When Does Repository Search Help Code Completion?

Separating Retrieval, Dependency Extraction, and Prompt Utility in AIRCoder

Ho Anh Minh
TU Darmstadt
Darmstadt, Germany

Ivan Smirnov
TU Darmstadt
Darmstadt, Germany

Maryana Smirnova
TU Darmstadt
Darmstadt, Germany

Vladislav Veselov
TU Darmstadt
Darmstadt, Germany

Abstract

A code completion can need evidence from another file without needing a second language-model call to find it. We test this distinction in AIRCoder on 192 paired CrossCodeEval Python tasks and 192 paired RepoEval API tasks. Static-First RRF saves 50.0% and 19.8% of Qwen dependency calls, respectively, for one fewer exact match per benchmark. On CrossCodeEval, No-Dependency RRF achieves 66 exact matches, compared with 64 for Always Qwen-RRF. On RepoEval, a prefix heuristic that skips 85 searches reaches 32.29% Code Exact Match, close to the 32.20% expected from uniform skipping. Repoformer replay retains more matches while skipping fewer searches. Removing the visible cursor fragment from DraCo's input reduces resolved cases from 96 to 77 while exact matches rise from 64 to 68. Moreover, a 136-fold speedup in dependency extraction translates into a much smaller reduction in warm response time because generation from retrieved prompts still dominates. These results motivate a confidence router that checks whether static evidence changes the retained context before paying for a second model.

Keywords

repository-level code completion, selective retrieval, static analysis, rank fusion, interactive latency

1 Introduction

Type `self.http.` in a Python client and the cursor exposes a problem that a code model cannot always solve from the current file. Several methods might be plausible. The repository defines which ones exist, how they are called, and which convention this project follows. A developer might jump to the receiver's definition, inspect a neighboring call site, or keep typing with no search at all. A completion system must make that choice in the time it takes to offer a suggestion.

Early code language models such as Codex learned useful patterns from large code corpora [6]. Models in the StarCoder family continued that line of work [16, 21], but a project can define a method name or convention that general training alone cannot reliably supply. Retrieval-augmented generation (RAG) adds selected source snippets to the model's prompt [14, 35]. The extra evidence can turn a plausible guess into the correct API call. It also consumes time twice. The system first finds the snippets, then runs the generator on a longer prompt. For an inline suggestion, a result can arrive after the developer has already typed past it.

AIRCoder [26] is a strong test of that tension. It searches for similar code, likely dependencies, and similar repository structure at the same time. Its ranking needs only a small learned module, yet one of its eight features starts a separate call to Qwen3-8B. If a quick static lookup finds the right definition, calling the second model may add little. If the current file already supplies the answer, the whole search may be expendable. The decisions have different failure modes. A bad skip removes every external snippet. A bad extraction can distort the ranking.

Research questions. We trace each decision through the final prompt.

- **RQ1. Retrieval need.** How many searches can a prefix rule or a learned gate skip, and which exact completions are gained or lost?
- **RQ2. Extraction need.** How much quality and warm response time do static replacement, static-first fallback, and removal of the dependency score recover?
- **RQ3. Prompt utility.** How does extractor substitution change an adaptive ranker's prompts and task-level outcomes, and what happens when the static extractor sees less of the cursor line?
- **RQ4. Design.** What should an editor measure before it searches, calls a dependency model, or accepts a retrieved prompt?

Approach. We hold the generator and tasks fixed and intervene at two different points in AIRCoder. One intervention decides whether to search at all. The other changes the dependency signal while retaining the rest of the retrieval stack. A separate cursor-fragment intervention changes only what DraCo sees. We report changes in exact matches in both directions and measure warm request latency separately from dependency-extraction time. Alongside the experimental results, we audit the public AIRCoder release to distinguish what the paper describes from what its example code can reproduce.

