITIONING

2.1 Data-Management Foundations

Materialized views trade construction and maintenance for query-time reuse [23]; repository indexes make this trade against a changing commit. Because text, vector, and graph operators use different records and return different values, CodeNib catalogs specialized indexes behind a manifest. This resembles polystore mediation [13], but uses curated physical routes rather than searching a costed plan space [20]. General agent-data engines such as CocoIndex incrementally maintain declared source-to-target flows with lineage [9]. CodeNib instead fixes the repository commit and build configuration for each view, then measures its build, update, load, and query costs separately.

2.2 Materialized Code Intelligence

LSP standardizes live client–server requests, whereas LSIF and SCIP define serialized code-intelligence index formats for locations and relationships [44, 45, 55]. Glean stores code facts for developer tools; Sourcegraph combines search and code intelligence, and Zoekt provides trigram search [43, 46, 57]. CodeNib links these ideas with dense and sparse retrieval in a local runtime; it claims neither Glean-scale storage nor a general fact language. Glean incrementally propagates fact ownership around reindexed units [42]. Industrial call-graph maintenance deletes invalid nodes and edges before

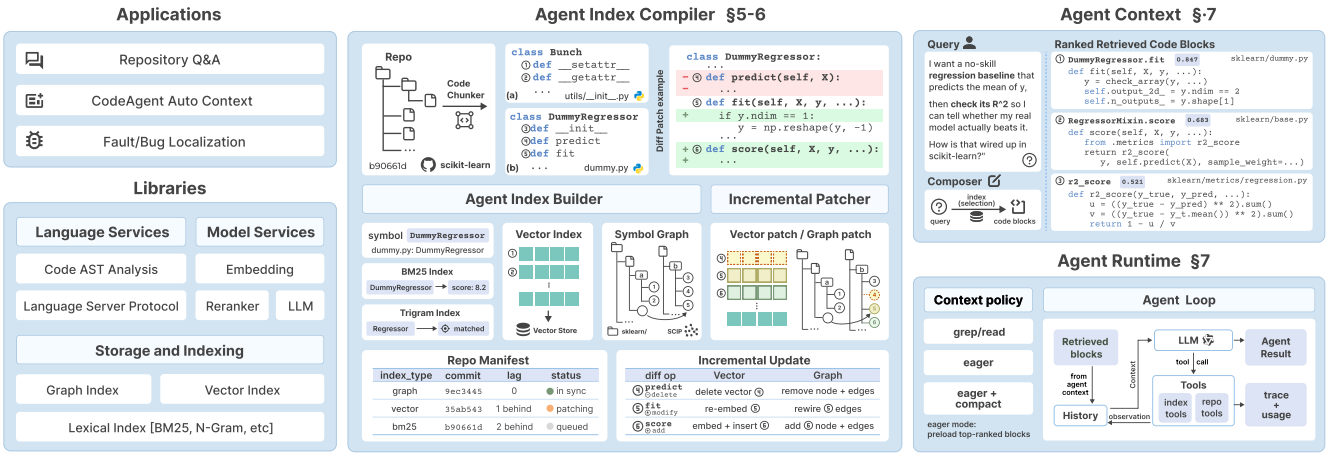


Figure 2: CodeNib's repository-to-agent dataflow. The left column lists supported applications and dependencies, not a pipeline stage. The center Repository View Compiler's *View Builders* and *View Maintainers* create and update manifest-linked views (Secs. 5–6). The upper-right Query Planning and Retrieval path composes source-linked ranked retrieved code blocks (Sec. 7.2); the lower-right Agent Runtime applies context policies and exposes index tools around the agent loop (Secs. 7.1, 7.3, and 7.4). Hashes, lag, and scores are illustrative, not measurements.

patching affected code, Stack Graphs constructs file-incremental name-resolution graphs, and incremental CodeQL reuses production analysis state [12, 59, 77]. CodeNib instead repairs source-anchored graph facts; Q4 compares them with independent rebuilds offline to qualify reported speedups. That comparison is not on the maintainer path. Tree-sitter nodes retain source start/end positions when semantic coverage is absent, and the manifest records whether semantic navigation is available [61].

LSPRAG retrieves task-specific definitions and references from live LSP back ends for real-time unit-test generation [24], while TypeScript indexing can replace per-symbol LSP calls with compiler-API analysis [50]. We compare static and live normalized locations across five language groups and retain a live provider because their outputs can differ.

2.3 Repository Retrieval and Structural Context

Lexical retrieval matches identifiers; code encoders map text and source into a shared space [16, 21, 22, 75]. Rerankers rescore candidates, while FAISS supplies exact, IVF, and HNSW organizations [28, 41, 58, 75]. We evaluate these physical choices rather than introduce a retrieval model or ANN algorithm.

Repository systems also differ in when and how they retrieve. RepoCoder alternates retrieval and generation, RepoFormer predicts when retrieval is useful, and CoRet trains a dense retriever for code-editing requests using repository structure and call-graph dependencies [14, 66, 73]. Direct Corpus Interaction instead exposes raw grep/read operations without a prebuilt index [34]; a cross-paradigm study separately compares similarity, static-analysis, and navigation context engineering [33]. These are complementary serving choices, not interchangeable implementations of one retrieval operator.

Repository-level localization ranges from staged, non-agentic narrowing in Agentless, through iterative repository exploration in AutoCodeRover, to two-phase graph search in CoSIL [26, 68, 76]. Graph-guided agents instead make structural navigation part of the action loop [7, 39, 49, 72]. RIG and Codebase-Memory serve deterministic structural context [8, 62]; AOCI proposes symbolic–semantic indexing, while Code Isn't Memory evaluates a structural codebase index inside a controlled coding-agent harness [5, 37]. SpIDER augments dense retrieval with graph exploration and LLM reasoning, while AIRCoder fuses textual, dependency, and structural-hierarchy metrics [6, 56]. CodeNib instead links ranked and navigation views to the same repository commit while keeping ranked code blocks distinct from source-location results; its graph ablation fixes the dense retriever and measures paired one-hop effects.

2.4 Agent Interfaces and Context

Agent interfaces affect behavior [1, 64, 70]. Devin combines an autonomous agent with shell, editor, and browser tools, while DeepWiki exposes generated documentation and search over indexed repositories [60, 67]. Serena exposes LSP-backed semantic retrieval and editing tools, Aider supplies a repository map, and Context as a Tool performs explicit history compression [17, 38, 48]; RepoShapley learns context filtering for repository-level completion [25], while SWE-Explore evaluates exploration under a line budget [74]. SpecAgent predicts completion context during indexing rather than serving reusable query views [40]. ContextBench, AGENTS.md studies, and deterministic anchoring measure context use or stability [19, 32, 35]. CodeStruct and CodeMEM add AST-scoped actions or session memory, while other work varies edit-time representation [30, 54, 63]. AgentDiet removes redundant and expired information from agent trajectories, the Complexity Trap compares raw history, observation masking, and summarization, and SWE-Pruner performs task-conditioned pruning of long agent contexts [36, 65, 69]. Repository-context compression instead shortens a static repository input before generation, not observations retained across an interactive loop [15].

We instead hold candidates fixed while varying initial delivery and one-time compaction during localization. Static/live latency is reported only where normalized path/start-line sets match, and

token savings only under a paired localization margin; neither claim extends to learned exploration or patch correctness.

3 SYSTEM OVERVIEW

The dataflow in Figure 2 begins in the center panel; the left column lists supported applications and dependencies rather than another pipeline stage. The center *Repository View Compiler* is the commit-indexed data plane (Sec. 3.1). The upper-right *Query Planning and Retrieval* path converts a query into ranked retrieved code blocks (Sec. 3.2), while the lower-right *Agent Runtime* loads views, exposes tools, and delivers context around the agent loop (Sec. 3.3). The planes meet at manifest M_c and repository-relative source addresses. We follow this center-to-right flow below.

3.1 Repository View Compiler

The center panel has two paths. Under *View Builders*, the Code Chunker and semantic backends derive source-linked units and write graph, vector, BM25, and optional Zoekt artifacts. Under *View Maintainers*, a Git diff drives LSP-assisted graph repair or content-addressed vector reuse. The bottom-center *Repo Manifest* is M_c: it catalogs each view's commit, profile, status, and capabilities. It is the runtime's lookup boundary, not a container for view payloads. Section 4.1 formalizes M_c, Section 5 defines the views, and Section 6 details the builder and maintainer paths.

The initial-build path runs requested builders sequentially and publishes the manifest only after every builder reports success or failure. A failed optional view does not invalidate successful siblings. Initial construction and delta maintenance have different publication boundaries: builders link a complete requested artifact set through M_c, whereas graph and vector maintainers update their own stores. This distinction lets the evaluation change one physical path without silently changing the others.

3.2 Query Planning and Retrieval

The request plane exposes two request classes, placed in different regions of Figure 2.

Ranked retrieval. In the upper-right panel, a query enters the *Composer*, which selects a dense, lexical, structural, or fused physical plan and returns ordered, source-linked code blocks. Section 7.2 defines this lowering.

Symbol navigation. A definition or reference request selects a static occurrence/graph provider or live JSON-RPC and returns normalized locations rather than ranked code blocks. The figure represents these providers as the lower-right loop's *index tools*; Section 7.3 specifies their shared location interface.

Provider choice remains trace-visible: a normalized location does not erase whether it came from persisted occurrences, graph traversal, or a live server.

3.3 Agent Runtime

The lower-right panel begins with a *Context policy* and ends with an *Agent Result* and trace. During session setup, the runtime reads M_c and opens only the required prebuilt views as process-local query state (Sec. 7.1); it does not build views or insert code into history. During task execution, the agent may call *index tools* or *repo tools* dynamically, while the policy may instead preload ranked L_2 code blocks and compact retained observations (Sec. 7.4). Skills and the stdio MCP adapter expose the same search, dependency, definition, reference, and route operations.

Runtime view loading is capability-driven. MCP loads configured resources once per process, while the agent path loads only indexes required by selected skills.

3.4 Boundaries

CodeNib does not choose edits or judge tests. Its manifest is not a cross-store transaction, and compilation records a commit without locking the worktree. Because static navigation does not reproduce every live response's normalized locations, callers requiring workspace semantics must retain a live LSP path. Q4 compares updated views with independent rebuilds only after timing; these comparisons do not run on the maintainer path.

These limits follow directly from the implementation. A recorded commit does not prove that a mutable checkout stayed quiescent during construction; a manifest does not make separate stores transactional; and source-address normalization does not make two providers behaviorally equivalent. Each experiment therefore states the concrete output it compares.

From dataflow to measurements. The remainder follows Figure 2's dependency chain. Section 4 first defines the shared commit and source-address model plus the result and metric for each operation. Section 5 instantiates the view state cataloged by M_c, and Section 6 gives the initial and delta paths that create or advance that state. Section 7 consumes the same views through ranked plans, symbol providers, and context policies, while Section 8 maps the abstractions to implementation packages and adapters. Finally, Section 9 preserves this decomposition: Q1–Q2 measure retrieval and dense indexes, Q3 measures symbol navigation, Q4 and the lifecycle trace measure maintenance and cross-stage composition, and Q5 measures context delivery.

4 REPOSITORY VIEWS AND REQUEST SEMANTICS

We distinguish four operations by their returned value and measured cost: ranked retrieval, symbol navigation, structural maintenance, and context delivery. Their output comparisons define the evaluation; they are not one shared runtime gate.

4.1 Views and Source Addresses

Let c identify a repository commit and let U_c be the source units extracted from its checkout. A source unit records

$$u = \langle p, r_s, r_e, \ell, \tau, x, s \rangle,$$

where p is a repository-relative path, $[r_s, r_e]$ is a source range, $\ell \in \{L_0, L_1, L_2\}$ is its granularity, τ is its node type, x is source text, and s is an optional resolved symbol. Files are L_0, type-like scopes are L_1, and callable definitions are L_2.

CodeNib materializes three views over U_c:

Lexical view V_c^{lex}. Posting or trigram records for identifiers, paths, comments, and source text.

Dense view V_c^{dense}. Embeddings of L_0 or L_2 units with a mapping back to their source ranges.

Structural view G_c. Typed containment and relationship edges among source-linked files, scopes, and definitions, plus persisted occurrence records where a backend supplies them.

Each view has profile θ (language, backend, schema, model, and options). Manifest $M_c = \langle c, V_c^{\text{lex}}, V_c^{\text{dense}}, G_c, K_c \rangle$ links profiles, artifact status, and capabilities K_c without implying one storage engine. Results expose address $\langle p, r_s, r_e, \tau \rangle$; range containment aligns granularities, while backend identifiers remain view-local. During session setup, the runtime uses M_c to locate required artifacts and check their recorded status and capabilities.

4.2 Four Operation Classes

Ranked retrieval. A request maps text query q to ranked source units and lowers to

$$z = \langle r, k, \rho, h \rangle.$$

Route $r \in \{A, B, C, D\}$ selects lexical, semantic, hybrid, or structural retrieval and route-local fusion; only hybrid C owns RRF. Width k contains retrieval fan-out k_{ret} and optional pre-rerank cut k'; final k_{out} remains a caller limit. ρ selects an optional reranker, and graph expansion h is legal only on D. The tuple records varying decisions, not a normal form; Q1's explicit dense–graph fusion is an ablation outside this automatic lowering space.

The executed result is a ranked list $[(u_1, \sigma_1), \ldots, (u_{k_{\text{out}}}, \sigma_{k_{\text{out}}})]$. Operationally, k_{ret} maps to `retrieve_top_k`, while k' maps to `rerank_candidate_top_k`; the latter is subordinate to the width policy rather than a fifth plan coordinate. This separation matters because increasing retrieval fan-out and increasing reranker work have different latency and recoverability effects. Named experimental pipelines may compose operators outside the automatic A–D routes, but their operators and fusion rules remain explicit.

Symbol navigation. An LSP-shaped request a specifies a capability and source position. Let $N(\cdot)$ project provider-limited results to deterministic unique path/start-line pairs. Static and live providers match when

$$N(P_{\text{static}}(a, M_c)) = N(P_{\text{live}}(a, c)).$$

This normalized location set omits characters, end ranges, and provider metadata. Equality defines the subset used for the conditional latency comparison in Figure 9, but does not prove interchangeability; match rate and matched-request latency must be reported together.

The normalized set is intentionally weaker than full location or response equality. A match says nothing about characters, end ranges, hover text, workspace diagnostics, or provider metadata, and a latency reduction on the matched subset says nothing about mismatches. The evaluation therefore reports coverage and conditional latency as separate quantities rather than treating the static provider as a universal LSP replacement.

Structural-view maintenance. Let D be the files changed by a transition $c \to c'$, let $\widehat{G}_{c'}$ be an incrementally repaired graph, and let $G_{c'}$ be a fresh target graph built independently for offline evaluation. Define $\mathcal{F}(G)$ as the declared graph-fact projection: the tagged multiset of vertex identities, types, source/selection lines, and typed edges with source anchors in G. It is not byte-level serialization equality. The offline graph-output equality check is

$$\mathcal{F}(\widehat{G}_{c'}) = \mathcal{F}(G_{c'}). \tag{1}$$

For deterministic definition/reference requests $\mathcal{A}_D$ anchored in D, let $R_G(a) = N(P_{\text{static}}(a; G))$. After persistence and reload, we apply a second offline serving check:

$$\forall a \in \mathcal{A}_D : \; R_{\widehat{G}_{c'}}(a) = R_{G_{c'}}(a). \tag{2}$$

Both checks run after timed maintenance against the independently rebuilt target. The fresh rebuild and comparisons do not execute on the current maintainer path and are not included in update latency. They qualify the reported speedup; they do not provide an online correctness oracle. Neither check covers live-LSP behavior or atomic cross-view publication.

