CodeScaleBench: Measuring Code-Retrieval Tooling for Coding Agents on Enterprise-Scale Codebases

Stephanie Jarmak
stephanie.jarmak@sourcegraph.com

July 20, 2026

Abstract

The contribution of external code retrieval to a coding agent is difficult to measure. The configuration that would isolate it (i.e., the same agent and model on the same task with retrieval removed) is rarely run alongside the configuration that includes it. Reported gains therefore confound model capability, task selection, and retrieval quality. We present `CodeScaleBench`, a benchmark that treats retrieval tooling as a controlled variable. The corpus comprises 370 developer tasks, 150 spanning the software-development lifecycle across nine work types and 220 organization-scale tasks across eleven, each pinned to a mirrored real-world repository drawn from more than forty projects and nine programming languages, with org-scale tasks spanning multiple repositories. Every task is evaluated under two matched configurations behind one agent harness: a local-filesystem baseline with standard tools and no retrieval server, and a remote configuration in which the source is withheld and the agent reaches the code through a Model Context Protocol server exposing Sourcegraph code-intelligence tools. Verification is deterministic rather than model-judged, using patch- and test-based scoring for lifecycle tasks and structured-artifact scoring (file-set F_1, symbol-resolution recall, dependency-chain recall, provenance and keyword checks) for organization-scale tasks, with every verifier gated before release by a null/golden/adversarial calibration triad. We report task outcomes jointly with information-retrieval quality. Evaluated with Claude Code on Haiku 4.5, retrieval's effect on task reward is small in aggregate and strongly conditional. The paired mean rises from 0.536 to 0.565, a difference of approximately 0.03 in reward (95 % bootstrap CI 0.009 to 0.056, Wilcoxon $p \approx 0.03$); that effect is carried by the organization-scale discovery tasks, where the paired difference is 0.033 ($p < 0.01$), and is not distinguishable from zero on the lifecycle suites, whose interval spans zero. Retrieval quality separates far more cleanly than reward, raising file recall from 0.127 to 0.277 and precision@5 from 0.140 to 0.478, which establishes that better retrieval is necessary but not sufficient for task success. Retrieval also reduces cost per task by roughly 30 % and agent execution time by roughly 38 %. The reported quantity is a conditional map rather than a uniform lift: retrieval value is set by how reachable the target code is from the agent's local view. This work provides the field with a retrieval benchmark built on matched code-retrieval counterfactuals, deterministic gated verifiers, and a fully specified, reproducible execution and scoring pipeline.

1 Introduction

Benchmarks for coding agents have matured along two largely separate axes. One measures whether a model can resolve a self-contained software task, typically a bug fix verified by a hidden test suite [15, 35]. The other measures whether a retriever ranks the correct code for a query, scored as an information-retrieval problem over a fixed candidate pool [13, 21]. Neither axis answers the question

that a team adopting a code-intelligence tool actually asks: holding the agent and the task fixed, how much does external code retrieval change what the agent produces, and on which kinds of work. Evidence for retrieval-augmented coding agents is most often drawn from a single configuration in which the tool is present, so the measured gain cannot be separated from the choice of model or the choice of tasks.

The separation matters because retrieval does not uniformly help. Yang et al. [34] show that an agent given an undirected search interface can perform worse than one reasoning from the context it already holds. Weng et al. [33] document retrieval turning net-negative under index staleness, injecting obsolete APIs. The repository-completion literature reports that retrieval value concentrates where the answer spans files outside the model's context and is small where the answer is already in view [8, 23, 32, 38]. A benchmark that runs only the retrieval-enabled configuration cannot distinguish these regimes from a uniform improvement.

1.1 Gaps in existing benchmarks

Four properties of prior coding-agent benchmarks motivate this work. First, scope: most cover narrow slices of software work, predominantly bug fixing, rather than the breadth of the development lifecycle from comprehension through documentation. Second, scale: evaluation codebases are small relative to the repositories engineers work in daily, where locating relevant code is itself the difficulty. Third, repository boundary: tasks are almost always single-repository, excluding cross-repository dependency tracing and organization-wide architectural questions. Fourth, measurement: task outcome is reported without the retrieval quality that produced it, so a benchmark cannot say whether an agent succeeded because it retrieved well or despite retrieving poorly. `CodeScaleBench` is designed to address all four. Figure 1 shows a single benchmark run end to end, from suite selection through the agent harness and retrieval configuration to the scored result.

1.2 Thesis

Our claim is narrow and falsifiable: *the effect of code-retrieval tooling on a coding agent is measurable only against a matched counterfactual, it is conditional on task type rather than uniform, and on enterprise-scale multi-repository work it is governed primarily by whether the target code is reachable from the agent's local view.* Retrieval is treated as a controlled variable. The same agent, model, task, oracle, and budget are evaluated with the source available locally and with the source available only through a retrieval server, and the paired difference is the quantity of interest. Because the comparison is paired per task, it separates two regimes that a single configuration conflates: a retrieval *ceiling*, observed on tasks whose answer the baseline holds locally but cannot surface by unindexed traversal within its budget, and a retrieval *lift*, observed where both configurations reach the answer readily and the question is which does so more reliably and at lower cost.

1.3 Research questions

We frame the work as three measurable questions.

- **RQ1.** Holding model and task fixed, does access to a retrieval server change reward, cost (tokens, tool calls, wall-clock), and completion time relative to a local-filesystem baseline, and is the paired difference distinguishable from zero?
- **RQ2.** How does that difference vary across the nine lifecycle and eleven organization-scale work types and across single- versus multi-repository scope, and is the aggregate effect a stable summary or an average over opposite-signed regimes?

- **RQ3.** Does retrieval improve information-retrieval quality (precision, recall, F_1) over the baseline's local search, and to what extent does that improvement translate into task outcome?

1.4 The constraint that governs the design

Properties unique to enterprise-scale evaluation, and those necessary for open reproducibility, govern the benchmark. The repositories under test are large, long-lived, and public, including Kubernetes, Chromium, LLVM, and Django. Public code of this kind is, with high probability, present in the pretraining corpus of every model evaluated [28], so a task whose answer can be recalled from memory measures contamination rather than retrieval. The design responds along two axes. Every repository is pinned to an exact commit and served from a content-controlled mirror, so the baseline and the retrieval configuration observe byte-identical source and the retrieval index is built against the exact revision under test, which also removes the staleness confound Weng et al. [33] identify. The verifier never rewards free-text recall in isolation: a structured artifact carries the score, and a calibration gate rejects any task that a keyword dump can pass. Pinning does not eliminate contamination, which we report as a limitation rather than a solved problem; it ensures that an answer recited without navigation is not scored as a navigated one.

1.5 Contributions

1. **A retrieval benchmark with matched counterfactuals.** 370 tasks across twenty work-type suites, each pinned to a mirrored real repository, with organization-scale tasks spanning multiple repositories, evaluated under matched local-baseline and retrieval configurations behind one harness so that the retrieval channel is the only variable that changes.

2. **Deterministic, gated verification with dual evaluation modes.** A direct mode scores whether an agent correctly changes code, and an artifact mode scores whether an agent retrieves and assembles the correct files, symbols, and dependency chains; both emit a scalar reward in $[0, 1]$, and every verifier is admitted only after passing a null/golden/adversarial calibration triad (Section 5).

3. **Joint reporting of outcome and retrieval quality.** Results aggregate to a work-type-balanced score reported alongside per-task information-retrieval metrics, so that an over-retrieval effect can be distinguished from a genuine precision gain.