Roadmap. Section 2 surveys repository completion. Section 3 explains AIRCoder, and Section 4 defines the interventions. Section 5 traces their effects on quality and latency. Section 6 develops a confidence router and discusses validity. Section 7 summarizes the design lessons. The companion site hosts measured replays and policy results.

2 Background and Landscape of Repository-Level Code Completion

Repository-level code completion (RLCC) predicts a short continuation at a cursor using evidence from a particular project. This setting narrows the broad RAG problem to a source tree, a partly observed edit, and a tight response budget. An API definition can establish the legal names and arguments. A nearby test can show what the project actually calls. A neighboring module may repeat the same pattern with different names. These are three different reasons for returning a chunk, and a successful retriever can exploit all three.

2.1 What to Retrieve

Analogous uses. Lexical search works when developers repeat recognizable terms. RepoCoder [35] retrieves similar repository windows, generates a draft, and uses part of that draft to update the next retrieval query. The draft supplies names missing from the visible prefix. It can also steer later rounds toward the first guess. Repository-level prompt generation (RLPG) learns which prompt proposal to use from sources such as neighboring files, parent classes, and imports [28]. Both exploit a useful feature of codebases. Developers often solve related problems in similarly situated parts of a project.

Explicit relationships. A shared token tells us where a name occurs. A dependency graph can tell us how a visible receiver or import connects to a definition. CoCoMIC exposes cross-file relations to the model during training [11]. DraCo uses a Python syntax and dataflow graph at retrieval time. Containment edges describe modules, classes, and functions, while dependency edges connect the entities a definition uses [7]. Its traversal can supply a type or provider directly, but an unfinished line may offer no useful starting symbol. GraphCoder expands the evidence to include local control and data relationships, which can distinguish similarly written snippets with different execution roles [20]. RepoHyper starts from a semantic search and expands and refines matches in a repository graph [24]. The graph is useful even when the first semantic hit is only a starting point.

Combining kinds of evidence. RepoFuse combines context that offers analogous patterns with context that explains the relevant dependencies [17]. AIRCoder implements a related distinction at the level of per-chunk features. It adds an explicit structural dimension and learns a mixture for each query. CodeRAG combines sparse, dense, and dataflow-guided retrieval with preference-aligned reranking [36]. The trade-off is where the expensive computation sits. In the configuration compared by Shi et al., CodeRAG reasons over candidates with an LLM. AIRCoder calls its model for dependency names, then scores candidates cheaply. These choices change the evidence a shared generator sees.

Beyond prompting. Monitor-Guided Decoding uses static information during token generation [1]. A decoder can consult a monitor to constrain an identifier as the model emits it. That addresses a different failure from fetching a good call-site example. Static information can therefore serve as a candidate source, a check on a retrieved candidate, or a constraint on the output. We vary the evidence presented to a fixed generator.

2.2 When to Retrieve

A gate learns a counterfactual. The question is whether a suggestion would improve if repository context were added. This is a relative judgment about two prompts for the same edit. Self-RAG makes a related choice in open-domain generation with learned retrieval and reflection tokens [3]. Repoformer trains a special retrieval decision on examples for which retrieval improved the reference-based edit similarity of the generated code [31]. The decision can run before repository lookup. It is tied to the downstream generator and candidate source used to produce its training labels. Replacing that downstream pipeline changes the meaning of the score.

Stopping and skipping. RLCoder learns retrieval from the generator's feedback and includes a stop signal [29]. Stopping lets the system decide when further context has low value. A pre-search gate decides whether to buy the first candidate at all. AIRCoder makes yet another decision given retrieved candidates. It learns how to combine their evidence. Table 1 places these choices side by side. They can be composed, but the training target must correspond to the stage being controlled. A model trained to select a better snippet has no direct supervision about whether the editor would have been better off with an in-file prompt.