The first check compares the declared graph-fact multiset in memory. The replay check crosses persistence and serving boundaries by reloading both graphs and comparing deterministic requests. It is a regression suite over the same transition, not additional statistical evidence. A transition that fails either check contributes no conditional speedup, even when its patch latency is low.

Context delivery. A context policy π maps ranked candidates and accumulated history to model prompts. We compare model-directed grep/read, eager L_2 injection, and the same injection followed by a one-time history rewrite. Q5 reports trajectory tokens and AnswerRecall@5 over the first five deduplicated source spans committed in the final answer. The @5 cutoff is a reporting choice, not an output cap; Figures 6 and 7 retain their separately frozen top-10 retrieval/index contracts. Eager and compact receive identical candidates, so only their direct contrast isolates retention; comparisons with grep/read also change candidate delivery. Retained content is charged each time it appears in a later prompt, independently of prefix-cache reuse.

4.3 Metrics and Cost Accounting

Each operation uses its declared output contract: target coverage for retrieval and final answers, all-target File Success, exact-Flat overlap for ANN, and output-match predicates for navigation and maintenance before conditional latency or speedup. Appendix B defines the denominators, deduplication and cutoff rules, and gives worked examples.

Maintenance speedup. Graph arm a is file replacement or symbol repair. Let $n_{\text{share}} \geq 1$ count scheduled transitions sharing one server setup (1 for an isolated update and 5 here). The terms T_f^G, $T_u^{G,a}$, and $T_s^{G,a}$ are graph rebuild, update, and server startup/warmup costs; T_f^V and T_u^V are vector rebuild and update costs after shared model loading:

$$\Gamma_{G,a} = \frac{T_f^G}{T_u^{G,a} + T_s^{G,a}/n_{\text{share}}}, \qquad \Gamma_V = \frac{T_f^V}{T_u^V}. \tag{3}$$

A ratio above one means the update plus its setup share is faster than rebuilding. Rebuild and post-timing comparison remain outside the update path; conditional summaries require source-changing output matches, while raw ratios remain visible.

Lifecycle projection. Unlike Q2's post-extraction timing, lifecycle accounting starts from a prepared checkout. With materialization B, fresh-process view loading L, and an N-session service trace S, the per-session costs at reuse scale q are $B/q + L + S/N$ for independent loading and $(B+L)/q + S/N$ for one resident runtime. The measured serve-only term S/N excludes concurrency and queueing and is not an end-to-end or hardware bound. Plan latency includes invoked operators; LSP replay measures marginal warm latency.

Agent token usage sums provider-reported prompt and completion tokens over all model invocations, including answer-format invocations; tool observations count when serialized into a submitted prompt. Appendix B gives the exact sum and a retained-history example.

For every model, let $\Delta\text{AR@5}(\pi)$ denote the paired mean AnswerRecall@5 change from grep/read. Policy π preserves quality under our reporting rule when the lower bound of its paired 95% interval satisfies the operational margin $\epsilon = 0.05$:

$$\text{LB}_{.95}[\Delta\text{AR@5}(\pi)] \geq -\epsilon.$$

Among policies that meet the margin, fewer tokens are better. This is a reporting threshold, not a pre-registered equivalence test.

5 MATERIALIZED REPOSITORY VIEWS

One checkout is represented by separate structural, dense, and lexical artifacts. Their shared fields are a repository-relative source range and the repository commit recorded in the manifest, not a shared storage engine or a globally meaningful backend identifier.

5.1 Source Units

Language adapters instantiate the $L_0/L_1/L_2$ hierarchy using tree-sitter ranges; a language may omit L_1. The hierarchy supports file- or callable-level indexing, enclosing-file projection, and stable boundaries across backend symbol IDs. The registry chunks 14 languages and records graph and incremental support separately. Appendix C gives the exact five-language source unit and repository-filter rules used by the experiments.

Adapters preserve zero-based parser coordinates internally and may omit L_1 when a language has no applicable named scope. The hierarchy lets retrieval switch between file and callable units, lets evaluation project a callable back to its enclosing file, and supplies source boundaries even when a semantic backend uses a different symbol identity.

5.2 Structural View

Graph $G_c = (V_c, E_c)$ stores directories, files, and typed symbols linked by `contain`; semantic backends add `reference`, `import`, and type-use edges. Nodes retain path, range, language, name, and an optional backend symbol ID.

An `igraph` query layer indexes ranges and edge anchors, supports bounded neighborhood/dependency queries, and maps positions to enclosing symbols when no exact occurrence is persisted.

Versioned graph pickles reject mismatched schemas rather than migrate silently.

Symbol subtypes include classes, functions, methods, and fields. The query layer indexes both source ranges and edge anchors, then filters incident adjacency by containment or semantic family; bounded expansion therefore does not scan every edge. Position-to-enclosing-symbol lookup is also the fallback used by static navigation when a backend did not persist an exact occurrence. Schema rejection makes a stale graph fail with a rebuild requirement instead of silently interpreting it under a newer node or edge schema.

5.3 Dense and Lexical Views

The dense builder embeds L_0 files or L_2 callables into FAISS and retains a source/text side mapping. Flat inner-product search is the default, IVF is configurable, and Q2 rebuilds fixed vectors into HNSW only for ablation.

BM25 ranks chunk text [53]; optional Zoekt indexes raw-file trigrams for substring and regex search. MCP normalizes both to paths, ranges, snippets, and optional scores.

The dense side mapping retains the exact source address and text associated with every vector. Q2's HNSW path rebuilds the same frozen vectors solely to compare physical organizations; HNSW is not silently substituted into the runtime configuration. BM25 and Zoekt also answer different requests: BM25 ranks source units, whereas Zoekt reports file-level text matches before MCP normalization.

5.4 Manifest Linking

Each builder writes independently and returns status and metadata. After all builders finish, M_c records type, path, timestamp, status, configuration, and duration, then derives capabilities. Thus BM25 can remain available after a vector failure. The manifest binds requested artifacts to commit c but does not provide cross-store transactions after edits.

This linking protocol provides failure isolation and discovery. Clients inspect capabilities instead of inferring availability from files on disk, and a query process can continue serving one successful view after another builder fails. The entry records what was requested for commit c; it does not certify later worktree state or synchronize independent post-build mutations.

6 VIEW CONSTRUCTION AND FRESHNESS

The materialization pipeline converts a checkout into the views of Sec. 5. Its primary correctness responsibility is provenance: every artifact and exposed capability must identify the repository state, profile, and backend that produced it. Construction and incremental repair deliberately retain different freshness and publication boundaries.

6.1 Language Backend Selection

A registry independently selects chunking, cold graph, incremental, and live language-server backends. Mature routes use SCIP; C/C++ uses clangd artifacts; other languages retain tree-sitter chunks and text retrieval. Capabilities thus distinguish syntactic hierarchy from resolved cross-file occurrences.

SCIP occurrences associate source ranges with optional symbol identifiers and role bitsets [55]; the decoder creates graph nodes/edges and can persist character-accurate lookups. Without exact occurrences, static serving falls back to the coarser graph and records that result granularity.

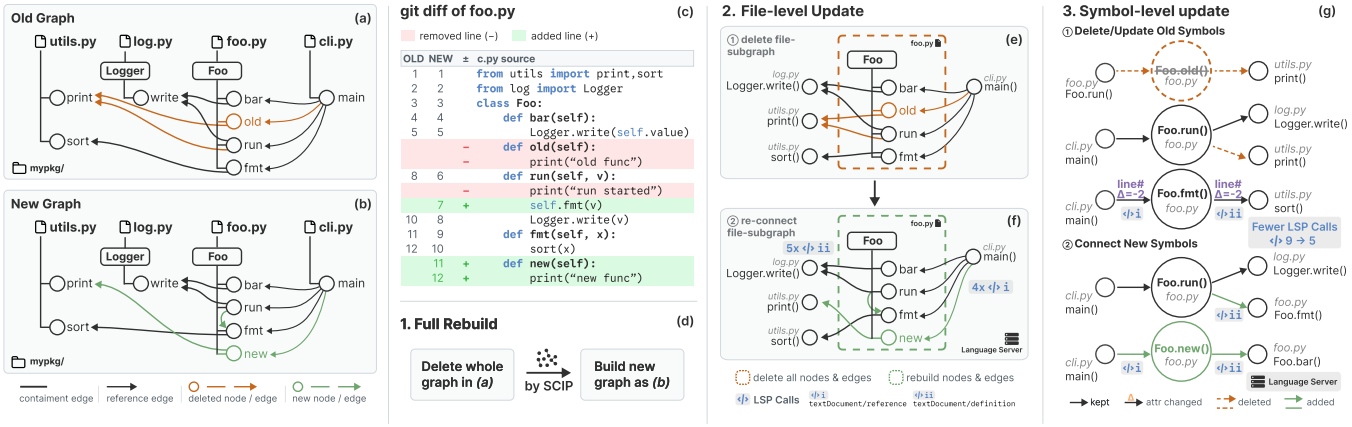


Figure 3: LSP-assisted incremental graph maintenance. (a–b) Old and target graphs for diff (c). Full rebuild (d) recreates the target; file-level update (e–f) deletes and reconnects the entire changed-file subgraph. Symbol-level update (g) preserves stable graph facts and repairs edit-invalidated state. File-level uses four reference plus five definition requests; symbol-level uses one plus four (9 → 5, four or 44.4% fewer). Counts cover displayed LSP requests, not all messages or aggregate speedup.

Compiled-language preparation may require dependencies, a compilation database, or a project build. Graph builders therefore fail explicitly rather than silently substitute weaker data under the same capability.

SCIP definitions become source-linked graph vertices and resolved occurrences become typed relationships. Where available, the decoder also persists an occurrence table for character-accurate definition and reference lookup; a graph-only fallback advertises its coarser behavior. This distinction is especially important for compiled projects, where dependency restoration, compilation-database generation, or a successful build is part of semantic coverage. An optional builder failure is recorded instead of being relabeled as equivalent semantic output. Appendix D records the evaluated Tree-sitter, cold-index, live-server, and static-position routes.

6.2 Initial-Materialization Pipeline

The store derives separate repository/commit and artifact-profile IDs, verifies the detached worktree, nests profiles below the source identity, and aliases benchmark IDs. Requested builders then run sequentially and write M_c. BM25 and vector currently extract units independently, so U_c is logical rather than a shared materialization; we claim neither extraction sharing nor parallel-build speedup.

Q2 build timers start after source extraction and include embedding plus FAISS, but exclude checkout, dependencies, SCIP, and graph decoding; they isolate granularity and vector-index choices rather than repository cold start.

`SourceSnapshot` hashes the canonical repository/commit pair separately from `ArtifactProfile`, which hashes language, schema, backend, and builder options. The store verifies the detached worktree commit, nests profiles under the commit identity, and treats benchmark instance IDs only as aliases. These identities prevent measurements from conflating repeated instances with unique repository/commit pairs. They do not remove the need for a quiescent checkout during a build.

6.3 Static Symbol Serving

Static and live providers receive the same capability, path, position, and options. Static serving prefers occurrences, otherwise resolves an enclosing graph node and follows source-linked records. Both emit normalized locations and provider, repository commit, and granularity metadata.

Figure 9 measures static/live normalized-location matches. Because the profile fails an all-request equality criterion, the interface retains live JSON-RPC and does not route unseen requests to static serving automatically.

The provider boundary converts coordinates once, applies identical result limits, and sorts normalized locations before comparison. Persisted occurrences serve position-based requests directly; the fallback maps a position to a graph node and follows definition or reference records. Trace metadata makes the path visible, but cannot predict whether an unseen static response will match a live workspace server.

6.4 Delta Maintenance and Freshness

Figure 3 contrasts full rebuild with file- and symbol-level repair. A Git hunk need not invalidate every declaration: the file-level baseline discards stable symbols and cross-file relationships and treats a line shift as a semantic edit. The symbol-level path instead combines zero-context hunks with one new `documentSymbol` tree to classify symbols as deleted, affected, shifted, unchanged, or added. In panel (g), *kept* `bar` and unchanged `run` edges are copied; *attribute-changed* `fmt` facts keep identity while line/anchor attributes rebase by −2; `old` and `run`→`print` are *deleted*; and `new`, `run`→`fmt`, and incident facts are *added* after all vertices exist. Thus one affected symbol can mix actions.

The file-level path reconnects four symbols and five outgoing relationships with four `references` and five `definition` requests. Symbol-level reuses preserved facts: one `references` request discovers incoming edges for `new`, while four `definition` requests

validate the two rebased `fmt` anchors and the new outgoing relationships from `run` and `new`. The example therefore saves four requests; the count excludes synchronization, `documentSymbol`, and other protocol messages. Q4 reports this saving only for transitions matching an independently rebuilt target in offline evaluation.

The classifier also retains an old backend-invisible symbol when its declaration line is unchanged. A file-wide line map treats each edge's `anchor_file` as location authority, preserving anchors on unchanged lines and removing anchors on edited lines before repair. Creating all new vertices before edge repair ensures that cross-file targets already exist when relationships are reconnected.

The patcher synchronizes changed text, re-resolves relocated call anchors with position-based definitions, discovers incoming edges with references, and resolves outgoing edges from changed-range semantic tokens; Appendix F gives protocol details. The vector updater reuses content-addressed embeddings and mutates or rebuilds FAISS according to the declared delta threshold; BM25 still rebuilds.

Live LSP work is confined to this maintenance path: changed text is synchronized before queries, incoming method references are validated through source-position definitions, and outgoing relationships are reconstructed from semantic tokens in changed ranges. Q4 evaluates the vector updater under a separate artifact-and-replay protocol; the current BM25 path has no integrated delta update.

Delta paths do not transactionally advance all views. Q4 therefore conditions each reported speedup on an offline comparison with an independently rebuilt target. That comparison is not on the maintenance path; burst throughput and cross-view staleness remain outside scope.

7 QUERY AND CONTEXT RUNTIME

7.1 Agent Loop and View Loading

Listing 1 shows the two phases. Compile isolates view construction and publishes artifact paths, status, and capabilities in M. Run-Agent derives `needed` from the selected skill metadata. `load_views(M, needed)` performs runtime view loading: it resolves those views in M, validates their entries, and opens or deserializes their prebuilt artifacts as runtime contexts before binding skills. It neither builds an index nor inserts retrieved code into model history. A missing required view aborts setup; the runner then filters unavailable skills and surfaces staleness warnings before exposing tool schemas. This runtime preflight checks manifest state, not fresh-target equivalence. Thus no builder is reachable on the measured request path.

The loop follows ReAct's standard reason–action–observation pattern [71]. CodeNib contributes the view-backed tool surface and the evaluated delivery policies, not the loop topology.

Figure 4 expands the listing's online half: manifest-driven runtime view loading and preflight construct the session tool set, while the issue and policy construct H_0. Each turn follows $H_t \rightarrow$ LLM $\rightarrow$ Tool Call $\rightarrow$ Dispatch $\rightarrow$ Observation $\rightarrow H_{t+1}$; a terminal answer instead produces the agent result. The trace ledger records tool and provider provenance alongside usage. The policy changes initial delivery and, for Compact only, one later history state.

```python
def COMPILE(checkout, requested):
    M = RepoManifest(commit=checkout.commit)
    for kind in requested:
        try:
            artifact, config = BUILDERS[kind].build(
                checkout)
            M.indexes[kind] = IndexEntry(
                path=artifact, status="fresh",
                config=config)
        except Exception as error:
            M.indexes[kind] = IndexEntry(
                status="failed", error=error)
    M.derive_capabilities()
    M.save("repo_manifest.json")
    return M

def RUN_AGENT(issue, M, skill_ids):
    specs = load_skill_metadata(skill_ids)
    needed = requirements(specs)
    # Open existing views; never build online.
    contexts = load_views(M, needed)
    registry = load_skills(specs, contexts)
    runner = AgentRunner(
        registry=registry,
        manifest=M,  # Runtime preflight
    )
    return runner.run(issue)
```

Listing 1: Manifest-mediated offline build and online agent setup. `load_views` opens existing manifest-linked views as runtime contexts; it neither builds views online nor adds retrieved code to model history.