4. **A reproducible pipeline and an empirical map of where retrieval helps.**[1] We specify the construction, harness, scoring, and validation procedures in full, and report, from the published Haiku 4.5 results, the conditional effect of retrieval across task type and repository scope, with explicit statement of the regimes where the effect is not distinguishable from zero.

2 Related Work

Code-retrieval research relevant to this work is two-fold: techniques and benchmarks built to rank a function for a query, and techniques and benchmarks built to act inside a repository the agent cannot hold in context. We organize the field by the decision each research area foregrounds (code-IR foundations, repository-scale retrieval, structural indexing, agentic localization, evaluation

[1]The task corpus, harness, verifiers, and frozen result snapshots are released at https://github.com/sourcegraph/CodeScaleBench; the task set is additionally published as a Hugging Face dataset at https://huggingface.co/datasets/sgjarmak/CodeScaleBench.

methodology, and the failure modes that appear at scale), state what each establishes, and then position `CodeScaleBench`. The decision that sets it apart is the counterfactual: rather than scoring one retrieval-on configuration, it runs matched retrieval-on and retrieval-off arms on identical pinned tasks and reads the difference, with deterministic gated verifiers and an explicit account of where the effect is real.

2.1 Code information retrieval and its vocabulary gap

Neural code-IR begins from a measured observation: a natural-language query and the code that answers it rarely share words, so text-IR baselines underperform. CodeSearchNet [13] named this gap and set the shared yardstick; CodeBERT and GraphCodeBERT [9, 12] pretrained joint code-text representations, the latter adding data-flow structure. CoIR [21] consolidates a fragmented landscape into one multi-task, multi-language code-IR suite scored by MRR and nDCG. This line establishes the metrics `CodeScaleBench` reuses for its retrieval-quality view (Section 5), but it evaluates ranking in isolation: a fixed candidate pool, a single query, no downstream task whose success or failure depends on what was retrieved. The benchmark here instead scores the agent's task outcome and treats retrieval quality as a second, jointly reported axis, because a high recall@k that the agent never acts on would otherwise conflate improved retrieval with improved reward.

2.2 Repository-scale retrieval and completion

Once the unit of work is a repository rather than a function, retrieval and generation interleave, inheriting the retrieval-augmented-generation substrate [20]. RepoCoder [36] iterates retrieval and completion; RepoBench [23] and CrossCodeEval [8] benchmark cross-file completion where the needed definition sits in an unopened file; CodeRAG-Bench [32] asks directly whether retrieval augments code generation, and dataflow-guided retrieval [5] sharpens what to fetch. The consistent result is that retrieval value concentrates where the answer spans files outside the model's context window, exactly the regime `CodeScaleBench` isolates as its cross-repository track. These benchmarks measure completion accuracy on held-out spans; they do not vary whether the file is reachable at all, which is the manipulation that separates a retrieval ceiling from a retrieval lift in enterprise multi-repository work.

2.3 Structural indexing beyond embeddings

Embeddings miss rare exact identifiers that a lexical or symbolic index recovers, and a recurring design lesson is to keep a structural lane. Code-graph and code-property-graph approaches (CodexGraph [24], AOCI [22]) answer multi-hop structural questions vector RAG cannot, and the industrial code-intelligence stack draws the same distinction operationally: a trigram text layer (Zoekt [29]) answers "where does this string appear," a symbol layer (Kythe [10], Glean [26], SCIP [30]) answers "where is this symbol defined." The Sourcegraph configuration in `CodeScaleBench` exposes both lanes to the agent (keyword search alongside go-to-definition and find-references over a symbol graph), so the benchmark measures the combined tool surface of a production code-intelligence server rather than an embeddings-only proxy.

2.4 Agentic localization and repository navigation

A separate line treats finding the code as its own component, distinct from changing it. SWE-agent [34] formalizes the agent-computer interface and corroborates a finding from this work: undirected search can degrade performance relative to reasoning from current context. LocAgent [4] performs

graph-guided localization; AutoCodeRover [37], MAGIS [31], LingmaAgent [25], and CodePlan [2] build repository exploration and planning into issue resolution; execution-free plan search [17] and architecture descriptors for navigation [16] front-load a map before exploration. Jain et al. [14] probe LLM agents specifically as semantic code searchers. `CodeScaleBench` sits downstream of all of these: it does not propose a navigation method, it measures what any agent harness gains from a retrieval channel, and its instrumentation records the tool-call sequence so that the undirected-exploration tax Yang et al. [34] describe is visible in the traces rather than hidden in an aggregate.

2.5 Evaluation methodology, contamination, and context value

The reference benchmark for agentic SE is SWE-bench [15] and its language variants [35], which score issue resolution by hidden test pass/fail. Two methodological concerns from this benchmark heritage inform the design of our work. First, contamination: from HumanEval [3] onward, public evaluation sets enter pretraining data, and Riddell et al. [28] quantify how much that memorization inflates measured capability, which is why `CodeScaleBench` pins exact revisions, serves byte-identical source to both arms, and refuses to let free-text recall carry a score. Second, the value of context to agents is itself an active measurement target: Dillon and Varanasi [7] report that supplying product context improves an agent's decision compliance, and the developer-survey literature [6] and library-evolution benchmarks [19] document how much real work depends on context outside the current file. `CodeScaleBench` differs from SWE-bench in three ways that follow from its question: verification is deterministic partial-credit rather than hidden-test pass/fail, the task corpus spans nine work types rather than bug-fix alone, and the headline is a paired retrieval-on-versus-off difference rather than a single resolution rate.

2.6 Failure modes at scale

The enterprise setting surfaces failures the single-file demo never shows. Retrieval can hurt: Weng et al. [33] diagnose staleness as the trigger, with a lagging index injecting obsolete APIs. Search can backfire: Yang et al. [34] show iterative search degrading performance. Metrics can mislead: text-overlap scores such as BLEU punish semantically correct code, which is why code-aware measures [27] and late-interaction reranking [18] exist. `CodeScaleBench` is built around these as first-class possibilities. Pinning removes the staleness confound by construction; the traces expose when an agent searches without converging; and the verifier scores structured artifacts against a calibrated oracle rather than text overlap, so a retrieval arm that injects plausible-but-wrong context is scored down rather than rewarded for fluency.

2.7 Positioning

`CodeScaleBench` shares with the repository-scale and agentic-SE benchmarks the use of real codebases, downstream task scoring, and a concern for retrieval that spans files outside context. Three points are differentiated. The corpus holds the agent and task fixed and varies only the retrieval channel, so the reported quantity is a paired counterfactual rather than a single configuration's score. The verification is deterministic and gated: file-set F_1, symbol recall, and dependency-chain recall, each admitted only after passing a null/golden/adversarial calibration triad, with an anti-adversarial gate against keyword-dump answers. And the effect is reported as a conditional map across task type and repository scope, separating a retrieval ceiling from a retrieval lift, rather than a uniform headline. We are direct about the limitations of our current design constraints: contamination cannot be fully excluded by pinning, the aggregate effect is small and averages over suites with different effects, and the published results use a single model and harness.

3 Benchmark Construction

A task in `CodeScaleBench` is a self-contained directory holding a metadata file, a natural-language instruction, one or more container definitions, and a verifier. The evaluated corpus comprises 370 tasks partitioned into two families. The lifecycle family (`CodeScaleBench-SDLC`, 150 tasks) covers nine phases of software work: comprehension and onboarding, design, feature implementation, bug fixing, testing and review, documentation, refactoring, security, and debugging. The organization-scale family (`CodeScaleBench-Org`, 220 tasks) covers eleven categories of cross-cutting work: dependency tracing, vulnerability remediation, framework migration, incident debugging, onboarding and comprehension, compliance, cross-organization discovery, domain lineage, organizational context, platform knowledge, and cross-repository discovery. Tasks span more than forty repositories across nine programming languages, and the organization-scale tasks are the multi-repository core of the benchmark.