Costs belong to the whole request. A gate that skips a search can avoid both dependency extraction and generation from a longer prompt. An extractor replacement can save most of the cost of getting names while leaving prompt length similar. A useful benchmark should record both changes. Otherwise a method can appear fast by timing only its cheapest stage, or appear reliably accurate by aggregating tasks on which its suggestions changed in opposite directions.

2.3 How RLCC Is Evaluated

The cursor determines what can be inferred. CrossCodeEval finds sites that rely on project-local imports using static analysis. It evaluates Python, Java, C#, and TypeScript examples with a visible portion of the target line [10]. A partially typed receiver or method can help a static resolver. RepoEval includes line, API, and function completion [35]. Its API tasks in our study start before the target line, giving the extractor less direct evidence about the upcoming call. They still test API usage across a repository. Describing both settings as line completion can hide the change in what the resolver sees.

Retrieval and generation are different tests. RepoBench separately tests retrieving evidence, completing with supplied context, and running the full pipeline [19]. This distinction matters when multiple completions are valid. A poor output can mean the retrieval stage missed the evidence, the evidence was crowded out by a budget, or the generator failed to use it. Selecting a single snippet that performs well on its own also differs from selecting a set of snippets that work well together. DevEval uses repository dependencies and runnable tests to probe functional correctness [15]. Long Code Arena adds project-level evaluation at larger context scales [5]. These designs measure complementary pieces of the same problem.

A metric must match the user task. Code Exact Match (Code EM) is a direct check against the reference. Edit Similarity (ES) measures how much text changes. Identifier scores make API spelling and name selection more visible. A small edit can still invalidate a method call, and valid variations can differ from the reference. Fill-in-the-middle (FIM) models also make it possible to condition on the existing code after the cursor [4]. That suffix can specify a variable or an indentation constraint unavailable to a prefix-only model. We use prefix-only completion for the paired evaluation and isolate suffix sensitivity in the retrieval-gate replay.

Table 1: Representative repository-level completion systems. Languages refer to reported evaluation or implementation scope. A stop decision inside retrieval differs from a gate before the first search.

Method	Retrieval evidence	Dependency mechanism	Selective gate	Languages	Main cost or failure point
RepoCoder [35]	Lexical overlap and draft completion	Implicit in similar windows	No	Python	Repeated generation and search
RLPG [28]	Prompt proposals from project structure	Imports and parent classes supply proposal sources	Prompt selection	Java	Proposal coverage
DraCo [7]	Syntax graph and dataflow	Static containment and dependency edges	No	Python	Visible symbols and graph coverage
GraphCoder [20]	Graph similarity between code contexts	Static control flow and dependencies	No	Python, Java	Context-graph construction
CodeRAG [36]	Graph-guided candidates and an LLM reranker	Graph context informs retrieval	No	Python	LLM reranking
Repoformer [31]	Lexically retrieved snippets	Implicit in retrieved text	Learned pre-search gate	Python, Java, C#, TypeScript	Calibrating search benefit
RLCoder [29]	Learned snippet relevance	Implicit in retrieved text	Learned stop signal	Python, Java	Generator-dependent reward
AIRCoder [26]	2 (25%) text, 1 (12.5%) dependency, 5 (62.5%) structure	LLM proposes names to match in definitions	No	Python, Java, C#, TypeScript	Separate dependency-model call

3 AIRCoder: Architecture, Fusion, and Bottlenecks

AIRCoder treats each repository chunk as a candidate answer to a question about context: which part of this project would help the code model complete the current edit? Its design separates the preparation of candidates, the computation of eight scores, and the combination of those scores. Figure 1 marks the two places where our study changes the pipeline.

3.1 Multi-Dimensional Scoring

Start with coherent candidates. AIRCoder uses structure-preserving chunking to avoid arbitrary cuts through definitions [26]. A chunk carries its path, enclosing scopes, and signatures defined inside it. This keeps a short method tied to the class that gives it meaning. The paper specifies 800-token chunks and an 8,000-token generator input limit. For generation, the study caps the visible prefix at 1,000 tokens. The gate and dependency extractor receive the full visible prefix. A weak high-ranked chunk can displace a stronger one under the token budget.