7.2 Ranked Query Plans

Plans may be explicit or selected by a deterministic query/budget heuristic. Q1 invokes them explicitly and does not evaluate selector quality. After route execution, the composer projects scored source units to code blocks; the agent path's top-ten L_2 code blocks form C_{10}^{ctx} in Sec. 7.4.

The implemented selector extracts lexical, semantic, and structural signals, combines them with a latency/quality budget and manifest capabilities, and lowers them to Figure 5's tuple. Because Q1 invokes plans directly, the paper evaluates physical operators and not this heuristic's route accuracy.

Dense search. $\mathrm{S}_{\text{dense}}(q, k_{\text{ret}})$ embeds q and returns FAISS top-k_{ret} L_0 or L_2 units. It is Q1's common starting point.

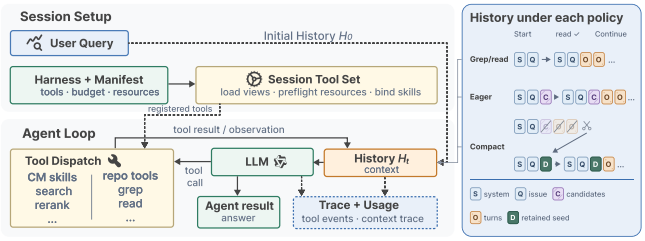
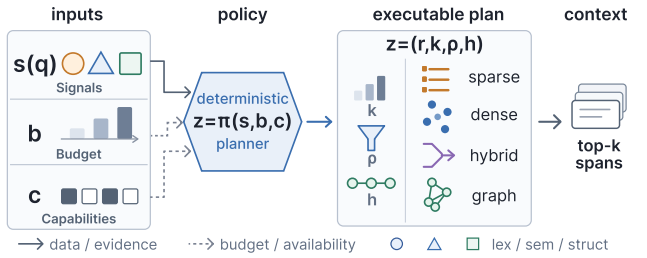
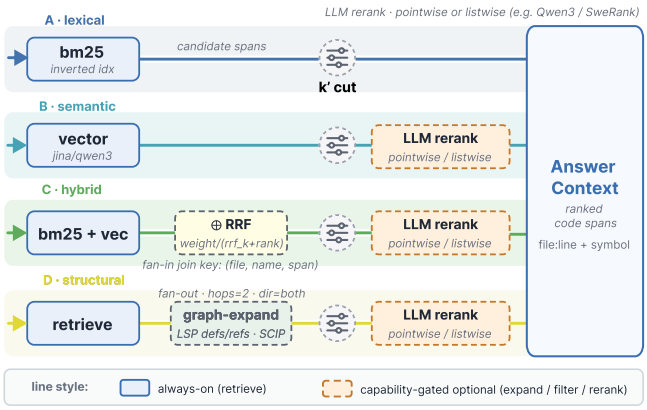


Figure 4: Agent loop, view loading, and evaluated context policies. Manifest preflight loads views and binds session tools. S/Q/C/O/D denote system prompt, issue, candidates, observations, and retained seed. Arms share issue, tools, and budget: Grep/read starts with [S,Q], Eager and Compact with [S,Q,C]; after the first successful read, Compact alone rewrites once to [S,Q,D] and then appends. Q5 fixes $k = 10$ candidates.



(a) Query-to-plan lowering



(b) Physical retrieval routes

Figure 5: Deterministic ranked-query compilation. (a) Signals, budget, and capabilities lower to $z = \langle r, k, \rho, h \rangle$. (b) Route r selects A–D; only hybrid C owns RRF. Width k includes retrieval fan-out and the pre-rerank k' cut; ρ is an optional reranker, and graph expansion h is available only on structural D.

Pointwise reranking. S_{dense}R truncates to k' and applies a Qwen3 pointwise reranker before final top-k_{out}. Q1 sweeps $k' \in \{30, 50, 100\}$ and model size.

Each candidate is paired with the query and scored independently. The pre-rank cut k' therefore controls both maximum recoverable recall and the number of model inferences; it is not interchangeable with final output width.

Graph expansion and experimental fusion. Automatic route D expands sparse seeds and optionally reranks without RRF. Q1 instead instantiates an explicit dense–graph ablation: S_{dense}XFus expands the first s dense seeds by one semantic-edge hop and combines dense and structural ranks [11]:

$$\text{score}(u) = \frac{w_d}{\kappa + r_d(u)} + \frac{w_g}{\kappa + r_g(u)},$$

where missing ranks contribute zero. We fix $w_d = 1$, $\kappa = 60$, select w_g on repository-disjoint tuning repositories, report the ablation on held-out repositories, and optionally append R. Explicit fusion prevents graph candidate generation from being mistaken for dense rescoring.

BM25, regex, and Zoekt serve lexical requests but are not compared in Q1; we do not infer their relative quality from the dense-plan experiment.

7.3 Static and Live Navigation

Definition/reference requests contain a repository-relative file, position, options, and limit. MCP converts its one-based line once before the zero-based provider interface. The server instantiates static graph/occurrence serving; agent and evaluation paths can instead inject live JSON-RPC. Both normalize to one location schema, and Q3 projects it to the path/start-line set in Sec. 4.2.

Metadata records backend, capability, commit, and granularity but does not show that static and live outputs agree. Q3 therefore compares their normalized location sets before timing. The MCP surface remains static, and no online classifier predicts unseen-request compatibility.

7.4 Context Policies

All arms share issue, tools, budget, turn cap, model parameters, repository commit, and answer format. Let C_k^{ctx} denote the frozen top-k embedding-ranked L_2 code blocks; Q5 fixes $k = 10$. *Grep/read* starts without candidates, while *eager* and *compact* receive the same C_{10}^{ctx}. Let j index the model invocation whose completed tool batch first contains a successful read. After that batch produces H_j, compact applies the one-time transition

$$H_j = [s, q \parallel C_{10}^{\text{ctx}}, e_{1:j}] \longrightarrow \widehat{H}_j = [s, q \parallel d_j], \tag{4}$$

where s is the system prompt, q the clean issue, $e_{1:j}$ the discarded exploration transcript, and d_j a direction seed containing deduplicated read paths, the latest successful read result in full, and a bounded prefix of the latest nonempty assistant message. Invocation $j + 1$ consumes $\widehat{H}_j$, and later invocations append normally. Thus the transition creates one new cache prefix, while every subsequent invocation extends that fixed prefix and can reuse its KV cache. The rewrite is deterministic and invokes no summarization model; it changes retention, not retrieval. Q5 sums the full trajectory, including tokens spent before the rewrite, under the quality threshold in Sec. 4.3.

Candidate assembly remains experiment-side; bounded history and one-time compaction are reusable runner mechanisms. At the transition, only the latest nonempty assistant message is shortened—to its first 600 characters—as a direction cue; the retained read content and final answer remain intact.

8 IMPLEMENTATION

CodeNib is implemented in Python 3.10+ and comprises compiler, graph, index, agent-runtime, and serving-adapter packages, including stdio MCP. Evaluation runners own sampling, arm wiring, timing, and post-run output comparisons; reusable operators and history policies remain in core packages.

8.1 Repository View Compiler and Runtime

`IndexCompiler` implements the initial-build path and records builder failures while retaining successful artifacts. `GraphPatcher` and `CodeVectorStore.delta_update` implement the two evaluated maintainer paths. `ServerContext` loads available resources once for process-resident serving; agents load only skill-required views. Versioned graph pickles fail with a rebuild instruction, and a C++/pybind SCIP decoder is tested for schema parity.

LSP-shaped skills call an injected static or live provider and record the chosen implementation. MCP currently instantiates static serving; live JSON-RPC is an agent/evaluation path, not an automatic fallback.

For MCP, `ServerContext` loads vector, BM25, graph, and Zoekt resources once at process startup. The agent runtime instead owns provider selection, tool schemas, history state, context policy, and per-run traces. LSP-shaped skills receive an injected provider, so static and live implementations share the agent-facing location schema while traces retain which backend actually served a request.

8.2 Tool and Serving Adapters

Agent skills and stdio MCP wrap the same operators and normalized results with separate model-facing schemas. MCP exposes semantic, BM25, regex, and Zoekt search; dependency, definition, reference, and route tools; and `get_manifest`. Missing optional indexes return explicit errors. Experiments invoke the operators through benchmark or agent harnesses; MCP is not an experimental factor.

Search returns snippets, while LSP-shaped tools return compact locations to keep high-fanout references small. Authentication, multi-tenancy, and network transport are outside the evaluated stdio implementation.

The concrete search surface is `search_semantic`, `search_bm25`, `search_regex`, and `search_zoekt`; structural tools expose dependency subgraphs, definitions, references, and routes. `get_manifest` reports commit, languages, artifact status, and capabilities. Missing optional indexes surface an explicit tool error rather than a silent fallback under the same name.

8.3 Runtime Entry Points and Observability

The public `query` entry point accepts exactly one of a repository path, caller-opened contexts, or a manifest. Only repository-path mode may compile and cache views before constructing `AgentRunner`. Evaluation uses manifest mode, which opens artifacts named by M_c and cannot build inside the measured loop.

`AgentRunner` separates index-backed `SkillRegistry` capabilities from ordinary `ToolRegistry` primitives. Loading validates required artifacts; `ResourceGuard` filters unavailable skills and warns on staleness. This setup checks recorded runtime state, not Q4's independent fresh-build equality.

`AgentRunTrace` records ordered events and context state, and `UsageTracker` records provider tokens. Evaluation consumes these records without embedding scoring, policy selection, or fresh-build comparisons in the trace schema.

9 EVALUATION

We evaluate CodeNib from retrieval operators to the agent loop. The experiments answer five questions:

- **Q1** What quality–latency tradeoffs arise when dense retrieval is composed with structural expansion and pointwise reranking?
- **Q2** How do dense-index designs trade construction time, search latency, and retrieval quality?
- **Q3** For graph-anchored symbol-navigation requests, how often does a static index reproduce the live LSP's normalized path/start-line set, and what marginal latency does it save on matching requests?
- **Q4** How often do incremental graph and vector updates match independently rebuilt targets, and what speedup do matching transitions achieve?
- **Q5** What token–quality tradeoffs arise from context-delivery policies across agent models and workload slices?

Table 2: Evaluation matrix and frozen record counts.

Study	Compared arms/providers	Frozen records
Q1 retrieval	5 dense embedders; partial 3-reranker matrix	100 snapshots; rerank $n = 98$–100
Q1 graph	Dense/graph, ± fixed 4B; tune/freeze w_g	58 tune/42 held-out; 15/10 repos
Q2 index	$L_0/L_2 \times 5$; Qwen-0.6B Flat/IVF/HNSW	500 pairs; 100 ANN
Q3 nav.	Static index versus 5 live LSPs	1,000 requests; 100 snapshots
Q4 maint.	Graph rebuild/file/symbol; vector rebuild/incremental	40 (8 repos); 33 G/31 V source-changing
Lifecycle	Materialize/load/mixed trace; isolated/resident	25 snapshots; 1,050 requests (3 reps)
Q5 context	Grep/read, Eager, Eager+Compact; 5 models	7,500 core trajectories
Embedding	SweRank-Small (137M)/Large (7B) [52]; Qwen3-Embed-0.6B/4B [75]; Jina-Code-1.5B [31]	Q1–Q2
Reranking	Qwen3-Reranker-0.6B/4B/8B [75]	Q1
Agents	Claude Haiku 4.5 [3]; Qwen3.5-9B/27B [51]; Gemma 4-12B-IT [18]; Gemini 2.5 Flash [10]	Q5

The questions follow the system boundary rather than assuming one common quality metric. Q1 tests ranked plans; Q2 separates dense construction, search, and approximation; Q3 tests navigation; and Q4 tests maintenance. The mixed trace checks compiler–runtime composition and stage costs; Q5 measures history delivery. We evaluate repository interaction and localization, not patch generation or issue resolution. MCP is not an experimental factor: no arm varies the serving protocol or attributes quality or latency to MCP itself.

9.1 Experimental Setup

Workloads. Table 2 aligns each study with its compared arms and frozen record counts. Q1–Q3 use the frozen CodeNib Base split drawn from SWE-Bench Verified [27, 47] and Multilingual [29]. Its five repositories per language group span file-count percentiles, and its instances span Opus-assigned difficulty strata. The query is the issue body; targets are files and pre-patch L2 blocks intersected by the developer patch. Only graph fusion partitions Base for tuning; the other Q1 operating points use all 100 snapshots. Appendix A records dataset construction.

Agent experiments use CodeNib Synthesis. Anthropic distinguishes pinned model IDs from convenience aliases [4]; this artifact records only the provider alias `opus`, not its immutable resolution. The same observed alias generates candidates and judges them in a separate pass, then deterministic checks reject duplicate identifiers, empty targets, and language-inconsistent files. One trajectory per query, policy, and model gives 7,500 trajectories: 2,500 query–model cells with three policies each. Intervals measure snapshot variation, not repeated model sampling. Haiku's compact arm was backfilled after its baseline/eager sweep under the same observed model ID

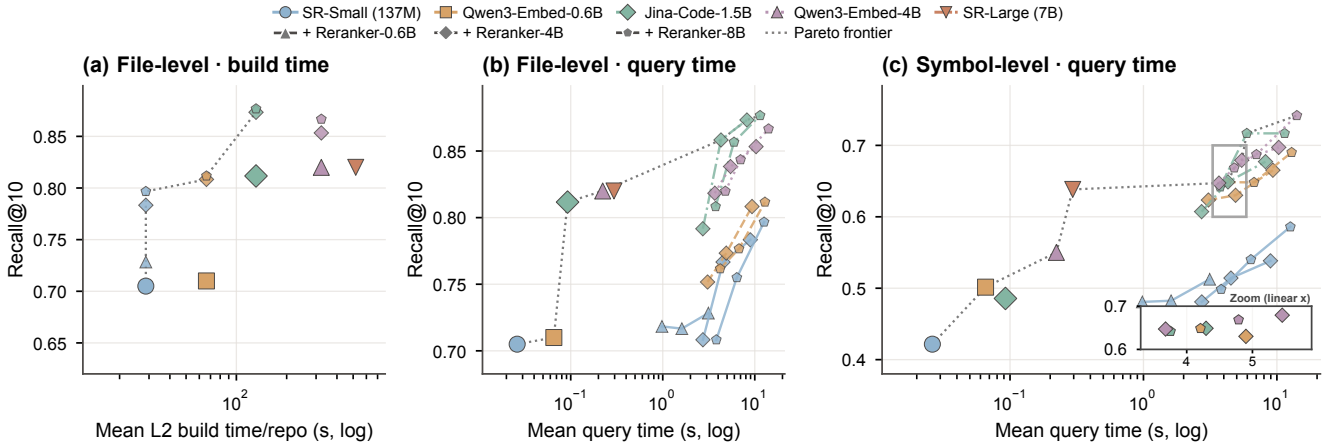


Figure 6: Embedding and pointwise-reranker operating points on the 100-snapshot corpus; Qwen-4B/8B score means use $n = 99/98$ at $k' = 30/50$, and all other score means use $n = 100$. (a) File Recall@10 versus mean L2 index-build time for the callable index used by retrieval. (b) File and (c) symbol Recall@10 versus mean query time. The SweRank embeddings SR-Small and SR-Large have approximately 137M and 7B stored parameters, respectively. Connected rerank points sweep pre-rerank cuts $k' \in \{30, 50, 100\}$; dotted lines show the empirical non-dominated points within each panel.

and harness configuration; both Qwen sizes and Gemma 4-12B use complete frozen three-arm matrices. Gemma is served by vLLM 0.25.1 from a pinned Hub revision; Gemini 2.5 Flash is accessed through Vertex AI with thinking disabled. Appendix H records provider controls, observed revisions, and remaining provider-time drift.