3.1 Corpus versions

Two corpus versions exist as the benchmark continues development beyond the snapshot analysis presented. The published results use a frozen suite, `csb-v1-mixed371`, that defines 372 tasks of mixed verifier type (152 lifecycle and 220 organization-scale); of these, 370 tasks (150 lifecycle and 220 organization-scale) have valid paired baseline-and-retrieval results and constitute the analysis set, with the two excluded tasks lacking a usable pair. The benchmark was subsequently re-curated into a maintained set with stricter dual-verifier support: `csb-v2-full-validated` (275 tasks) and the canonical `csb-v2-dual264` (264 tasks supporting both direct and artifact scoring). The maintained set is not a subset of the frozen one; it retains 251 of the version-one tasks, adds 24, and drops 121 single- or mixed-verifier tasks. A current development checkout therefore provides the maintained corpus, while the numbers reported in Section 6 are on the frozen version-one suite; we cite suite identifiers throughout so that a result names the exact task set it was measured on. The frozen `csb-v1-mixed371` suite and its result snapshot are retained under a tagged public release, so a reproduction runs against the exact 370-task analysis set behind the reported numbers rather than against the re-curated development corpus.

Figure 2 summarizes the authoring and admission pipeline: templates and a use-case registry generate task skeletons, ground truth is curated and validated, and a preflight check and a calibration triad gate each task before it enters the corpus.

3.2 Task sourcing and grounding

Each task is grounded in a real repository at a pinned commit and targets a development scenario drawn from GitHub issues, pull requests, and codebase analysis. Lifecycle tasks combine original authoring with patterns adapted from existing software-engineering datasets. Organization-scale tasks are generated from a use-case registry that records, per use case, a customer-framed prompt, a task family, the target work-type suite, a repository-set identifier, an a-priori difficulty, the retrieval capabilities the task is expected to exercise, and whether the answer is reachable without retrieval. The registry is the human design surface; a generator stamps each record into a task skeleton, so the structure is mechanical while the prompt, difficulty, repository choice, and retrieval-reachability annotation are authored decisions that encode the benchmark's hypothesis about each task before any agent runs.

3.3 Repositories are mirrored and pinned

Tasks run against open-source projects chosen for scale and realism, grouped into repository-set fixtures that declare which repositories a baseline agent receives locally and which exist only behind retrieval. Each repository is mirrored under a controlled namespace and pinned to an exact commit. Pinning serves three purposes that recur in the literature: the retrieval index is built against the exact revision under test; the baseline and retrieval configurations observe byte-identical source, so the comparison is paired; and a pinned revision removes the index-staleness failure that turns retrieval net-negative [33]. Container base images are themselves pinned by content digest and validated against a recorded manifest, so a rebuild cannot silently alter the environment beneath an agent.

3.4 Oracle construction

The ground truth for an organization-scale task is the set of files, symbols, and dependency chains a correct answer must identify. This oracle is produced by a curator agent (Claude Code with Opus-class generation) and calibrated against ContextBench human annotations to a quality threshold of $F_1 > 0.65$ before acceptance. To guard against the oracle being an artifact of a single search query, generation issues several semantically diverse queries, each pinned to the fixture revision; a file enters the oracle only if it appears in at least K of N queries, line matches are classified into definition, usage, comment, or string contexts so that only code-context hits count as required, and files are annotated as *required* (defines the concept) or *sufficient* (references it) to weight the F_1. Oracle coherence is then verified against the repository: every oracle file must exist, every (file, symbol) pair must resolve through structural parsing, and a leakage pass confirms that answer tokens do not appear in README or test files the agent would read, so a task cannot be solved by reading a giveaway. For lifecycle bug-fix tasks the oracle is the human-written patch from the originating pull request.

3.5 Difficulty: an authored label and a computed index

Two difficulty signals are maintained separately because they disagree, and reporting both is informative. The registry carries an authored difficulty, predominantly *hard*, reflecting the judgment that locating relevant code in a multi-gigabyte repository is difficult even when the required edit is small. A separate complexity index is computed from observable features (repository size, repository count, ground-truth file count, total file count, and authored difficulty, weighted 0.30/0.20/0.20/0.15/0.15) and classifies most tasks as medium. The authored label expresses intent; the computed index expresses measurable structure. Their disagreement characterizes the corpus precisely: structurally moderate, navigation-hard.

3.6 Suite selection by experimental design

Tasks are assembled into versioned suites that fix exactly which tasks constitute a benchmark run, so that a result cites a frozen task set rather than a mutable directory. Suite sizes are determined by a design-of-experiments procedure with Neyman allocation, which sets per-suite sample sizes from observed variance to balance statistical power across suites; initial runs used twenty tasks per suite and were adjusted upward where variance analysis required it. Orthogonal indexes view the corpus by computed complexity, by language, by repository-size bucket, and by multi-repository scope, supporting stratified analysis of the conditional effect that is the central result.

3.7 Scale of construction

The corpus was assembled over approximately one month across a multi-agent authoring and quality-assurance effort spanning several agent harnesses, comprising on the order of 2387 agent sessions. A substantial fraction of that effort was quality assurance and verifier auditing rather than initial authoring, which is consistent with the finding that verifier correctness, not task generation, is the binding constraint on a benchmark of this kind.

4 Execution Harness and Configurations

The validity of a paired counterfactual depends on the machinery that holds every factor constant except retrieval. A run proceeds from a typed configuration through a launcher into an isolated container in which a single agent harness executes the task; the harness, model, instruction, and budget are identical across configurations, and the configuration alters only what source the agent can see and through which channel. This section specifies that machinery and states which components are original to this work and which are integrated dependencies, because the contribution is the controlled comparison rather than a new agent runtime.

4.1 Two configurations isolate the variable

The published evaluation compares two matched configurations defined in a single canonical map from configuration name to environment. The baseline (`baseline-local-direct`) provides the full source on the agent's local filesystem together with standard tools (grep, find, file reading) and no retrieval server. The retrieval configuration (`mcp-remote-direct`) withholds the local source and requires the agent to reach the code through a Model Context Protocol server [1] exposing thirteen Sourcegraph code-intelligence tools: keyword and natural-language search, file reading and listing, repository listing, go-to-definition, find-references, commit and diff search, revision comparison, and a deep-search subagent. This surface combines the lexical and symbolic lanes that the structural-indexing literature argues for [24, 30], rather than an embeddings-only proxy. The harness also supports additional retrieval backends behind the same interface for internal comparison; the results reported here are confined to the baseline and the Sourcegraph configuration, which constitute the published study.

4.2 Execution infrastructure

Figure 3 shows the execution boundary: the configurator and launcher inside the harness, and the retrieval backends reached as external services. Tasks are orchestrated by Harbor, which isolates each task in a Docker container, installs the agent, runs it, and collects results, and are executed in parallel through Daytona, with on the order of twelve concurrent tasks locally and substantially more on remote sandboxes. Original to this work are the configuration-to-environment map, the retrieval-server configurator that writes the agent's server configuration and removes it for the baseline, a guarded sandbox wrapper that labels and retries sandboxes and enforces a cost preflight, multi-account credential and capacity management for parallel runs, and a post-run validation and quarantine stage. Integrated as dependencies are the Harbor task runner, the Daytona sandbox service, the agent CLI, and the retrieval servers. The published results use Claude Code on Haiku 4.5 as the agent and model.