Textual signals. The sparse feature asks how many token types a candidate shares with the immediate cursor context. It will favor a neighboring example that uses the same API vocabulary. The dense feature compares UniXcoder vectors [12] and can retrieve an analogous calculation whose author chose different names. Sparse and dense features can agree on a copied pattern or disagree when an identical identifier has different semantics in a different module.

Dependency signal. Qwen3-8B [33] receives the visible code and is asked for at most two class or function names to guide completion. The released matcher tests whether an extracted name of at least three characters occurs as a substring in a candidate's class or function signatures. A chunk gets a binary feature even if several names match. DraCo can supply names to this interface using the graph instead. A retrieved class can have the right spelling in the wrong namespace. This possibility matters when the feature gives the same boost to every candidate matching an extracted name.

Structural signals. Four sparse scores measure proximity in the file tree and lexical similarity of paths, class signatures, and function signatures. A second embedding compares the structural summaries. It can surface corresponding modules despite shallow naming overlap. Table 2 gives all eight computations. Text embeddings see code. Structure embeddings see where it belongs.

3.2 Fixed and Query-Conditioned Fusion

Reciprocal ranks. The eight raw scores answer different questions on different scales. Reciprocal Rank Fusion (RRF) [9] converts each feature into a ranked list and rewards candidates supported by several lists. For candidate c, query q, and active metrics $\mathcal{M}$,

$$\mathrm{RRF}(c, q) = \sum_{m \in \mathcal{M}} \frac{1}{k + \mathrm{rank}_m(c, q)}, \qquad k = 60. \tag{1}$$

Ranks start at one. The constant dampens the difference between very high ranks. RRF gives every metric the same formula, so the ranking contains no learned preference for one feature over another. Our implementation averages tied ranks within a metric. An all-zero dependency column then adds the same value to every candidate and cannot affect their relative order.

Learned weights. A query-conditioned ranker gives every candidate for the same cursor the same eight weights. For a query embedding $e_q \in \mathbb{R}^{768}$ and feature values $f_m(c, q)$, the released deep model computes

$$\begin{aligned} w(q; \theta) &= \mathrm{softmax}(\mathrm{MLP}_\theta(e_q)), \\ s_\theta(c, q) &= \sum_{m=1}^{8} w_m(q; \theta)\, f_m(c, q). \end{aligned} \tag{2}$$

The MLP has widths $768 \to 256 \to 128 \to 64 \to 8$. Each hidden linear layer is followed by LayerNorm, GELU, and dropout at 0.1 during training. Softmax makes the weights nonnegative and sum