The lifecycle trace reuses the 25 synthesis snapshots and fixes 20 source-query sessions per snapshot. Each query runs once through Qwen3-Embedding-0.6B and once through BM25; two additional deterministic definition probes include static navigation. The resulting count is therefore $20(1+1)+2 = 42$ service requests, rather than a separately chosen scale. The selected definitions are nonempty, stable, and return the same normalized path/start-line sets from the static and live providers in every calibration comparison. References are excluded because their normalized sets drifted during calibration. This mix is not an observed agent-tool distribution. Runs use isolated quiescent checkouts and fresh runtime processes, but warm machine caches; Appendix G specifies the boundary.

Incremental maintenance follows five first-parent commit transitions in each of eight repositories ($n_{\text{share}} = 5$), with two repositories in each of Go, Python, Rust, and TypeScript/JavaScript. The graph scope includes language-source tests; the vector builder's declared scope excludes test-like paths. This yields 33 graph and 31 vector source-changing transitions. No-source transitions remain in the artifact but are excluded before computing update ratios. Graph arms start from the same base artifact and compare a fresh target rebuild, file replacement, and symbol repair; vector arms compare a fresh target rebuild with content-addressed embedding reuse and FAISS delta update. The fresh target supplies both the rebuild baseline and the offline equivalence reference; its construction and comparison are not added to incremental update latency. Each incremental graph arm reuses one LSP process per repository sequence, whereas each graph rebuild constructs an independent target. The study spans eight repositories but does not estimate production arrival rates or concurrent update throughput.

Ground truth. We project patch line ranges onto pre-patch L2 chunks using the tree-sitter hierarchy of Sec. 5. Retrieval runs against the base commit, so only pre-patch files and blocks can be targets. Added files and added-only symbols are excluded because they have no retrievable source unit in that snapshot. Edited blocks are a localization proxy, not an assertion that every useful supporting block was changed by the patch.

Metrics and inference. We use the operation-specific metrics summarized in Sec. 4.3; Appendix B gives their formal definitions and examples. Recall and match-rate estimates macro-average per-instance scores. File Success is an all-target indicator, Neighbor Recall measures exact-Flat fidelity rather than patch relevance, and AnswerRecall@5 scores the first five deduplicated source spans committed in the final answer. An answer with no usable in-scope committed span receives zero recall. Agent intervals use 10,000 bootstrap samples clustered by repository snapshot. Graph effects use 20,000 paired repository-clustered bootstraps: pointwise in Figure 7(a) and max-deviation simultaneous over ten cross-embedding contrasts. ANN means use 20,000 clustered resamples. Latencies are warm wall-clock measurements unless stated otherwise. Lifecycle medians and accounting projections use 10,000 snapshot-level percentile bootstraps. Incremental transitions use the offline output comparisons in Eqs. (1) and (2); graph maintenance also checks the changed-file projection. Vector maintenance requires exact document identities, numerical vector equivalence, and exact ordered Flat top-k replay. Transition ratios follow Eq. (3). Successful-arm ratios remain visible, but conditional summaries retain only source-changing transitions that match the independently rebuilt target.

Execution environment and models. Unless stated otherwise, local inference uses one NVIDIA H100 PCIe GPU (80 GB), and CPU experiments use two Intel Xeon Gold 5416S processors; FAISS search uses one CPU thread. Agent temperature is zero, and both eager-context policies share the frozen Qwen3-Embedding-0.6B top-10 L_2 result. Historical retrieval and Haiku/Qwen agent runs record model IDs,

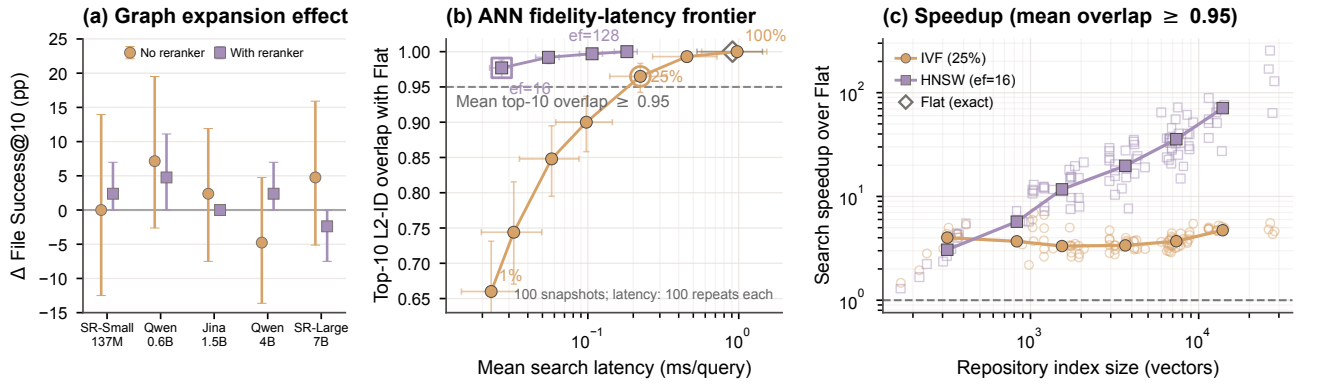


Figure 7: Task-level graph and physical ANN ablations. (a) Paired graph-minus-dense ΔFile Success@10 (all target files in the top-10 distinct-file prefix). (b–c) 100 Qwen3-Embedding-0.6B L_2 indexes. (b) Mean exact-Flat top-10 L_2-ID overlap versus search latency; outlines mark each family's fastest configuration at mean overlap ≥ 0.95. (c) Faint marks are snapshot speedups; six large marks per curve are log-size-bin medians joined as guides.

not immutable revisions; the artifact recovers available cache revisions, while Haiku hardware and revision remain unobserved. The Gemma run pins its Hub commit, serving template, and vLLM version; the Gemini run records its Vertex configuration and access date, but not an immutable provider revision. The lifecycle trace separately pins its embedding snapshot and build configuration at execution time: batch size 32 and a 2,048-token document cap fixed before the final batch. Appendix E gives model architectures, prompts, truncation, batching, and operator hyperparameters.

9.2 Q1: Retrieval-Plan Tradeoffs

Dense retrieval and reranking. Figure 6 maps empirical operating points and non-dominated frontiers under three distinct budgets rather than selecting one "best" pipeline. Mean dense-query time spans 26–295 ms. At $k = 10$, file recall rises from 0.705 to 0.820 across embedding families, while symbol recall spans 0.422–0.638. Embedding choice shifts both recall and latency; the measured higher-recall operating points generally pay at both build and query time.

Pointwise reranking moves the high-recall frontier at a much larger online cost. For example, Jina plus the 4B reranker at $k' = 50$ reaches 0.858 file Recall@10 in 4.29 s, compared with Jina dense at 0.812 in 92 ms: a 4.6-point gain at 46.6× latency. At symbol level, the highest measured recall is 0.742 for Qwen3-Embedding-4B plus the 8B reranker at $k' = 100$, requiring 14.1 s. Across the three measured cuts, larger k' generally raises recall and latency. On this host, mean dense-query latency stays below 300 ms across embeddings, whereas reranking requires seconds. The measured file- and symbol-recall maxima occur at different embedder–reranker pipelines.

Graph expansion. To isolate the graph operator, dense and graph arms share the same embedding index and per-instance candidate budget. We retrieve 300 L2 chunks, expand the first ten seeds over incoming and outgoing reference edges, and fuse semantic and structural ranks with weighted RRF [11]. On the 58-snapshot, 15-repository tuning partition, we sweep $w_g \in \{0.25, 0.5, 0.75, 1, 1.25, 1.5, 2\}$ and select lexicographically by File Success@10, then @5 and @1. This uniquely selects $w_g = 0.5$, which we freeze before reporting effects on 42 snapshots from 10 disjoint repositories, avoiding evaluation on the labels that selected the weight.

In Figure 7(a)'s before-rerank series, point estimates range from −4.8 points (Qwen3-Embedding-4B) to +7.1 ([−2.6, +19.5]; Qwen3-Embedding-0.6B). Every model-level interval in both series includes zero; so do all ten familywise cross-embedding intervals. The study establishes neither a general gain nor an embedding-specific routing rule. Expansion adds 15–39 ms to median non-reranked latency, while reranked arms cost 2.7–6.8 s in total. Panel (a) uses all-target File Success@10 rather than mean File Recall@10, so its effects are not shifts of Figure 6's frontier. Expansion remains an optional plan requiring deployment-specific validation.

Answer to Q1. Reranking is a controllable seconds-scale tradeoff; graph expansion remains unresolved, so ρ and h require separate plan policies.

9.3 Q2: Index Construction and Search

Q1 measures composed retrieval plans. Q2 separates offline dense-index construction from online vector-search cost and approximation error.

Build-time scaling. Figure 8 separates repository size, granularity, and model choice across 500 L0/L2 pairs. Median construction spans 3.8–56.7 s for L0 and 19.3–285.0 s for L2; within a model, L2/L0 is 5.0–6.4×. All ten LOC fits are positive; in parameter-count rank, L0 and L2 slopes rise from 0.029–0.644 and 0.186–4.282 s/kLOC. This matches the first-order model $T_{\text{build}} \approx T_{\text{enc}}(m, W_\ell) + O(N_\ell d_m)$: LOC proxies encoded-token work W_ℓ, while Flat writes N_ℓ vectors of dimension d_m. The trend is consistent with this decomposition; architecture, chunking, token lengths, and batching explain residuals, so the fits are descriptive rather than universal laws. Timers start after chunking and exclude checkout, compiler/SCIP execution, and graph decoding.

Panel (c) shows median dense-query latency rising from 20.9 to 233.8 ms. Output dimension (768–3584) and parameter count have the same rank order, while the timed query combines encoding, an $O(N_\ell d_m)$ Flat scan, and fixed-k result materialization; separately

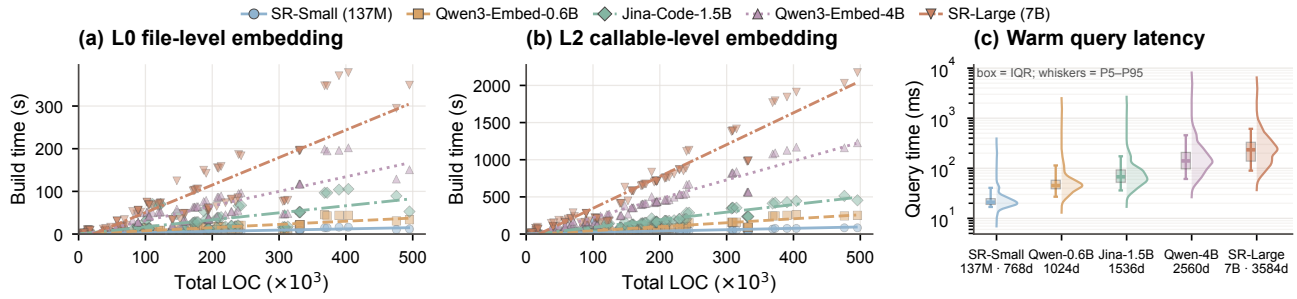


Figure 8: Dense-view construction and query profiles over 100 snapshots. (a) L0 file and (b) L2 callable construction time versus repository lines of code (LOC); lines are per-model fits. (c) Warm end-to-end L_2 dense-query latency; labels give parameter scale and output dimension d. Half violins show distributions, boxes show IQR, and whiskers show P5–P95.

timed index loading is excluded. The trend is descriptive, not causal; the FAISS ablation isolates scan organization.

Index-family ablation. With vectors and queries fixed, we compare exact `IndexFlatIP` [28], inverted-file `IndexIVFFlat`, and `IndexHNSWFlat` [41]. IVF sweeps probe fractions of 1%, 2%, 5%, 10%, 25%, 50%, and 100%. HNSW fixes $M = 32$ and $ef_{\text{construction}} = 200$ while sweeping $ef_{\text{search}} \in \{16, 32, 64, 128\}$. Each method–snapshot cell uses 10 warmups and 100 timed searches. In this descriptive sweep, each approximate family contributes its fastest configuration with across-snapshot mean Neighbor Recall@10 at least 0.95. Figure 7(b–c) shows that HNSW with $ef_{\text{search}} = 16$ reaches 0.977 neighbor Recall@10 and reduces mean FAISS search from 0.910 to 0.0268 ms (33.9×). IVF at 25% probes reaches 0.965 at 0.223 ms (4.1×). Both preserve Flat's patch-target-file Recall@10 on every snapshot (macro mean 0.620).

These ratios are component-local: HNSW reduces mean FAISS search by 0.883 ms, whereas the complete dense-query median is 45.1 ms. It also raises mean index-only build time from 6.4 ms (Flat) to 2.00 s and mean serialized size from 21.74 to 23.18 MiB; the IVF means are 0.889 s and 22.33 MiB. At the measured means, extra construction amortizes after about 1,300 searches for IVF and 2,300 for HNSW; these index-local crossovers exclude loading and updates. Flat is therefore simpler at this scale; larger indexes and query rates remain unmeasured, and the FAISS ratios are not end-to-end speedups.

Answer to Q2. At this scale, embedding and granularity dominate; ANN's sub-millisecond saving justifies additional construction only under sufficient query reuse.

9.4 Q3: Static versus Live Symbol Compatibility and Latency

Q1–Q2 concern ranked retrieval. Q3 turns to symbol navigation, which returns a set of source locations rather than a ranking.

This experiment changes only the backend of the same agent-visible `definition` or `references` request. For each snapshot, we sort source-mappable reference-edge anchors, select five evenly spaced positions, and issue both capabilities. Definitions return at most eight locations; references return at most 40 and include the declaration. After provider-side limits, normalization deduplicates and sorts repository-relative path/start-line pairs; this comparison

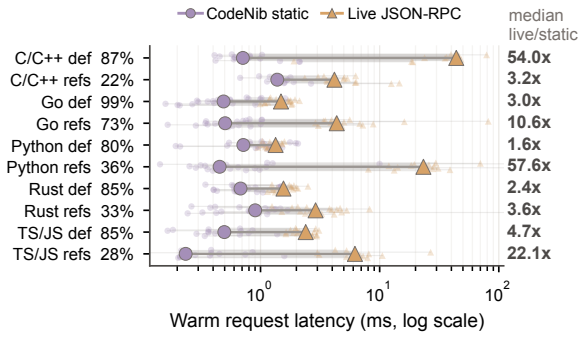


Figure 9: Static-index versus live JSON-RPC replay over 100 snapshots. Labels report path/start-line match rates. Faint pairs are per-snapshot medians of request-level repetition medians; large points and right-hand values are pooled request medians and median live/static ratios. Non-matches are excluded from conditional latency summaries.

omits characters, end ranges, and metadata. After the normalized outputs stabilize, we apply the equality test once, measure each matching request ten times, and summarize its repetition median. Because positions originate from the static graph, the match rate does not estimate arbitrary editor requests. The live providers are clangd, gopls, basedpyright, rust-analyzer, and typescript-language-server for C/C++, Go, Python, Rust, and TypeScript/JavaScript (TS/JS), respectively.