4.3 Forcing the variable to bind

A retrieval configuration in which the agent silently falls back to local file tools would not test retrieval. The harness prevents this in two ways: the retrieval configuration removes the local source, so no fallback target exists, and the agent's local discovery tools are disabled in retrieval mode, so code discovery must pass through the server. The instruction is augmented with the retrieval tool surface, because an available tool is not a used tool unless the prompt names it; this is a documented prerequisite rather than an optimization. Every tool call is recorded, so whether the agent navigated through the server, and how efficiently, is observable after the fact rather than assumed.

4.4 Verification under withheld source

Organization-scale tasks in the retrieval configuration run against a withheld working tree, which the verifier must nonetheless score against repository state. At verification time a wrapper restores the full repository so that the verifier observes the same ground truth in both configurations. This separates the agent's information environment (withheld source, retrieval only) from the verifier's information environment (full source), so that retrieval is constrained during the task without weakening the check applied to the result.

4.5 Result contract

Every task emits a fixed output tree: the runner's result record with task metadata, pass/fail status, reward, and timing; a metrics record with token counts, cost, and tool usage; a verifier record carrying the schema-versioned scalar reward; and the agent's turn-by-turn trajectory and full transcript. This contract is what allows a run to be re-scored, audited, and compared across configurations without re-execution, and is the basis for the published result snapshots. A post-run stage reads the tree and quarantines runs that are infrastructure failures rather than agent failures, the most common being a rate-limited trial returning reward zero in under thirty seconds; such trials are set aside rather than deleted, and a gap scanner schedules reruns, so that transient rate limiting does not contaminate a configuration's mean.

4.6 Reproducing a run

A run is reproducible from three inputs: a frozen suite definition naming the task set, a typed run configuration naming the agent, model, and retrieval configuration, and the pinned repository mirrors and digest-pinned base images. The launcher resolves the suite, materializes each task's container, executes the agent under the named configuration, and writes the result tree to a content-addressed run directory. Because the suite, the repository revisions, and the base images are all pinned, and the verifier code is versioned and synchronized across task copies by an integrity gate, an independent party can re-execute the same suite under the same configuration and obtain comparable results, or substitute a different agent or model behind the same harness to extend the study. The published artifact includes frozen result snapshots with per-task traces, scores, timing, and cost, so that the reported numbers can be audited without re-execution.

5 Scoring and Validation

A retrieval benchmark depends on its verifier, because a verifier that a fluent but incorrect answer can pass will credit retrieval for confident hallucination. `CodeScaleBench` scores deterministically, gates every verifier against synthetic answers before release, and keeps any model-judged signal

separate from the reward. Every verifier emits a scalar reward in $[0, 1]$. Figure 4 shows the scoring and metrics flow from the run-result tree to the work-type-balanced aggregate.

5.1 Two evaluation modes

The benchmark scores two distinct claims. *Direct mode* tests whether an agent correctly changes code, and applies task-specific patterns implemented in a per-task test script: checklist composites for build and feature work, test-ratio scoring for fix tasks, rubric scoring for fault localization, and a hybrid of file-set F_1 and fix quality for review tasks. *Artifact mode* tests whether an agent retrieves and assembles the correct files, symbols, and dependency chains, and applies a deterministic oracle check: file-set F_1, symbol-resolution recall, dependency-chain recall, provenance and keyword presence, and schema validation. Organization-scale tasks are scored primarily in artifact mode, lifecycle tasks primarily in direct mode, and a subset supports both, which allows the same task to be read either as a code change or as a retrieval problem.

5.2 Deterministic partial-credit scoring

In artifact mode the agent's structured answer carries the score. File-set F_1 is computed against the oracle with three-pass repository-name normalization: matching first on exact (repo, path), then after stripping the mirror identifier and organization prefix, then on path alone, so that an answer naming the upstream repository still matches a mirror-named oracle. Symbol resolution is recall over (repo, path, symbol) triples; dependency-chain scoring is ordered recall over the required sequence; provenance and keyword checks are substring matches into the rationale. The composite is the mean of the per-check primary scores. One gate is load-bearing: a text-only check scores zero when the structured answer is empty, so a rationale that names every file in prose while leaving the file array blank earns nothing. This mechanism prevents fluency from substituting for navigation, and operationalizes the methodological point that answer quality alone cannot validate a retrieval system.

5.3 The calibration triad gates the verifier

Before a task is released, its verifier is run against three synthetic answers, which tests the verifier rather than any agent. A *null* answer with empty structured content must score at most 0.1; a *golden* answer derived from the oracle must score at least 0.9; an *adversarial* answer that dumps every repository name, path, and symbol into free text with empty structured arrays must score at most 0.5. A task failing any gate is held back until its verifier is corrected. The three gates detect three distinct defects: a null above floor indicates an oracle that is too permissive, a golden below ceiling indicates a mis-specified verifier or oracle, and an adversarial above floor indicates gameable matching. The distinction the triad enforces, that a synthesized perfect answer is a probe for the verifier rather than the key against which the agent is graded, is essential and easy to misapply.

5.4 Aggregation balanced by work type

Per-task rewards aggregate to a single score balanced across work types: within each work type a pass rate is computed against a threshold, and the score is the mean of the per-type pass rates scaled to one hundred. The balancing is deliberate. A sample-weighted mean would allow the largest category to determine the headline; a hundred failed fix tasks beside one passing test task would read as approximately one percent under sample weighting and as fifty under type balancing. Because the central result is a conditional effect across work types, the aggregate must not be a

quantity that a single dominant category can move. The balanced score is reported alongside the per-type breakdown, never in isolation.

5.5 Retrieval quality, separately measured

Outcome reward indicates whether the agent succeeded; it does not indicate whether retrieval was precise, and a retrieval configuration can raise reward by over-fetching. Retrieval quality is therefore reported on its own axis, computed from the agent's file accesses against the oracle: precision@k, recall@k, mean reciprocal rank, normalized discounted cumulative gain, mean average precision, and a context-efficiency ratio, with paths normalized through the same repository-name passes as the verifier. Reporting retrieval quality beside reward distinguishes an over-injection effect from a genuine precision gain, the failure mode an outcome-only benchmark cannot detect.

5.6 An optional judge kept off the reward path

An optional model-judged score [11] is computed for tasks where free-text quality matters, but it is stored in a separate field and never enters the reward path, so the headline never depends on one model grading another.

5.7 A multi-dimension quality-assurance pipeline

Verifier correctness is enforced by a pipeline spanning six dimensions: task validity (the task is well-formed and answerable), outcome validity (the reward reflects the work performed), reporting (results are complete and schema-conformant), reproducibility (re-execution yields comparable scores), tool effectiveness (the retrieval configuration actually exercises retrieval), and statistical validity (sample sizes support the claims drawn). The pipeline runs as preflight checks before a benchmark run, as the calibration triad at task admission, and as post-run validation. Two safeguards in this pipeline derive directly from observed failures during construction. A determinism check rejects any verifier whose score varies across repeated execution on identical input. A container-parity and broken-environment check compares the baseline and retrieval images for the same task and drops any baseline trial whose transcript reports an empty or misconfigured workspace at low reward, because a baseline that fails for an infrastructure reason would otherwise inflate the measured retrieval effect. Both safeguards exist because an early uncorrected baseline-environment defect produced a spurious retrieval advantage that disappeared once the control was repaired. The pipeline treats environment failure as a first-class threat to validity. The drop rule keys on an empty workspace at low reward, and a post-hoc audit of the frozen snapshot found a small number of organization-scale tasks that slipped it: their baseline shipped with an empty local checkout yet scored non-trivially by recalling well-known public code rather than navigating it, above the reward floor the rule triggers on. Excluding them shifts the organization-scale paired difference from 0.033 to 0.031 and leaves the conditional result unchanged, and the underlying tasks are corrected in the maintained corpus.