Figure 9 shows that across 1,000 requests, 632 (63.2%) normalized path/start-line sets match: 437/500 definitions (87.4%) and 195/500 references (39.0%); 630 of the 632 matches return non-empty results from both providers. The matches yield 6,320 timing rows. Summarizing ten-repetition request medians gives static/live p50 of 0.62/2.26 ms, a median paired saving of 1.69 ms, and a median live/static ratio of 4.72×. Match rate is the limiting result: definitions range from 80–99% across languages, while references range from 22–73%. The static provider therefore does not reproduce the live provider on all requests. These measurements characterize conditional marginal latency only where the normalized path/start-line sets match. Even definitions mismatch on 12.6% of sampled anchors,

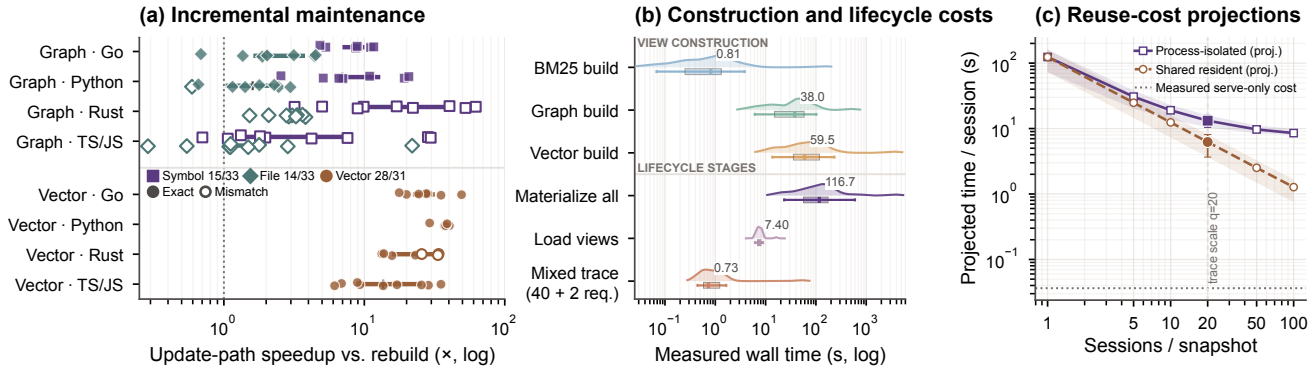


Figure 10: Incremental maintenance and lifecycle accounting. (a) Update speedup; fill denotes exact output match, labels give exact/measured transitions, and segments/ticks show IQR/median. (b) View-construction components and complete, non-additive materialize–load–serve stages; the trace records 20 paired dense/BM25 queries plus two definitions (42 requests). (c) Projections from measured B, L, S at $q \in \{1, 5, 10, 20, 50, 100\}$; open marks are projected medians joined as guides, bands are bootstrap 95% CIs, and filled $q = 20$ marks identify the controlled trace scale.

so capability type cannot route safely: 4.72× is a conditional opportunity, not an achieved workload speedup; the study supplies neither full-response equivalence nor an online oracle.

Timing excludes graph loading, server startup, and warmup; it measures marginal cost after readiness. Frozen paired replay removes agent tool-choice and model API variance while preserving repository-specific request positions and provider outputs.

Answer to Q3. Static navigation is lossless only under the matched location projection; without an online compatibility test it is a conditional provider, not an automatic replacement.

9.5 Q4: Incremental View Maintenance

Q3 evaluates requests against a fixed repository commit. Q4 evaluates the per-transition speedups in Eq. (3), then applies the post-timing output comparisons. For every graph source change, we execute a fresh LSP rebuild, file-level replacement, and the symbol-level path in Figure 3. Deterministic definition/reference requests are anchored in changed files. Vector maintenance compares a fresh Flat index with content-addressed embedding reuse and deterministic Flat replay.

Figure 10(a) reports all 33 graph and 31 vector source-changing transitions; no-source rows are outside the ratio population, and mismatches remain open. Symbol repair matches the independent rebuild on 15/33 (45.5%) transitions; among those transitions, its amortized speedup has median 8.67× and IQR 5.95–10.99×. File replacement matches on 14/33, with median 1.95×; on the 14 transitions where both paths match, symbol repair is 4.25× faster at the median. All Go (7/7) and Python (8/8) symbol updates pass. Rust and TS/JS have high median edge F1 (99.12% and 97.61%) and serving agreement (97.40% and 99.53%), but none passes both offline whole-graph and serving checks; their points therefore remain open.

Vector maintenance matches the independent rebuild on 28/31 (90.3%) source-changing transitions, with median speedup 25.44× and IQR 15.24–35.15×. All 31 persisted artifacts reproduce target document identities and vectors; three Rust rows fail exact ordered and set top-10 replay and are excluded from the matching-transition speedup summary.

A separate four-case same-commit rebuild audit gives median fresh/fresh edge F1 of 99.75% and serving agreement of 99.35%, with changed-scope exactness on 3/4 cases. Thus a cold live-LSP rebuild is not a deterministic oracle for every Rust/TS graph. We report exact-match counts and raw fidelity together rather than tuning a post hoc tolerance. Appendix F gives the protocol, language breakdown, and failure audit.

Answer to Q4. Vector reuse matches an independent rebuild on more transitions than graph repair (90.3% versus 45.5%). This offline comparison identifies where the measured fast path preserved output; it is not a runtime guarantee.

9.6 Cross-Stage Lifecycle Accounting

Figure 10(b) separates two granularities. The upper group's BM25, structural-graph, and vector construction medians are 0.81, 37.97, and 59.48 s; they contribute to the lower group's complete materialization stage and are not query latency. The lower group reports materialization $B = 116.7$ s (95% CI 65.6–153.9), fresh-process runtime loading $L = 7.40$ s (6.99–8.07), and one warm trace $S = 0.727$ s (0.593–1.079). These two granularities are non-additive. The median artifact occupies 160.3 MiB.

The trace fixes $N = 20$ source-query sessions, the reuse scale in panel (c). Each query runs once through dense retrieval and BM25; two calibrated definition probes include static navigation. Hence $20(1 + 1) + 2 = 42$ is a derived request count, not a chosen workload scale. This reproducible build–load–serve trace exercises all three view families but is not an observed agent distribution. Panel (c) projects 13.20 versus 6.24 s/session at $q = 20$ and 8.53 versus 1.27 s/session at $q = 100$ for process-isolated and shared-resident runtimes. The $1/q$ term approaches measured serve-only S/N, not an end-to-end or hardware floor; connected points are accounting projections, not six executed scales or a break-even claim. Appendix G gives intervals and the asymptotic interpretation.

9.7 Q5: Agent Context Delivery

Q5 separates delivery from retention. Grep/read has no injected L_2 candidates; Eager injects frozen C_{10}^{ctx}; Compact receives the same candidates and rewrites history once after the first successful read,

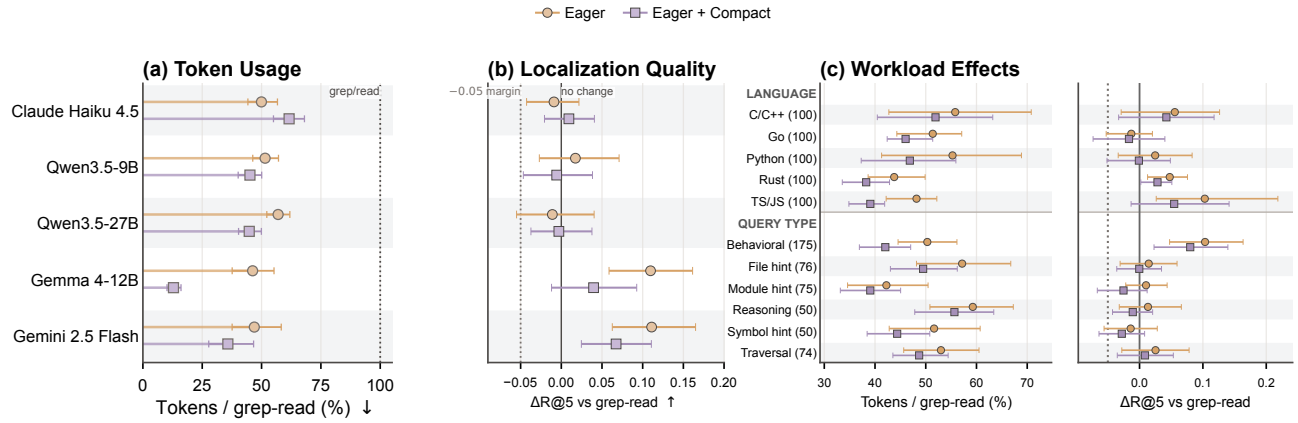


Figure 11: Agent context-policy effects. (a–b) Model-level token and AnswerRecall@5 **effects for Eager/Compact against paired grep/read; candidates are shared. (c) Pooled workload effects (labels give query counts across five models and three policies). Whiskers are snapshot-clustered 95% intervals. Compact resets the prefix once; tokens are provider-reported trajectory totals.**

then appends. Thus Eager/Compact isolates retention. Figure 11(a–b) normalizes against paired grep/read because baseline volume spans 25.9–159.7 thousand tokens per query.

Token axes sum provider-reported prompt/completion tokens rather than cache-adjusted cost. Eager keeps its prefix; Compact resets once and reuses the shorter prefix. Cache hits and prefill latency are unmeasured.

All models use ordinary tools, a 16-turn cap, and frozen top-10 candidates; accounting includes full provider-reported trajectories and fixed answer-format prompts, with zero recall for invalid answers. The rule in Sec. 4.3 selects Eager for Haiku and Compact otherwise. In model order, these use 49.9, 45.1, 44.8, 12.9, and 35.8% of paired grep/read tokens; ΔAR@5 ranges from -0.009 to $+0.067$, with all lower bounds above -0.05. Selection is unchanged at $k \in \{1, 3, 5, 10\}$; Qwen-27B Eager alone misses the margin (Appendix Table 9).

Compaction is not uniformly dominant: Compact/Eager token usage ranges from 123.3% (Haiku) to 27.9% (Gemma). Gemma/Gemini trade 0.070/0.044 AnswerRecall@5 against Eager while clearing the baseline gate. Panel (c) is descriptive: several quality intervals cross the margin, so we make no subgroup-routing claim. Appendix H gives paired intervals and validity audits.

Answer to Q5. Candidate injection and one-time compaction occupy model-dependent token–localization operating points; compaction is not uniformly dominant.

10 DISCUSSION AND FUTURE DIRECTIONS

A concurrent heterogeneous repository database. CodeNib currently materializes independently managed views for quiescent repository snapshots. A full database service should admit concurrent agents and updates, maintain per-view versions under atomic publication, and provide recovery, multi-tenancy, and cost-based routing across lexical, dense, and structural state. The central challenge is to preserve each view's explicit validity boundary while accommodating heterogeneous hardware, update rates, and freshness requirements.

Agent harnesses, post-training, and a data flywheel. The runtime records requests, selected views, tool interactions, delivered context, and their costs. A post-training-compatible harness could turn these traces into supervision for retrieval routing, tool use, context selection, and compaction, then feed evaluated outcomes back into subsequent data collection and fine-tuning. The serving layer would thereby become both an execution substrate and a controlled source of training data rather than remaining tied to one fixed agent policy.

Resource-efficient context serving. Models and workloads differ in useful context, latency tolerance, and CPU/GPU demand. A production scheduler should coordinate CPU-oriented lexical, graph, parsing, and navigation work with GPU-oriented embedding, reranking, and model inference. It should also decide what to batch, preload, retain, or evict under memory, token, and latency budgets.

Scope. Our measurements characterize controlled repository, navigation, maintenance, and localization workloads. They do not yet establish concurrent publication, learned online scheduling, or gains from post-training; these define the next system boundary.

11 CONCLUSION

CodeNib's repository view compiler (C1) materializes lexical, dense, and structural views per commit under explicit validity boundaries. Its graph-repair and vector-reuse paths (C2) reach 8.67×/25.44× median speedups on transitions matching independent rebuilds. Its cost-visible runtime (C3) serves ranked plans, static/live navigation, and bounded context policies: compatible static requests have a 4.72× median per-request live/static latency ratio, while selected policies preserve the localization margin with 50–87% fewer provider-reported trajectory tokens.

Across these stages, Pareto and quality–cost analyses keep quality, compatibility, update fidelity, latency, and token usage distinct. Together, they frame repository context as a measurable serving problem without collapsing exact maintenance, projection-compatible navigation, and policy-dependent localization into one unqualified notion of reuse.

REFERENCES

[1] Anomaly. 2025. OpenCode: The open source coding agent. https://opencode.ai/docs/tools. Accessed 2026-07-25; LSP integration is experimental.

[2] Anthropic. 2024. Model Context Protocol: Transport Specification. https://modelcontextprotocol.io/specification/2024-11-05/basic/transports. Version 2024-11-05; accessed 2026-07-25.

[3] Anthropic. 2025. Introducing Claude Haiku 4.5. https://www.anthropic.com/news/claude-haiku-4-5

[4] Anthropic. 2026. Model IDs and Versioning. https://platform.claude.com/docs/en/about-claude/models/model-ids-and-versions Accessed 2026-07-25.

[5] Ishaan Bhola, Adithyan Krishnan, Sravanth Kurmala, and Mukunda NS. 2026. Code Isn't Memory: A Structural Codebase Index Inside a Coding Agent. arXiv:2606.22417. https://arxiv.org/abs/2606.22417

[6] Shravan Chaudhari, Rahul Thomas Jacob, Mononito Goswami, Jiajun Cao, Shihab Rashid, and Christian Bock. 2025. SpIDER: Spatially Informed Dense Embedding Retrieval for Software Issue Localization. arXiv:2512.16956. https://arxiv.org/abs/2512.16956

[7] Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-Guided LLM Agents for Code Localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vienna, Austria, 8697–8727. https://doi.org/10.18653/v1/2025.acl-long.426

[8] Tsvi Cherny-Shahar and Amiram Yehudai. 2026. Repository Intelligence Graph: Deterministic Architectural Map for LLM Code Assistants. arXiv:2601.10112. https://arxiv.org/abs/2601.10112

[9] CocoIndex contributors. 2025. CocoIndex. Incremental data-transformation framework. https://cocoindex.io/docs-v0/ Accessed 2026-07-25.

[10] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, et al. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. arXiv:2507.06261. https://arxiv.org/abs/2507.06261

[11] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Büttcher. 2009. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. Association for Computing Machinery, New York, NY, USA, 758–759. https://doi.org/10.1145/1571941.1572114

[12] Douglas A. Creager and Hendrik van Antwerpen. 2023. Stack Graphs: Name Resolution at Scale. In *Evolution, Originality, and Community in Computer Science (EVCS) (OpenAccess Series in Informatics (OASIcs))*, Vol. 109. 8:1–8:12. https://doi.org/10.4230/OASIcs.EVCS.2023.8

[13] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Samuel Madden, David Maier, Tim Mattson, and Stanley B. Zdonik. 2015. The BigDAWG Polystore System. *ACM SIGMOD Record* 44, 2 (2015), 11–16. https://doi.org/10.1145/2814710.2814713

[14] Fabio James Fehr, Prabhu Teja S, Luca Franceschi, and Giovanni Zappella. 2025. CoRet: Improved Retriever for Code Editing. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Vienna, Austria, 775–789. https://doi.org/10.18653/v1/2025.acl-short.62

[15] Jia Feng, Zhanyue Qin, Cuiyun Gao, Ruiqi Wang, Chaozheng Wang, Yingwei Ma, and Xiaoyuan Xie. 2026. On the Effectiveness of Context Compression for Repository-Level Tasks: An Empirical Investigation. arXiv:2604.13725. https://arxiv.org/abs/2604.13725 Work in progress.

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[17] Paul Gauthier and Aider contributors. 2024. Aider: Repository map. https://aider.chat/docs/repomap.html. Accessed 2026-05-12.

[18] Gemma Team. 2026. Gemma 4 Technical Report. arXiv:2607.02770. https://arxiv.org/abs/2607.02770

[19] Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. 2026. Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents? arXiv:2602.11988. https://arxiv.org/abs/2602.11988

[20] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the 9th International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 209–218. https://doi.org/10.1109/ICDE.1993.344061

[21] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations (ICLR)*.

[23] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/170035.170066

[24] Gwihwan Go, Quan Zhang, Chijin Zhou, Zhao Wei, and Yu Jiang. 2026. LSPRAG: LSP-Guided RAG for Language-Agnostic Real-Time Unit Test Generation. In *Proceedings of the 48th International Conference on Software Engineering* (Rio de Janeiro, Brazil) *(ICSE '26)*. Association for Computing Machinery, New York, NY, USA. https://conf.researchr.org/details/icse-2026/icse-2026-research-track/147/LSPRAG-LSP-Guided-RAG-for-Language-Agnostic-Real-Time-Unit-Test-Generation arXiv:2510.22210; authors announce DOI 10.1145/3744916.3773189, not yet registered as of 2026-07-25.

[25] Yu Huo, Kun Zeng, Siyu Zhang, Yuquan Lu, Cheng Yang, Yifu Guo, and Xiaoying Tang. 2026. RepoShapley: Shapley-Enhanced Context Filtering for Repository-Level Code Completion. In *Findings of the Association for Computational Linguistics: ACL 2026*. Association for Computational Linguistics, San Diego, California, United States, 10390–10412. https://doi.org/10.18653/v1/2026.findings-acl.505

[26] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Issue Localization via LLM-Driven Iterative Code Graph Searching. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 3034–3045. https://doi.org/10.1109/ASE63991.2025.00249

[27] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues?. In *International Conference on Learning Representations (ICLR)*.

[28] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (July 2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[29] Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. 2025. SWE-bench Multilingual. SWE-bench benchmark release. https://www.swebench.com/multilingual.html Accessed 2026-07-25.

[30] Myeongsoo Kim, Chao-Chun Hsu, Dingmin Wang, Shweta Garg, Varun Kumar, and Murali Krishna Ramanathan. 2026. CODESTRUCT: Code Agents over Structured Action Spaces. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, San Diego, California, United States, 13290–13306. https://doi.org/10.18653/v1/2026.acl-long.607

[31] Daria Kryvosheieva, Saba Sturua, Michael Günther, Scott Martens, and Han Xiao. 2025. Efficient Code Embeddings from Code Generation Models. In *Fourth Workshop on Deep Learning for Code (DL4Code), NeurIPS*. https://openreview.net/forum?id=smR7RPdWZs arXiv:2508.21290.

[32] Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. 2026. ContextBench: A Benchmark for Context Retrieval in Coding Agents. arXiv:2602.05892. https://arxiv.org/abs/2602.05892

[33] Yichen Li, Qiye Lin, Yun Peng, Zhihan Jiang, Jinyang Liu, Chaozheng Wang, Yintong Huo, and Cuiyun Gao. 2026. One Size Does Not Fit All: Revisiting Code Context Engineering for Repository-Level Code Generation. *Proceedings of the ACM on Software Engineering* 3, FSE (2026), 2952–2975. https://doi.org/10.1145/3808138

[34] Zhuofeng Li, Haoxiang Zhang, Cong Wei, Pan Lu, Ping Nie, Yi Lu, Yuyang Bai, Shangbin Feng, Hangxiao Zhu, Ming Zhong, Yuyu Zhang, Jianwen Xie, Yejin Choi, James Zou, Jiawei Han, Wenhu Chen, Jimmy Lin, Dongfu Jiang, and Yu Zhang. 2026. Beyond Semantic Similarity: Rethinking Retrieval for Agentic Search via Direct Corpus Interaction. arXiv:2605.05242. https://arxiv.org/abs/2605.05242

[35] Zhihao Lin, Mingyi Zhou, Yizhuo Yang, and Li Li. 2026. How Much Static Structure Do Code Agents Need? A Study of Deterministic Anchoring. To appear in the 35th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). https://arxiv.org/abs/2606.26979 arXiv:2606.26979.

[36] Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. 2025. The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management. In *Fourth Workshop on Deep Learning for Code (DL4Code), NeurIPS*. https://openreview.net/forum?id=OHVzruJl5k arXiv:2508.21433.

[37] Jinshi Liu, Hanying Zuo, Congyin Cao, Anran Zhang, Yixuan Liu, and Xinzhou Xie. 2026. AOCI: Symbolic–Semantic Indexing for Practical Repository-Scale Code Understanding with LLMs. arXiv:2605.02421. https://arxiv.org/abs/2605.02421

[38] Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. 2026. Context as a Tool: Context Management for Long-Horizon SWE-Agents. In *Findings of the Association for Computational Linguistics: ACL 2026*. Association for Computational Linguistics, San Diego, California, United States,

20604–20617. https://doi.org/10.18653/v1/2026.findings-acl.1032

[39] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. 2025. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Association for Computational Linguistics, Albuquerque, New Mexico, 142–160. https://doi.org/10.18653/v1/2025.naacl-long.7

[40] George Ma, Anurag Koul, Qi Chen, Yawen Wu, Sachit Kuhar, Yu Yu, Aritra Sengupta, Varun Kumar, and Murali Krishna Ramanathan. 2026. SpecAgent: A Speculative Retrieval and Forecasting Agent for Code Completion. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, San Diego, California, United States, 17266–17327. https://doi.org/10.18653/v1/2026.acl-long.786

[41] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[42] Simon Marlow. 2022. Incremental indexing with Glean. Glean documentation, https://glean.software/blog/incremental/. Accessed 2026-07-15.

[43] Meta Engineering. 2024. Indexing code at scale with Glean. https://engineering.fb.com/2024/12/19/developer-tools/glean-open-source-code-indexing/. Accessed 2026-07-15.

[44] Microsoft. 2022. LSIF: Language Server Index Format specification, v0.6.0. https://microsoft.github.io/language-server-protocol/specifications/lsif/0.6.0/specification/. Accessed 2026-05-12.

[45] Microsoft. 2024. Language Server Protocol Specification. https://microsoft.github.io/language-server-protocol/. Accessed 2026-05-12.

[46] Han-Wen Nienhuys and Sourcegraph contributors. 2024. Zoekt: Fast trigram-based code search. https://github.com/sourcegraph/zoekt. Accessed 2026-05-12.

[47] OpenAI. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/. Accessed 2026-05-12.

[48] Oraios AI. 2026. Serena: The IDE for Your Coding Agent. Software, version 1.6.1. https://github.com/oraios/serena/tree/v1.6.1 Accessed 2026-07-25; MIT License.

[49] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=dw9VUsSHGB

[50] Junsong Pu, Yichen Li, and Zhuangbin Chen. 2026. TypeScript Repository Indexing for Code Agent Retrieval. arXiv:2604.18413. https://arxiv.org/abs/2604.18413 Tool demonstration; unrefereed preprint.

[51] Qwen Team. 2026. Qwen3.5-9B and Qwen3.5-27B. Official model cards at https://huggingface.co/Qwen/Qwen3.5-9B and https://huggingface.co/Qwen/Qwen3.5-27B. Accessed 2026-07-25.

[52] Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. 2026. SweRank: Software issue localization with code ranking. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2505.07849 arXiv:2505.07849.

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

[54] Brian Sam-Bodden. 2026. What Context Does a Coding Agent Actually Need to Act? arXiv:2607.09691. https://arxiv.org/abs/2607.09691

[55] SCIP contributors. 2026. SCIP: A Language-Agnostic Protocol for Indexing Source Code. Protocol specification and software, version 0.9.0. https://github.com/scip-code/scip/tree/v0.9.0 Accessed 2026-07-25.

[56] Chuanqi Shi, Miao Gao, and Zhiqiang Gao. 2026. AIRCoder: Adaptive Integration of Multi-dimensional Retrieval for Repository-level Code Completion. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, San Diego, California, United States, 25458–25470. https://doi.org/10.18653/v1/2026.acl-long.1166

[57] Sourcegraph. 2024. Sourcegraph: Code search and intelligence. https://sourcegraph.com/. Accessed 2026-05-12.

[58] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023. Is ChatGPT good at search? Investigating large language models as re-ranking agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Singapore, 14918–14937. https://doi.org/10.18653/v1/2023.emnlp-main.923

[59] Tamás Szabó. 2023. Incrementalizing Production CodeQL Analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. Association for Computing Machinery, 1716–1726. https://doi.org/10.1145/3611643.3613860

[60] The Cognition Team. 2025. DeepWiki: AI docs for any repo. https://cognition.com/blog/deepwiki. Accessed 2026-05-13; public, no-login wiki for any GitHub repository, powered by Devin.

[61] Tree-sitter contributors. 2026. Tree-sitter Parser and Node Documentation. https://tree-sitter.github.io/tree-sitter/using-parsers/1-getting-started.html. Accessed 2026-07-25.

[62] Martin Vogel, Falk Meyer-Eschenbach, Severin Kohler, Elias Grünewald, and Felix Balzer. 2026. Codebase-Memory: Tree-Sitter-Based Knowledge Graphs for LLM Code Exploration via MCP. arXiv:2603.27277. https://arxiv.org/abs/2603.27277

[63] Peiding Wang, Li Zhang, Fang Liu, Chongyang Tao, and Yinghao Zhu. 2026. CodeMEM: AST-Guided Adaptive Memory for Repository-Level Iterative Code Generation. In *Findings of the Association for Computational Linguistics: ACL 2026*. Association for Computational Linguistics, San Diego, California, United States, 16903–16917. https://doi.org/10.18653/v1/2026.findings-acl.834

[64] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*.

[65] Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. 2026. SWE-Pruner: Self-Adaptive Context Pruning for Coding Agents. arXiv:2601.16746. https://arxiv.org/abs/2601.16746

[66] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 235. PMLR, 53270–53290. https://proceedings.mlr.press/v235/wu24a.html

[67] Scott Wu and Cognition Labs. 2024. Introducing Devin, the first AI software engineer. https://cognition.com/blog/introducing-devin. Accessed 2026-05-12.

[68] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824. https://doi.org/10.1145/3715754

[69] Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. 2026. Reducing Cost of LLM Agents with Trajectory Reduction. *Proceedings of the ACM on Software Engineering* 3, FSE (June 2026), 1241–1263. https://doi.org/10.1145/3797084

[70] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent–computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. Neural Information Processing Systems Foundation, 50528–50652. https://doi.org/10.52202/079017-1601

[71] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=WE_vluYUL-X

[72] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. OrcaLoca: An LLM Agent Framework for Software Issue Localization. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 267. PMLR, 73416–73436. https://proceedings.mlr.press/v267/yu25x.html

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

[74] Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, and Xiaodong Gu. 2026. SWE-Explore: Benchmarking How Coding Agents Explore Repositories. arXiv:2606.07297. https://arxiv.org/abs/2606.07297

[75] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. arXiv:2506.05176. https://arxiv.org/abs/2506.05176

[76] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 1592–1604. https://doi.org/10.1145/3650212.3680384

[77] Zelin Zhao, Xizao Wang, Zhaogui Xu, Zhenhao Tang, Yongchao Li, and Peng Di. 2023. Incremental Call Graph Construction in Industrial Practice. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 471–482. https://doi.org/10.1109/ICSE-SEIP58684.2023.00048

A DATASET CONSTRUCTION AND GROUND TRUTH

Frozen dataset identities. CodeNib Base is the 100-row test split of `fishmingyu/codenib-base-dataset` at Hub revision `4eb84e2e8918474969ce68c5b06facf14d6be604`. The rows bind 100 unique (`repository`, `base commit`) snapshots from 25 repositories. CodeNib Synthesis is the five-config, 500-row test split of `sysevol-ai/codenib-synthesis` at revision `5ac36c39ef69bbfe2e14dac58b6067b8c350c53e`. The artifact records the parquet hashes and canonical row-identity hashes for both datasets.

Upstream provenance and task boundary. Neither split is a raw SWE-bench agent-resolution workload. CodeNib Base resamples Python issue–patch pairs from SWE-bench Verified and C/C++, Go, Rust, and TypeScript/JavaScript pairs from SWE-bench Multilingual [27, 29, 47]. It retains the repository, pre-solution `base_commit`, issue body, and developer patch. CodeNib does not run the upstream pass/fail tests. Q1–Q2 use the patch only to derive pre-patch localization labels; Q3 reuses the snapshots but samples independent graph-covered source positions. CodeNib Synthesis is a second-stage derivative of Base, not another upstream sample. It reuses one selected Base snapshot from each of the same 25 repositories and replaces the issue query with source-grounded localization queries. Thus the two workloads permit paired system analysis but are not independent evidence across repositories.

Language	Rows	Repos	LOC p50	L_0/L_2 p50
C/C++	20	5	206,900	279/7,483
Go	21	5	147,434	383/4,790
Python	20	5	158,938	379/6,690
Rust	20	5	104,917	404/4,110
TS/JS	19	5	57,942	251/1,751

Table 3: Frozen Base composition: per-snapshot medians, not sampling quotas.

Base repository and instance sampling. The collector first removes repositories with fewer than three candidate issues. Within each language group, it counts non-hidden files in each repository, sorts repositories by this count, and selects five approximately at the minimum, 25th, 50th, 75th, and maximum ranks; rounded-rank collisions use a deterministic center-out fallback. It then asks the recorded Claude Opus alias to classify every issue as low, medium, or high from the issue body and the first 6,000 patch characters. Classification runs in batches of ten, retries in smaller batches, and uses medium only if structured parsing still fails.

Ground-truth extraction checks out the exact base commit, parses patch target files, and tree-sitter-chunks supported files before and after applying the developer patch. A pre-patch symbol is retained when its source range overlaps a changed hunk and its content changes, or when it is deleted. The collector rejects a row if extraction fails, the target set is empty, any new symbol is required, or more than ten target blocks remain. It takes one available row from each difficulty stratum per repository, fills the remaining per-repository quota deterministically, and distributes shortfalls round-robin across language groups until reaching 100. The frozen result contains 56 low-, 36 medium-, and 8 high-difficulty rows. Of its 151 target blocks, 74 rows have one block, 15 have two, and 11 have three to six. These are purposeful strata rather than a population sample of repositories or issues.

Coordinates and evaluation targets. Tree-sitter chunk rows are stored as zero-based inclusive `CodeChunk` ranges. Dataset export adds one to both endpoints and emits one-based inclusive `CodeLocation` ranges. Retrieval runs only on the base snapshot. Consequently, the scored targets are the pre-patch modified or deleted source units and their enclosing files; added-only code has no retrievable target. The issue body is the Base query. File and block recall use the same frozen target records, without consulting the post-patch repository at query time.