6 Evaluation

We report the published study: 370 tasks evaluated with Claude Code on Haiku 4.5 under the matched `baseline-local-direct` and `mcp-remote-direct` configurations. Across the frozen result snapshot the mean reward rises from 0.536 at baseline to 0.565 with retrieval, a paired difference of approximately 0.03 in reward whose 95 % bootstrap confidence interval, 0.009 to 0.056 with

Suite	n	Relative reward gain
Comprehension (understand)	10	+11.5%
Incident debugging	20	+11.2%
Security	24	+10.6%
Refactoring	16	+10.3%
Fix	26	+9.9%

Table 1: Per-suite reward gain for the retrieval configuration over the local baseline, as a fraction of baseline reward, Claude Code on Haiku 4.5. Gains concentrate on discovery-heavy and cross-repository work. At these sample sizes the paired confidence intervals for individual suites mostly include zero; the figures are reported as directional, and the tested claim is the family-level effect in the text, not any single row. Other lifecycle suites show smaller or opposite-signed differences and are omitted.

a Wilcoxon signed-rank $p \approx 0.03$, excludes zero but is small. That aggregate averages over task families with opposite-signed effects, so it is a summary of convenience rather than the result. Split by family, the organization-scale tasks carry the effect (paired difference 0.033, CI 0.013 to 0.055, $p < 0.01$), while the lifecycle suites are jointly not distinguishable from zero (paired difference 0.029, CI -0.015 to 0.076). The contribution of retrieval is therefore best characterized by where it concentrates.

6.1 Retrieval improves discovery-heavy and cross-repository work

Disaggregated by suite, the reward gains concentrate on the work types where locating relevant code dominates the task (Table 1), and the pattern follows the retrieval-lift reading rather than a ceiling artifact: the gains are largest on comprehension, incident debugging, security, refactoring, and fix work, all cases in which both configurations can reach the same repository content, and the retrieval arm does so with less enumeration. The entries in Table 1 are gains relative to baseline reward, not percentage points, so the 11.5 % comprehension entry is a mean reward of 0.692 times 1.115. These per-suite figures are directional rather than individually conclusive: at 10 to 28 tasks per suite the paired confidence intervals mostly include zero, so the claim that survives a paired test is the family-level organization-scale effect reported above, not any single suite in isolation.

6.2 Retrieval quality improves more than outcome

The information-retrieval view shows a larger and cleaner separation than reward (Table 2). Over the full corpus, retrieval raises file recall from 0.127 to 0.277, precision@5 from 0.140 to 0.478, and F_1@5 from 0.099 to 0.262. On the organization-scale family the contrast is starker: precision@5 rises from 0.007 to 0.471 and recall@5 from 0.002 to 0.149. The baseline's near-zero precision here is not an information gap but a navigation gap. Both configurations operate over the same repository content at the same pinned commit: the baseline holds a full local checkout and searches it with unindexed `grep` and `find`, while the retrieval arm reaches the identical content through the retrieval server's index. The organization-scale targets sit in multi-gigabyte trees, and within a fixed turn and token budget the baseline's unindexed enumeration rarely converges on the target files, whereas the index reaches them directly. The measured effect on this family is therefore one of effective reachability under a budget, not of one arm being denied information the other holds. The gap between a large retrieval-quality gain and a smaller reward gain is itself informative: improved retrieval does not translate one-to-one into improved outcomes, because an agent may retrieve

Metric	Baseline	MCP
File recall (full corpus)	0.127	0.277
Precision@5 (full corpus)	0.140	0.478
F_1@5 (full corpus)	0.099	0.262
Precision@5 (org-scale)	0.007	0.471
Recall@5 (org-scale)	0.002	0.149

Table 2: Information-retrieval quality, baseline versus retrieval, Haiku 4.5. The organization-scale rows isolate the regime in which both configurations reach the same content, the baseline through a local checkout and the retrieval arm through the server's index, yet the baseline's unindexed traversal of multi-gigabyte trees rarely surfaces the target within the agent's budget.

the correct files and still fail the task, or succeed without retrieving cleanly. Retrieval quality is necessary but not sufficient for task success.

6.3 Retrieval reduces cost and time

Retrieval reduces cost per task by roughly thirty percent, from approximately $0.73 to $0.51, reduces wall-clock time by about 47 seconds, and reduces agent execution time by about 90 seconds, a reduction of roughly thirty-eight percent. The mechanism is that targeted search replaces enumerate-then-read traversal of a large tree. A single organization-scale Kubernetes task illustrates the extreme of this effect: the baseline reached a near-two-hour timeout, while the retrieval configuration completed in 89 seconds with a reward of 0.90.

6.4 Agents under-use semantic retrieval

Tool-usage instrumentation shows that agents default to lexical search unless directed otherwise. Across the retrieval runs there were 4813 keyword-search calls and 6324 file-read calls, against only 587 natural-language-search calls, the latter used in roughly forty-two percent of tasks; the deep-search facility was invoked on six tasks, with eight calls across 602 retrieval runs. The available semantic and graph-navigation capabilities are therefore substantially under-exercised relative to lexical search, which bounds the measured retrieval effect from below: the observed gains were obtained despite agents using the richer tools sparingly.

6.5 Threats to validity

Five threats bound the results. Contamination: the repositories are public and pinning does not remove them from pretraining, so an unknown share of single-repository success may reflect recall rather than navigation; the structured-answer requirement and the anti-adversarial gate mitigate but do not eliminate this. Statistical power: the evidence is directional, most per-suite confidence intervals include zero at these sample sizes, and stronger claims require additional runs and more balanced task distributions. Single model and harness: the published results use one model (Haiku 4.5) and one agent harness (Claude Code), so generalization across models and harnesses is not yet established. Oracle circularity: the organization-scale ground truth is generated by a curator agent of the same model class as one evaluated configuration and admitted at an F_1 above 0.65 against human annotation, so the oracle disagrees with human labels on a substantial fraction of items, the retrieval-quality denominator is itself noisy, and a prior shared between the oracle-maker and the evaluated agent could favor that model family in ways this design cannot rule out; human

relabeling of verifier outputs, deferred to future work, is the direct check. Conflict of interest: `CodeScaleBench` is authored at Sourcegraph, and the retrieval configuration exposes a Sourcegraph code-intelligence server. The design controls for this through information parity: both arms reach the same repository content at the same pinned commit, so the contrast is indexed retrieval against a baseline that navigates its local checkout with unindexed `grep` and `find`, not retrieval against no access. That baseline is the default local tool surface of a coding agent, so the cost, time, and retrieval-quality gains are a valid measurement of what an index adds over unaided local search under a fixed budget. What is configuration-dependent is their magnitude: a baseline given local indexing (ctags, a language server, or a warm code index) would likely narrow the retrieval-quality and cost gaps, and measuring that is future work. The reward lift on organization-scale discovery does not depend on how the baseline is tooled, since it persists with the target bytes already on the baseline's local disk, and it is the claim we anchor on.

7 Future Work

The published results are directional evidence rather than a settled verdict, and the agenda follows from the threats to validity. The aggregate effect is small while the conditional effect is large, so the priority is to make the conditional claim powered, model- and harness-general, contamination-resistant, and validated against human judgment of its own oracles.

Multiple harnesses and models. The reported study uses one harness and one model. Evaluating the same task suites across additional agent harnesses, including Codex, Cursor, Gemini, Copilot, and OpenHands, and across additional models, would establish whether the conditional retrieval effect is a property of the tasks or of a particular agent. The harness already supports configuration-level substitution of agent and model, so this extension is procedural rather than architectural.