Synthesis construction. The synthesis planner takes all five Base repositories in each language group and chooses one Base snapshot per repository: the snapshot whose prebuilt graph contains the most symbol vertices. This yields 25 snapshots, each assigned 20 queries. For each snapshot the nominal budget is seven behavioral, three file hint, three module hint, three traversal, two reasoning, and two symbol hint queries. Behavioral anchors are non-test source symbols of at least 100 characters, sampled from at most 24 candidates using size, symbol type, and graph connectivity; up to eight one-hop neighbors provide disambiguating context. Traversal rows instead choose a two- or three-symbol reference chain, expose only its anchor symbols, and reject verbatim leakage of hidden symbols.

Generation uses the recorded `opus` alias, a ten-turn cap, sampling seed 42, assignment seed 0, a 0.5 simple-query ratio, and three behavioral consensus runs. A separate call to the same alias judges target discrimination; failed rows enter at most three fix/regenerate attempts, while traversal generation allows two judge retries. The provider alias has no recoverable immutable revision. Deterministic validation rejects duplicate query IDs, empty targets, and target files outside the declared language group. The frozen counts are 175 behavioral, 76 file hint, 75 module hint, 50 reasoning, 50 symbol hint, and 74 traversal rows; one Rust traversal replacement accounts for the 76/74 imbalance. The quality report has zero deterministic errors and five non-`valid` judge warnings; excluding those five does not change which Q5 policy meets the localization margin for any model.

B METRIC DEFINITIONS AND WORKED EXAMPLES

The studies expose different output contracts, so their metrics are not interchangeable. Recall and match indicators are computed per instance and then macro-averaged; an instance with more targets therefore receives no extra weight.

Patch-target coverage and all-file success. For instance i at granularity $g \in \{\text{file}, \text{symbol}\}$, let Y_i^g be the nonempty target set and $R_{i,k}^g$ the first k distinct returned units; symbol targets use the evaluated L_2 callable identities. Over M instances,

$$\text{TaskRecall}^g@k = \frac{1}{M}\sum_{i=1}^{M}\frac{|R_{i,k}^g \cap Y_i^g|}{|Y_i^g|}.$$

At file granularity, repeated blocks from one path are collapsed before the cutoff. The stricter $\text{FileSuccess}_i@k = \mathbf{1}[Y_i^{\text{file}} \subseteq R_{i,k}^{\text{file}}]$ requires every target file; its macro-average is the reported

success rate. For example, target files $\{A, B\}$ and ranked blocks $[A{:}f_1, A{:}f_2, C{:}g, B{:}h]$ produce distinct-file order $[A, C, B]$. At $k = 2$, File Recall is $1/2$ and File Success is 0; at $k = 3$, both are 1.

Physical-index fidelity. Let $E_{i,k}$ and $A_{i,k}$ be the L_2 identity sets returned by exact Flat and an approximate index. Neighbor Recall is

$$\text{NeighborRecall}_i@k = \frac{|A_{i,k} \cap E_{i,k}|}{|E_{i,k}|}.$$

Thus eight shared identities at $k = 10$ score 0.8, while a permutation of all ten scores 1.0. This is set overlap with Flat, not patch relevance or an ordered-rank comparison.

Navigation compatibility. For request a_i, the match indicator is

$$I_i^{\text{nav}} = \mathbf{1}[N(P_{\text{static}}(a_i, M_c)) = N(P_{\text{live}}(a_i, c))].$$

Its mean is the match rate. Providers returning the same normalized set $\{(\texttt{a.py}, 10), (\texttt{b.py}, 30)\}$ match even if character columns, end ranges, or metadata differ; omitting either pair or changing a start line is a mismatch. Conditional latency summaries include only requests with $I_i^{\text{nav}} = 1$.

Maintenance output checks. A graph transition matches its independent rebuild only when the whole-graph and serving checks in Eqs. (1) and (2), together with the changed-file fact projection, all hold. A vector transition must reproduce document identities, numerically equivalent vectors, and exact ordered Flat replay. One missing typed edge, or a replay with ranks 9 and 10 swapped, fails the corresponding check even if an auxiliary F1 or set-overlap score is near one. Its raw measured ratio remains visible, but it is excluded from the matching-transition speedup summary.

Committed-answer localization. Let Y_i be the target source spans and $A_{i,k}$ the first k final-answer spans after parsing structured locations, resolving usable symbol entries, and removing overlapping duplicate predictions. A target is covered when an answer span has the same path and an overlapping inclusive line range:

$$\text{AnswerRecall}_i@k = \frac{1}{|Y_i|} \sum_{y \in Y_i} \mathbf{1}[\exists a \in A_{i,k} : \text{overlap}(a, y)].$$

The reported score macro-averages this quantity over instances. The main result fixes $k = 5$ for reporting and does not limit the number of locations admitted by the answer schema; retrieval and ANN use their separate top-10 contracts. For targets $\{A{:}10\text{-}20, B{:}30\text{-}40, C{:}50\text{-}60\}$, an answer committing to $A{:}15\text{-}18$, $C{:}55\text{-}65$, and $D{:}1\text{-}8$ covers two targets and scores $2/3$. An answer with no usable in-scope committed span scores zero.

Trajectory-token accounting. For m recorded model invocations, let U_t^{in} and U_t^{out} be the provider-reported prompt and completion tokens. The reported trajectory token usage is

$$T_{\text{traj}} = \sum_{t=1}^{m} \left(U_t^{\text{in}} + U_t^{\text{out}} \right). \tag{5}$$

The count m includes any triggered answer-format invocation; it is neither the number of tool calls nor necessarily the 16-turn main-loop count. For an observation o_j produced after invocation j, let $I_{j,t} = 1$ when it is retained in the prompt for invocation t, and let $b_{j,t}$ be its marginal serialized token count there. Its repeated prompt-token contribution is

$$V(o_j) = \sum_{t=j+1}^{m} b_{j,t} I_{j,t}.$$

Only if the observation remains unchanged in every later prompt with marginal size b does this reduce to $b(m - j)$. For $m = 6$, $j = 2$, and $b = 100$, it appears in prompts 3–6 and contributes 400 input tokens. Eviction or a history rewrite makes the corresponding indicators zero; neither can erase usage already recorded through invocation j. The experiment reports the direct provider sum in Eq. (5), not this decomposition, and does not estimate cache hits, prefill work, KV memory, latency, or billing.

C MULTILINGUAL SOURCE UNITS

Levels. The levels are semantic adapter outputs, not fixed AST depths shared by every grammar. L_0 emits at most one signature-only skeleton per processed file; it contains top-level declarations and, where available, member signatures. Files with no recognized declaration produce no L_0 document. L_1 denotes the adapter's named top-level constructs. The evaluated L_2 configuration sets `l2_level_exclusive=true`: it retains top-level leaf declarations and nested members but omits their class, struct, trait, interface, or impl containers. Table 4 lists the concrete five-language mapping. L_1 defines the hierarchy but is not separately materialized in Q1–Q2.

Repository and document filters. Repository traversal uses only the declared language extensions. It excludes the standard VCS/-cache/environment/build directories, files larger than 10 MiB, hidden/binary/backup/minified/bundle basename patterns, and files containing a line longer than 10,000 characters. A test path has a `test`, `tests`, `__tests__`, `spec`, or `specs` directory; a basename starting with `test`; or a `_test`, `_spec`, `.test.`, or `.spec.` marker. Headers, imports, module docstrings, and trailing free text are not emitted as separate L_2 chunks. Q1–Q2 set no line-count split, preserving each recognized logical unit; model token caps may still truncate its encoded text. The production compiler used by lifecycle and incremental-vector experiments instead caps a logical L_2 piece at 300 lines. Split pieces retain the same symbol identity and contiguous ranges.

An L_2 document prepends its repository-relative `path:symbol` identity to the source span and, for class methods, adds a compact enclosing class line. An L_0 document embeds the signature skeleton while retaining path and full-file range in metadata. Source metadata stores path, symbol type, name, node ID, and range alongside each vector.

D PARSING AND SEMANTIC-NAVIGATION BACKENDS

Separation of responsibilities. Tree-sitter supplies deterministic syntax trees and source-unit ranges; it does not resolve imports, types, or cross-file symbols [61]. SCIP is an offline interchange format from which CodeNib builds persistent semantic artifacts [55]. LSP is the live workspace protocol used for the Q3 reference provider and for Q4 repair requests [45]. A file can therefore

Language	Extensions/parser	L_1 recognized constructs	Evaluated exclusive L_2 output
Python	`.py/.pyi/.pyx`; Python	Module functions and classes, including async/decorated forms	Module functions and class methods; class containers are omitted.
Go	`.go`; Go	Functions, structs, named types, interfaces, `var`, and `const`	Functions, receiver methods, `var`, and `const`; type containers are omitted.
Rust	`.rs`; Rust	Functions, structs, enums, traits, impls, constants, statics, and type aliases	Free functions, methods defined in impl blocks, constants, statics, and type aliases; struct/enum/trait/impl containers are omitted.
C/C++	`.c/.h/.cc/ .cpp/.cxx/.hpp`; C++	Function definitions, classes/structs, non-prototype declarations, and macros	Function definitions, in-class method definitions/declarations, non-prototype declarations, and macros; class containers are omitted.
TS/JS	`.ts/.tsx/.js/.jsx`; JavaScript/TypeScript	Functions, classes, initialized variables, object literals, and exported/assigned function forms	Functions and assignments, class methods, initialized variables, and objects; class containers are omitted.

Table 4: Five-language source-unit definitions; JavaScript and TypeScript share one adapter, and `.c` uses the C++ adapter.

remain available to lexical and dense retrieval when its semantic backend is unavailable; the manifest reports the missing graph or exact-position capability instead of treating a syntax-only chunk as resolved semantic data.

SCIP lowering and persistence. For the four SCIP-backed groups, the indexer emits documents keyed by repository-relative path and occurrences carrying a declared position encoding, half-open line/character range, resolved symbol identifier, and role bitfield. CodeNib lowers definition occurrences to source-linked graph vertices and resolved uses to source-anchored relationships, then writes a versioned `graph.pkl`. A separate `lsp_index.pkl` retains every occurrence for native position requests. Its lookup first selects the smallest occurrence containing the zero-based query position, keys global symbols by SCIP identity and local symbols by file plus identity, and returns deduplicated, sorted source locations. Definition roles are selected by the SCIP definition bit; reference queries optionally retain the declaration. The evaluated limits are eight definition and 40 reference locations. If an exact occurrence does not yield a location, the static provider maps the position to the enclosing graph symbol; provider metadata identifies this fallback behavior. C/C++ instead decodes clangd background-index records into the graph/range representation and does not claim a SCIP occurrence path.

Project preparation and capability checks. Cold semantic construction runs against the exact detached checkout. Go uses module metadata, Rust the repository Cargo workspace/toolchain, and TS/JS the detected npm/yarn/pnpm/bun workspace plus a generated or patched `tsconfig`/`jsconfig` with JavaScript enabled. C/C++ requires a nonempty `compile_commands.json`; the builder reuses one when present or attempts CMake and Bear-based generation. Its artifact report records compilation database entries, resolved-path ratio, source coverage, graph coverage, and range-bearing files. Missing tools, failed preparation, or failed quality gates produce an unavailable capability rather than a smaller artifact advertised under the same semantic profile.

Live-LSP replay boundary. Q3 gives the static and live providers the same repository-relative file, zero-based line/character, options, and limit. Each of the 100 snapshots supplies five definition and five reference positions. After JSON-RPC initialization, the runner requires at least two and at most eight warmup rounds until normalized live outputs stabilize; it then measures each request ten times. The idle grace is ten seconds for clangd background indexing and one second for the other servers. Both definition and reference comparisons use deduplicated, path/start-line-sorted location sets. Graph loading, static-provider initialization, live process startup, idle waiting, and warmup are recorded separately from the reported marginal request latency. Tree-sitter chunk ranges and SCIP/LSP positions remain zero-based internally; only exported `CodeLocation` lines cross the one-based user-facing boundary.

E RETRIEVAL MODELS AND FROZEN PARAMETERS

Embedding stack. All five embedders run through Sentence-Transformers with their shipped pooling modules and a Flat inner-product FAISS index. We apply no extra vector normalization beyond modules contained in each model snapshot. Table 6 reports parameter counts obtained by summing the stored safetensor shapes and architectural fields from the recovered cache revisions. Those revisions are post-hoc provenance for historical Q1–Q2 runs, not execution-time locks; the lifecycle run separately pins Qwen3-Embedding-0.6B at execution time.

The SR aliases resolve to `Salesforce/SweRankEmbed-Small` and `fishmingyu/SweRankEmbed-Large`. The artifact ledger records all five Hub IDs plus their recovered commit and tensor hashes. SR-Small's query prefix is "Represent this query for searching relevant code." SR-Large and both Qwen embedders use "Given a github issue, identify the code that needs to be changed to fix the issue" in an instruction/query template; this overrides Qwen's shipped web-search instruction. Jina uses its shipped natural-language-to-code query and document prompts. The document side is unprefixed for both SweRank models, explicitly empty for Qwen, and prefixed with "Candidate code snippet" for Jina.

Pointwise reranking. Qwen3-Reranker-0.6B, 4B, and 8B use Qwen3 causal backbones with respectively 28/16, 36/32, and 36/32 layers/attention heads. The wrapper formats one instruction–query–document prompt, reserves an 8,192-token left-padded input, and scores relevance as $\exp z_{\text{yes}}/(\exp z_{\text{yes}} + \exp z_{\text{no}})$ at the last position. Batch size starts at eight and halves on CUDA out-of-memory; a single still-failing document receives score zero and remains in the recorded run. The instruction is the same GitHub-issue localization task used for the Qwen embeddings. Each curve retrieves $k' \in \{30, 50, 100\}$ dense L_2 candidates, reranks them pointwise, and evaluates the top ten. SR-Small is paired with all three rerankers; Qwen3-Embedding-0.6B, Jina-Code-1.5B, and Qwen3-Embedding-4B are paired with the 4B and 8B rerankers. SR-Large is a dense baseline only. The corresponding Hub IDs are `Qwen/Qwen3-Reranker-0.6B`, with analogously suffixed 4B and 8B variants.

Group	Tree-sitter source-unit adapter	Cold structural artifact	Live JSON-RPC provider	Q3 static position path
Python	Python grammar	`scip-python` → SCIP	`basedpyright-langserver`	SCIP occurrence index, then graph fallback
Go	Go grammar	`scip-go` → SCIP	`gopls`	SCIP occurrence index, then graph fallback
Rust	Rust grammar	`rust-analyzer scip` → SCIP	`rust-analyzer`	SCIP occurrence index, then graph fallback
C/C++	C++ grammar, including `.c`	`clangd` background `.idx` plus compilation database	`clangd`	graph position/range index
TS/JS	Shared adapter over TypeScript and JavaScript grammars	`scip-typescript` → SCIP	`typescript-language-server`	SCIP occurrence index, then graph fallback

Table 5: Evaluated five-language backend routes. Chunking, cold semantic construction, and live serving are selected independently by the language registry; C/C++ is the non-SCIP cold-build route.