Powered, pre-registered comparison. A single-model comparison at sample sizes chosen by power analysis per work type, pre-registered so that the conditional claim is confirmatory rather than exploratory, would convert per-suite point estimates whose intervals currently include zero into confirmed effects, and would balance the task distribution so that the aggregate is not an average over unequally powered suites.

Human validation of verifiers. The calibration triad tests verifiers against synthetic answers, not against real agent outputs that a human has judged correct. Sampling agent answers, labeling them by hand, and measuring verifier precision and recall against those labels would quantify the false-negative rate that deterministic scoring pays for its auditability, and would address the strongest objection a reviewer can raise.

Contamination-resistant tasks. Pinning addresses reproducibility, not memorization. A held-out split mined from private or recent repository history, in the spirit of contamination-resistant probing [28], would allow the single-repository regime to be re-measured on code the models have not seen, separating navigation from recall on exactly the tasks where the two are currently confounded.

Staleness and permissions as task variables. Two failure modes the literature identifies are not yet manipulated here. Deliberately serving an index that lags the working tree would turn the staleness diagnostic of Weng et al. [33] into a benchmark axis, measuring how gracefully each

backend declines stale context. Scoping retrieval to what a caller is permitted to read would test whether find-references respects access boundaries, the least-studied corner of enterprise retrieval.

Closing the analysis loop. The trace-level instruments, namely the tool-call error analysis and the behavioral failure taxonomy, are not yet fed back into task selection. Using the failure taxonomy to target where retrieval breaks down would let the corpus evolve toward the tasks that carry signal rather than growing by accretion. The finding that agents under-use semantic and graph-navigation tools also suggests a direct line of work on instruction and interface design to elicit the retrieval behavior the tools were built to support.

8 Conclusion

`CodeScaleBench` addresses a question that prior coding-agent benchmarks leave unanswered: holding the agent and the task fixed, how much does external code retrieval change what the agent produces, and on which kinds of work. By evaluating 370 tasks under matched local-baseline and retrieval configurations behind one harness, with deterministic verifiers gated by a null/golden/adversarial calibration triad and outcomes reported jointly with retrieval quality, the benchmark measures retrieval as a controlled variable rather than crediting it from a single configuration.

The published results, obtained with Claude Code on Haiku 4.5, show that the effect is real, conditional, and uneven. The reward gain is small in aggregate and concentrated: it is carried by the organization-scale discovery work, where the paired difference of about 0.03 is distinguishable from zero, and is jointly not distinguishable from zero on the lifecycle suites, while retrieval quality improves substantially and cost and execution time drop. Retrieval quality improves more than outcome, which establishes that better retrieval is necessary but not sufficient for task success, and agents under-use the semantic and graph-navigation tools available to them, which means the measured gains are a lower bound on what the tools can support.

The benchmark's larger contribution is the methodology and the artifact: a fully specified construction, execution, scoring, and validation pipeline, with pinned repositories, digest-pinned environments, dual evaluation modes, gated verifiers, and frozen result snapshots, such that an independent party can reproduce the published study, substitute a different agent or model, or extend the corpus. The open question is whether the conditional map, once measured across additional models and harnesses, powered to adequate sample sizes, and validated against human judgment of its oracles, holds the pattern the current evidence indicates: retrieval as a navigation capability whose value is set by reachability, not a uniform improvement to be assumed.

References

[1] Anthropic. Model Context Protocol. https://modelcontextprotocol.io, 2024.

[2] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level Coding using LLMs and Planning. *arXiv preprint*, 2023.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias

Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *arXiv preprint*, 2021.

[4] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-Guided LLM Agents for Code Localization. *arXiv preprint*, 2025.

[5] Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion. *arXiv preprint*, 2024.

[6] Rudrajit Choudhuri, Christian Bird, Carmen Badea, and Anita Sarma. To Copilot and Beyond: 22 AI Systems Developers Want Built. *arXiv preprint*, 2026.

[7] Drew Dillon and Kasyap Varanasi. Context-Augmented Code Generation: How Product Context Improves AI Coding Agent Decision Compliance by 49 *arXiv preprint*, 2026.

[8] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. *arXiv preprint*, 2023.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint*, 2020.

[10] Google. Kythe: A Pluggable, Language-Agnostic Code Indexing Service. 2016.

[11] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. A Survey on LLM-as-a-Judge. *arXiv preprint*, 2024.

[12] 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 Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv preprint*, 2020.

[13] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint*, 2019.

[14] Sarthak Jain, Aditya Dora, Ka Seng Sam, and Prabhat Singh. LLM Agents Improve Semantic Code Search. *arXiv preprint*, 2024.

[15] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint*, 2023.

[16] Ruoqi Jin. Formal Architecture Descriptors as Navigation Primitives for AI Coding Agents. *arXiv preprint*, 2026.

[17] Zaid Khan, Ali Farhadi, Ranjay Krishna, Luca Weihs, Mohit Bansal, and Tanmay Gupta. MutaGReP: Execution-Free Repository-Grounded Plan Search for Code-Use. *arXiv preprint*, 2025.

[18] Omar Khattab and Matei Zaharia. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. *arXiv preprint*, 2020.

[19] Sachit Kuhar, Wasi Uddin Ahmad, Zijian Wang, Nihal Jain, Haifeng Qian, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, and Anoop Deoras. LibEvolutionEval: A Benchmark and Study for Version-Specific Code Generation. *arXiv preprint*, 2024.

[20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *arXiv preprint*, 2020.

[21] Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. CoIR: A Comprehensive Benchmark for Code Information Retrieval Models. *arXiv preprint*, 2024.

[22] Jinshi Liu, Hanying Zuo, Congyin Cao, Anran Zhang, Yixuan Liu, and Xinzhou Xie. AOCI: Symbolic-Semantic Indexing for Practical Repository-Scale Code Understanding with LLMs. *arXiv preprint*, 2026.

[23] Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *arXiv preprint*, 2023.

[24] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. *arXiv preprint*, 2024.

[25] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. Alibaba LingmaAgent: Improving Automated Issue Resolution via Comprehensive Repository Exploration. *arXiv preprint*, 2024.

[26] Meta. Glean: System for Collecting, Deriving and Working with Facts about Source Code. 2021.

[27] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint*, 2020.

[28] Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models. *arXiv preprint*, 2024.

[29] Sourcegraph. Zoekt: Fast Trigram-Based Code Search. https://github.com/sourcegraph/zoekt, 2016.

[30] Sourcegraph. SCIP: A Code Intelligence Protocol. https://sourcegraph.com/blog/announcing-scip, 2023.

[31] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. *arXiv preprint*, 2024.

[32] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. CodeRAG-Bench: Can Retrieval Augment Code Generation? *arXiv preprint*, 2024.

[33] Haojun Weng, Qianqian Yang, Hao Fu, Haobin Pan, and Xinwei Lv. When Retrieval Hurts Code Completion: A Diagnostic Study of Stale Repository Context. *arXiv preprint*, 2026.

[34] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *arXiv preprint*, 2024.

[35] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. *arXiv preprint*, 2024.

[36] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. *arXiv preprint*, 2023.

[37] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous Program Improvement. *arXiv preprint*, 2024.

[38] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. *arXiv preprint*, 2023.