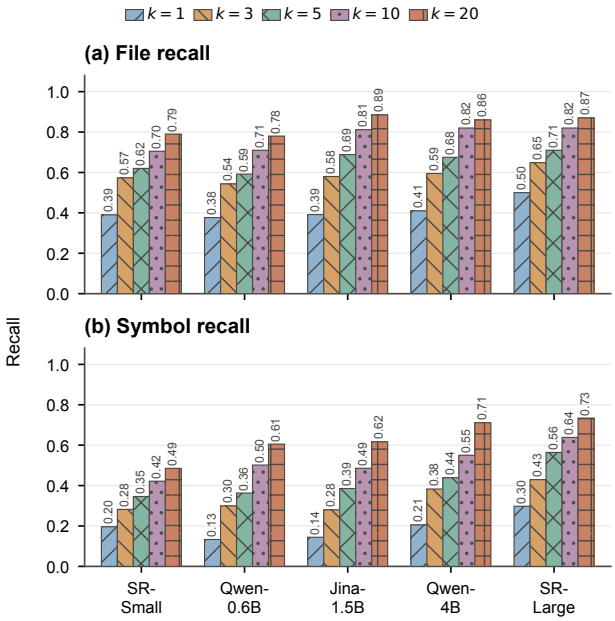


Figure 12: Pure-embedding Recall@k on the 100 CodeNib Base snapshots used by Q1, without graph expansion or reranking. Bar labels are macro means. (a) File results use path deduplication; (b) symbol results use L_2 identities. Figure 6 shows the $k = 10$ slice.

F INCREMENTAL-MAINTENANCE PROTOCOL

Update sequences and execution. The workload selects five first-parent commit transitions from each of eight repositories, with two repositories per language; hence $n_{\text{share}} = 5$. File/symbol arms independently advance the same base graph with one long-lived LSP; fresh targets are independent. In Eq. (3), $T_u^{G,a}$ covers detection, repair, and persistence, while arm-specific $T_s^{G,a}$ covers server start and workspace warmup. Each update is therefore charged $T_s^{G,a}/n_{\text{share}}$. A scheduled transition with no source change does not alter n_{share}, because the setup was incurred for the full sequence. Checkout and base loading are excluded. One resumed, nonmatching Ruff point uses $n_{\text{share}} = 1$ and is outside conditional summaries. Vector fresh/delta arms share a preloaded model; model loading, base construction, checkout, and post-timing comparison are excluded, and their targets remain independent. Graph/vector selectors yield 33/31 source-changing transitions; no-source rows remain outside ratios. Reported protocols are 21/4.

Repair mechanics. Before each batch, the patcher synchronizes Git-modified files into the long-lived server with versioned `didOpen`/`didChange` notifications. It obtains incoming edges with `references`; method locations are retained only when a definition query resolves back to that concrete method, preventing interface–implementation expansion from creating false call edges. Outgoing repair filters semantic tokens to changed ranges and resolves definitions by source position. Repeated semantic tokens at the same `(line, character)` are deduplicated, but equal token text at different positions is not because shadowing and imports can resolve it differently. Latency therefore depends on affected-symbol and reference fan-out, not only changed-line count.

Offline output comparison. After timing, the evaluation compares each update with an independent rebuild. Graph comparison requires exact vertex and typed anchored-edge multisets, both globally and on changed-file facts, plus exact normalized definition/reference replay between the reloaded artifacts. F1 and serving agreement remain diagnostics, not tolerances. Vector comparison requires equal document identities, numerically equal reconstructed vectors, and exact ordered Flat top-10 replay; model revision, dimension, token cap, metric, and chunking levels are compared explicitly. Fresh-target construction and comparison are excluded from maintenance latency. Mismatches retain latency but are excluded from the matching-transition speedup summary.

View	Lang.	n	Exact	Speedup p50 [IQR]	Aux. fidelity
Graph	Go	7	7	8.89 [6.99, 10.01]	E/B 100/100%
Graph	Python	8	8	6.95 [6.22, 13.00]	E/B 100/100%
Graph	Rust	9	0	17.00 [8.97, 40.42]*	E/B 99.12/97.40%
Graph	TS/JS	9	0	1.97 [1.32, 7.59]*	E/B 97.61/99.53%
Vector	Go	7	7	27.61 [22.18, 31.77]	A/R 7/7
Vector	Python	6	6	38.18 [37.42, 38.75]	A/R 6/6
Vector	Rust	9	6	19.43 [14.26, 24.79]	A/R 9/6
Vector	TS/JS	9	9	13.63 [8.92, 25.53]	A/R 9/9

Table 8: Source-changing transitions. Exact counts pass the checks; E/B is median edge F1/serving agreement and A/R artifact/replay exact count. Unstarred speedups use passing rows; starred values are raw ratios when none pass and are excluded from conditional aggregates.

Mismatch audit. Across all graph rows, symbol and file maintenance match 15/33 and 14/33 independent rebuilds, with median

Model	Backbone (layers/heads)	Params	d	Pool/norm	Token cap	Batch	Query/document prefix
SR-Small	NomicBERT (12/12)	137M	768	CLS/none	8,192	$8 \to 4$	shipped `query`/none
Qwen3-Embed-0.6B	Qwen3 (28/16)	596M	1,024	last/ℓ_2	8,192	$8 \to 4$	code-task/empty
Jina-Code-1.5B	Qwen2 (28/12)	1.54B	1,536	last/ℓ_2	8,192	$4 \to 2$	`nl2code_query`/`nl2code_document`
Qwen3-Embed-4B	Qwen3 (36/32)	4.02B	2,560	last/ℓ_2	8,192	$2 \to 1$	code-task/empty
SR-Large	Qwen2 (28/28)	7.07B	3,584	last/none	32,768	$2 \to 1$	shipped `query`/none

Table 6: Dense-index configuration; batch arrows denote failure-only retry.

Operator	Frozen parameters
Dense	Flat IP; L_2; output $k = 10$; model-native query/document prompts.
Graph expansion	Dense pool 300; first 10 chunks seed one-hop incoming and outgoing reference expansion; at most 10 unique neighbors/seed; semantic weight 1; graph weight 0.5; RRF constant 60.
IVF-Flat	$n_{\text{list}} = \min(\text{round}(4\sqrt{N}), \lfloor N/39 \rfloor)$; probe 1, 2, 5, 10, 25, 50, or 100% of lists.
HNSW-Flat	$M = 32$; $ef_{\text{construction}} = 200$; $ef_{\text{search}} \in \{16, 32, 64, 128\}$.
ANN timing	Qwen3-Embedding-0.6B frozen float32 vectors/query; one FAISS CPU thread; search $k = 10$; 10 warmups and 100 timed repetitions per method–snapshot cell.
BM25 lifecycle	Production L_2 chunks; 300-line chunk cap; maximum requested $k = 128$.

Table 7: Retrieval/index parameters; graph weight is development-selected and ANN recall is against exact Flat top 10.

speedups of 8.67× and 1.95× on those transitions. The 14 rows where both paths match yield a 4.25× median file/symbol time ratio. Four independent Rust/TS rebuilds reach 99.75% median edge F1 and 99.35% serving agreement, but one is not exact on the changed scope; strict checks thus expose maintenance or live-provider repeatability rather than tune a tolerance. All 31 vector targets match artifact identities and vectors, while three Rust rows fail exact replay. Protocol 22's later provider-hierarchy fallback is not pooled with the reported protocol-21 campaign.

G LIFECYCLE EXECUTION BOUNDARY

Each materialization uses an isolated, quiescent checkout and fresh output and runtime processes, but does not flush host caches or reload remote model weights; reported construction is therefore warm-host, not machine cold-start. The evaluated vector builder materializes both L_0 and L_2 even though the trace queries only L_2, and lifecycle cost includes both. After fresh-process runtime view loading, replay runs one warmup and three measured repetitions; S is one warm service trace over $N = 20$ sessions. Panel (c) evaluates $B/q + L + S/N$ and $(B + L)/q + S/N$ at $q \in \{1, 5, 10, 20, 50, 100\}$; only $q = 20$ is the controlled trace scale, while the others project the same measured terms. At $q = 20$, process-isolated and shared-resident projections are 13.20 s/session (95% CI 10.43–15.96) and 6.24 s/session (3.66–8.11); at $q = 100$, medians are 8.53 and 1.27 s/session. Runtime view loading floors the former model; the latter approaches measured serve-only S/N, not a hardware or cross-system floor.

H AGENT-CONTEXT PROTOCOL AND VALIDITY DETAILS

Compaction transition. After the complete tool batch containing the first successful `read`, and before the next model invocation, the runner deterministically restores the clean issue, deduplicated paths, newest successful read, and a direction cue copied from the first 600 characters of the latest nonempty assistant message. Only this cue is capped; the issue, retained read, and final answer are not. It removes the candidate dump and prior assistant/tool messages; later messages append normally. Accounting retains all pre-rewrite usage.

Local-model execution. Both Qwen models use vLLM 0.23.0 with a 65,536-token server cap. Gemma 4-12B uses vLLM 0.25.1 with a pinned Hub revision, the versioned Gemma tool template, and a 131,072-token cap. Local runs use 48,000-token history, 4,096-token completion windows, and prefix caching. Gemini 2.5 Flash uses Vertex AI with `thinkingBudget=0`; manifests retain these controls and access date. Compact starts one new prefix at its transition; subsequent calls append to and can reuse that prefix's KV cache. The transition is the policy's single cache discontinuity: we do not claim that the pre-transition cache survives it. Q5 reports provider-reported trajectory tokens, not transition latency or cache-adjusted cost.

Answer-format accounting. When a run terminates without the required answer schema, the runner may issue fixed answer-format invocations after early termination or around the 16-turn cap; their usage is included. Baseline/eager/compact trigger rates are 24.4/16.6/13.2% (9B), 10.8/7.6/3.6% (27B), and zero (Haiku). The corresponding rates are 12.0/1.8/0.6% for Gemma and 3.2/0/0% for Gemini. Invalid answers remain in the quality metric with zero recall.

Direct history-policy contrast. For Haiku/9B/27B, compact/eager token ratios are 123.3% ([113.4, 134.1]), 87.6% ([82.3, 93.0]), and 78.7% ([72.9, 85.4]); paired AnswerRecall@5 changes are +0.018 ([−0.013, +0.051]), −0.023 ([−0.052, +0.005]), and +0.008 ([−0.014, +0.032]). Gemma/Gemini ratios are 27.9% ([23.4, 32.5]) and 76.3% ([63.9, 92.3]), with recall changes of −0.070 ([−0.097, −0.045]) and −0.044 ([−0.074, −0.018]); all intervals are 95% CIs.

Model (arm)	ΔAR@1	ΔAR@3	ΔAR@5	ΔAR@10	$\min_k$ LB$_{.95}$
Haiku (E)	+.005	−.005	−.009	−.007	−.043
Qwen-9B (C)	+.030	+.002	−.006	−.006	−.049
Qwen-27B (C)	+.002	+.000	−.003	+.001	−.037
Gemma (C)	+.076	+.045	+.040	+.040	−.012
Gemini (C)	+.100	+.072	+.067	+.066	+.021

Table 9: Q5 final-answer cutoff sensitivity. At each k, we reapply the -0.05 lower-bound gate and minimum-token rule. Entries are paired mean ΔAnswerRecall@k from grep/read; the last column is the worst lower endpoint among the four snapshot-clustered 95% CIs. Selected arms are invariant (E/C: Eager/Compact).

Validity. Removing the five judge-warning queries does not change which policy meets the localization margin. Haiku backfill

retains the observed model ID and harness, but provider-time drift remains unresolved.

I AUTHOR CONTRIBUTIONS

Zhongming Yu: Conceptualization and Methodology (lead); Software (lead: LSP, retrieval, incremental indexes, agent/web infrastructure, CI/CD, and deployment); Data curation; Investigation; Formal analysis; Validation; Visualization; Writing – original draft; Writing – review and editing; Project administration; Supervision. **Hengjia Yu:** Software (LSP, incremental graphs, and agent infrastructure); Investigation; Validation; Visualization. **Boqin Yuan:** Data curation; Software (dataset, web, and deployment). **Shuting Zhao:** Methodology and Software (retrieval). **Yizhao Chen:** Software (agent infrastructure). **Aryan Dokania:** Software (graph retrieval); Investigation; Formal analysis. **Mihir Jagtap:** Software (incremental vector indexing). **Jiayu Chang:** Software (agent infrastructure). **Yitong Ma:** Visualization; Writing – review. **Yash Jayswal:** Software (web); Writing – review. **Wentao Ni:** Methodology (vector indexes); Writing – review. **Hejia Zhang:** Software (agent infrastructure); Writing – review. **Zhaoling Chen:** Software (agent infrastructure); Writing – review. **Gangda Deng:** Conceptualization; Methodology (graph retrieval, agent methods, benchmarks, and experiments); Writing – review. **Jishen Zhao:** Supervision (PI); Project administration; Writing – review.

J ADDITIONAL DISTRIBUTIONAL VIEWS

Figure 13 visualizes Table 9; Figures 14 and 15 expose distributions compressed in Figures 9 and 11(c).

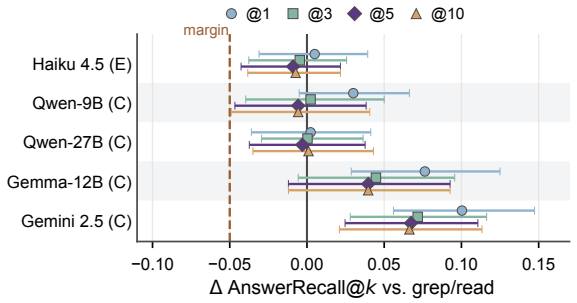


Figure 13: Complete interval view behind Table 9. Points are paired mean ΔAnswerRecall@k from grep/read; whiskers are snapshot-clustered 95% CIs. The Eager/Compact arm named beside each model is invariant across the four cutoffs.

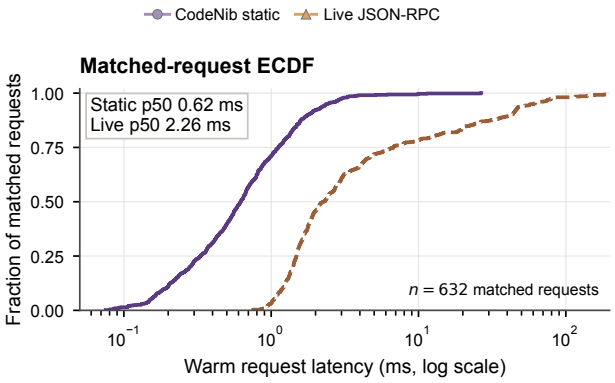


Figure 14: Matched-request latency distribution for Q3. Each observation is one request's median over ten repetitions. The ECDF excludes 368 nonmatching requests, startup, loading, and warmup; it therefore characterizes the conditional compatible subset, not an achieved workload speedup.

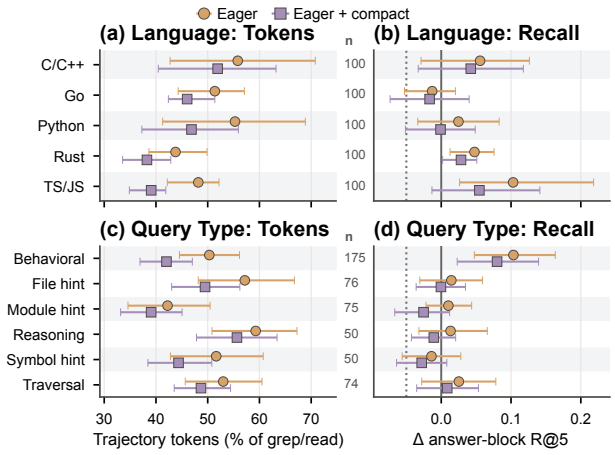


Figure 15: Expanded workload-slice view underlying Figure 11(c). (a–b) Language and (c–d) query-type effects on provider-reported tokens and AnswerRecall@5**. Whiskers are snapshot-clustered 95% CIs. These descriptive intervals do not define a subgroup-routing policy.**