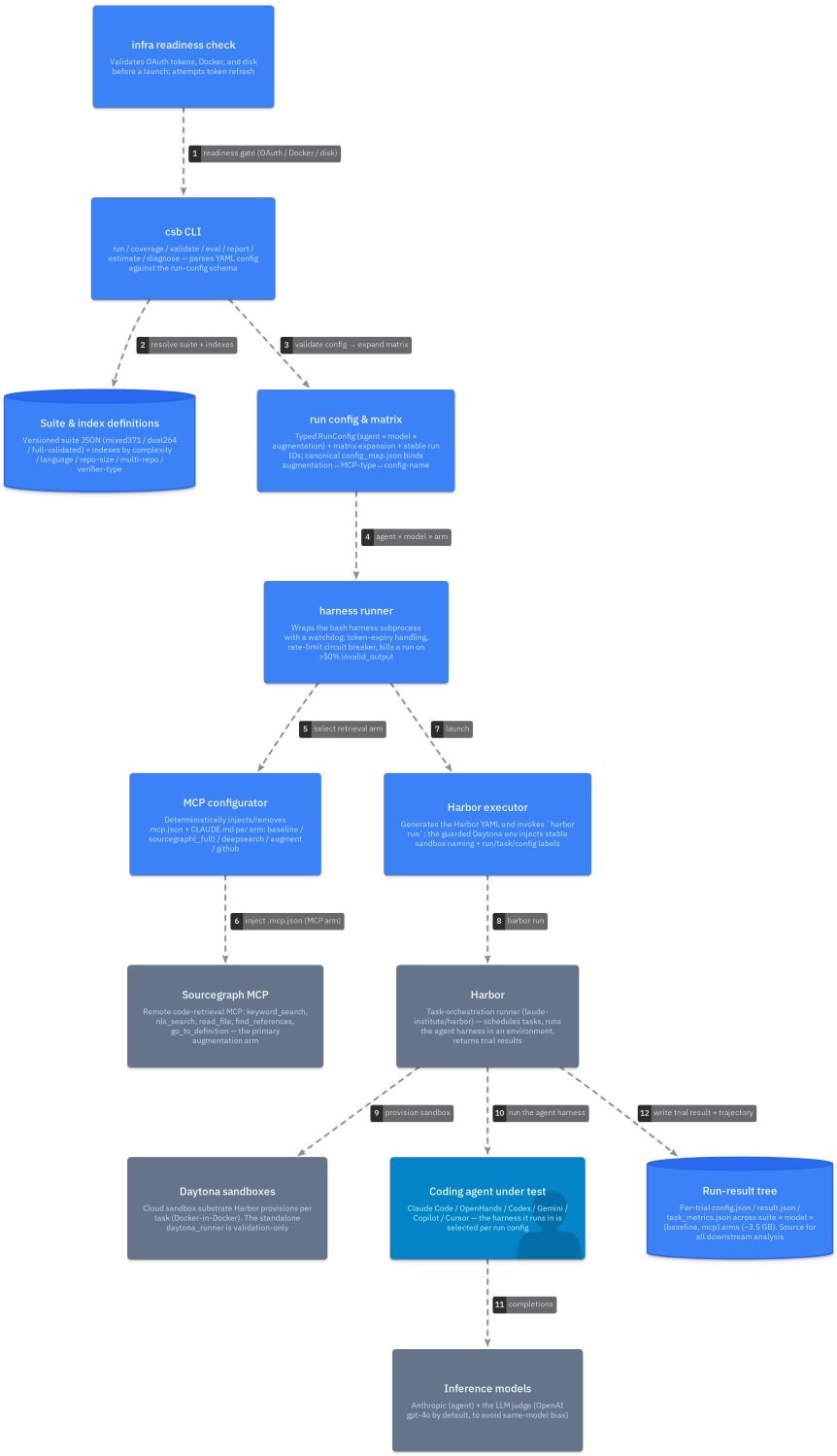

Figure 1: A benchmark run end to end. A typed run configuration selects a frozen suite; the launcher resolves each task, sets the retrieval configuration (local baseline or a Model Context Protocol server), and executes the agent in an isolated container; the result tree feeds scoring. The same agent, model, instruction, and budget are held fixed across configurations, so the retrieval channel is the only variable that changes.

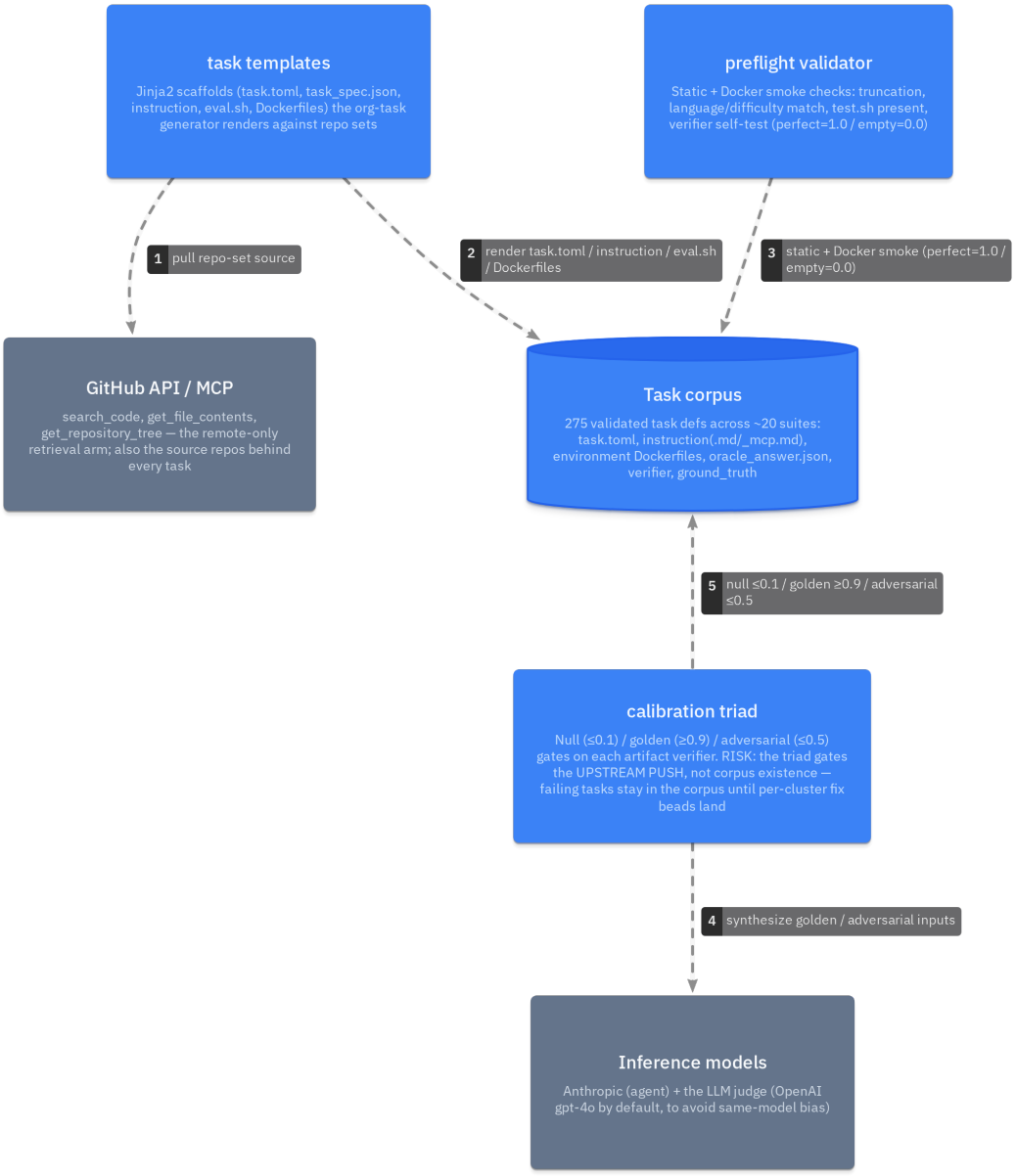

Figure 2: Task authoring and admission for the maintained corpus. Jinja templates and the use-case registry render a task skeleton against a pinned repository set; a preflight validator runs static and container smoke checks; and a null/golden/adversarial calibration triad gates each artifact verifier before the task is admitted. The figure shows the maintained version-two corpus (275 validated tasks across approximately twenty suites); the evaluation in Section 6 reports the frozen version-one suite of 370 paired tasks (Section 3.1).

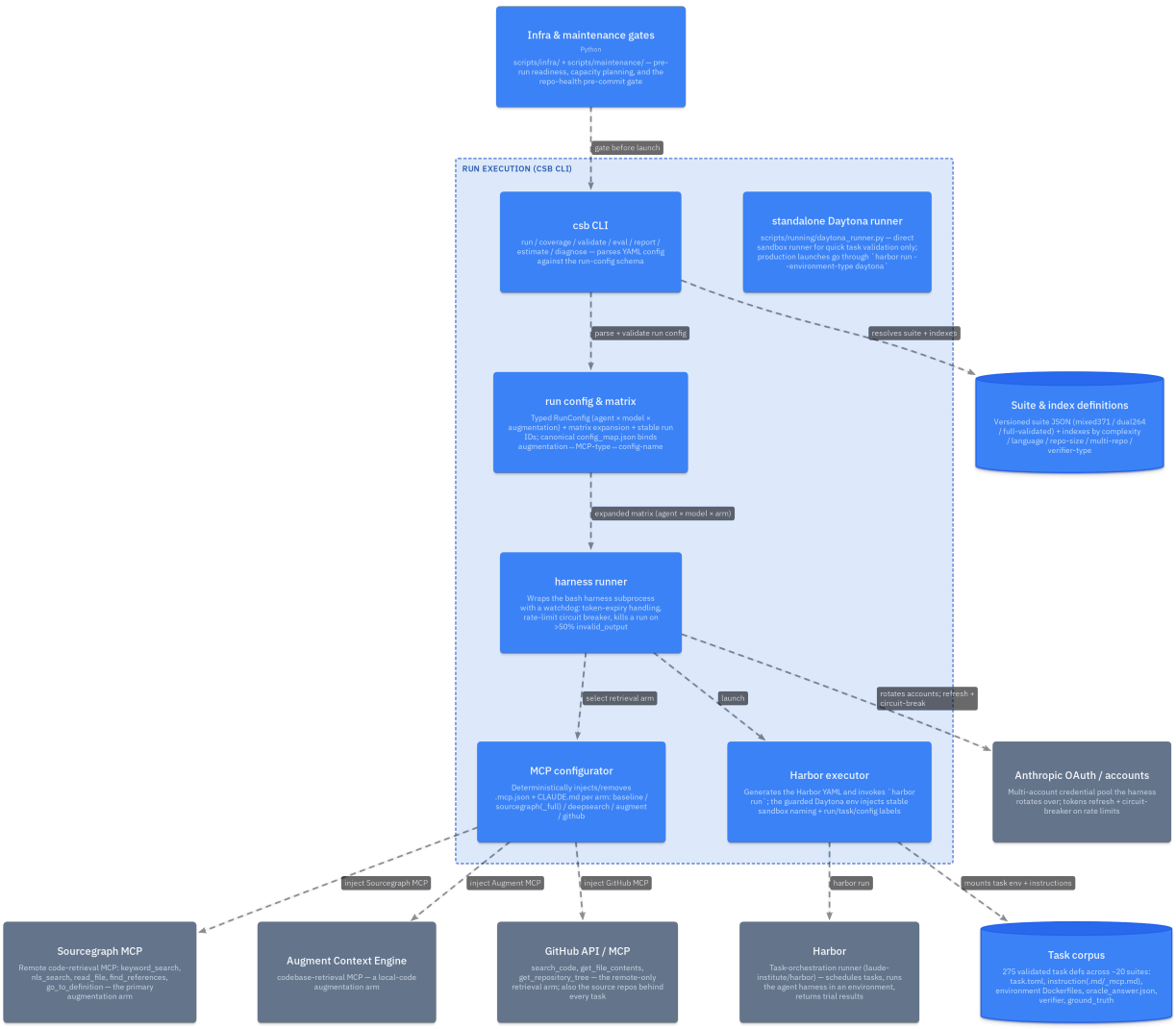

Figure 3: The run-execution boundary. The retrieval-server configurator and the launcher are original to this work; the agent runtime and the retrieval backends (Sourcegraph, and additional servers supported behind the same interface) are integrated dependencies. The configuration-to-environment map selects the container variant, the retrieval-server type, and the source-access mode for each run.

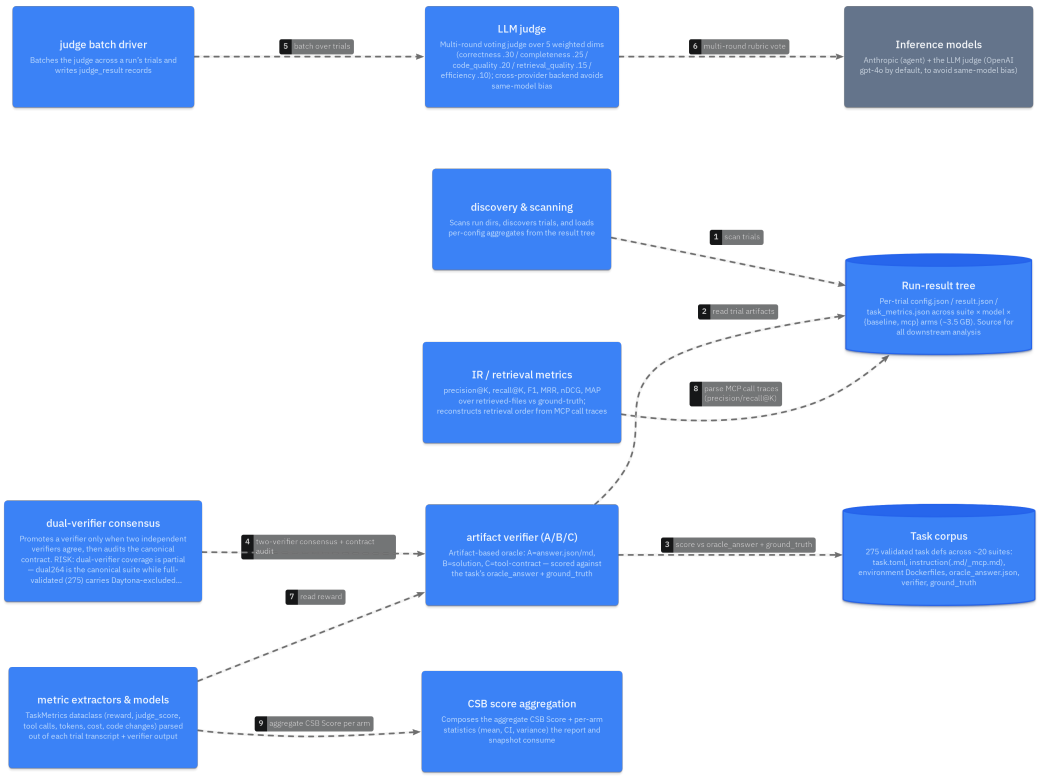

Figure 4: Scoring and metrics. The deterministic artifact verifier and the dual-verifier consensus produce the reward; information-retrieval metrics are computed from the agent's file accesses against the oracle; an optional LLM judge is recorded separately and never enters the reward path; and per-task rewards aggregate to the work-type-balanced score.