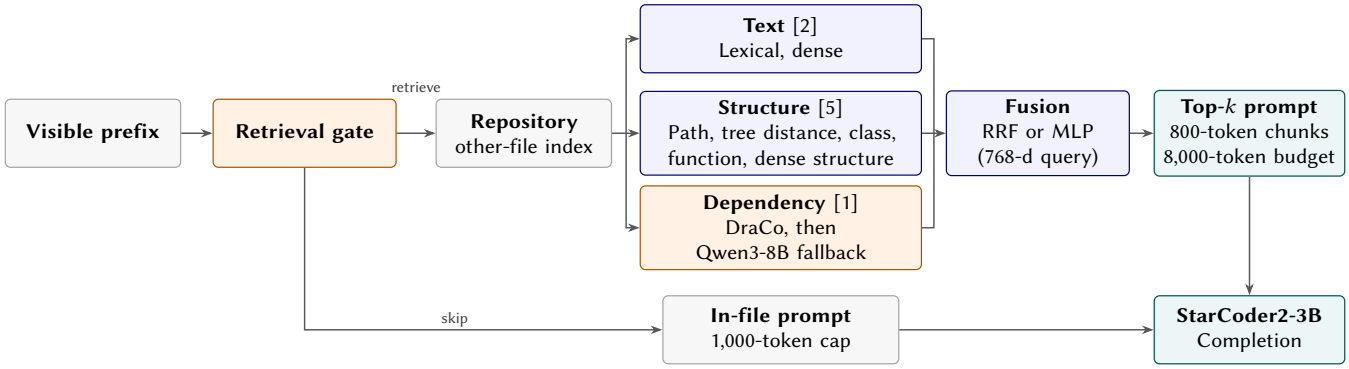


Figure 1: AIRCoder's three retrieval dimensions and the study interventions. The gate can route directly to generation with the visible prefix. On the search branch, Static-First RRF changes only the dependency feature. Both paths use StarCoder2-3B. The 768-dimensional UniXcoder query vector supplies the learned fusion weights. Sizes refer to the study's token caps.

Table 2: The eight AIRCoder retrieval features. The computation describes the released score definitions used as the basis of the paired study. E is a UniXcoder embedding and $\mathrm{ES}(a,b) = 1 - \mathrm{Lev}(a,b)/\max(|a|,|b|)$ is normalized character edit similarity.

Dimension	Signal	Type	Computation for candidate c and query q	Typical failure mode
Text	Token overlap	Sparse	Jaccard $\lvert T_q \cap T_c\rvert / \lvert T_q \cup T_c\rvert$ on token sets, with the latest 100 query tokens	Frequent tokens or copied boilerplate dominate
Text	Code similarity	Dense	Cosine $\cos(E(q), E(c))$ of code embeddings	Semantically similar code calls a different API
Dependency	Dependency existence	Binary	**1** if an extracted name occurs in a candidate class or function signature in the released matcher	A plausible name can refer to a different definition. Static coverage falls on empty lines
Structure	File-path similarity	Sparse	$\mathrm{ES}(\mathrm{path}(q), \mathrm{path}(c))$	Inconsistent naming hides related files
Structure	File proximity	Sparse	$1/\max(1, d)$, with d the number of path edges outside the shared prefix	Neighboring files can implement unrelated roles
Structure	Class similarity	Sparse	Maximum ES from the last query class to the candidate's class signatures, or zero if absent	Generic class names and absent enclosing classes
Structure	Function similarity	Sparse	Maximum ES from the last query function to candidate function signatures, or zero if absent	Similar helper names cover different behaviors
Structure	Structure similarity	Dense	Cosine of embeddings of path, ancestors, and local signatures	Compressed structure can obscure the decisive name

to one. The ranker consumes the feature values directly, with zero for missing fields in the released formatter. The two embedding features use cosine similarity and the lexical scores follow their own normalization in Table 2. The reduced fit constructs e_q from the structural summary and the last 15 lines of visible text.

Pairwise supervision. For a development query, a candidate's label is the ES of the continuation produced using that candidate alone. Suppose candidates c_i and c_j give labels y_i and y_j. The training target is their gap, which compares snippets on the same edit even when different edits have different difficulty.

$$\mathcal{L}_{ij}(\theta) = \left[(s_\theta(c_i, q) - s_\theta(c_j, q)) - (y_i - y_j)\right]^2. \tag{3}$$

The reduced fit averages this loss over candidate pairs. At inference, all eligible candidates are scored, ordered, and considered under the prompt budget. Several strong candidates can still repeat or contradict each other.

3.3 Published Findings and Structural Bottlenecks

Multi-dimensional search raises quality. With StarCoder2-3B, the learned AIRCoder reported in Table 3 of the seed paper scores 36.98% Code EM on CrossCodeEval Python, 2.98 percentage points (pp) above CodeRAG. On RepoEval API it scores 50.12%, 5.33 pp above RepoCoder [26]. In Table 7, retrieval takes 0.366 s per query on average, 10.2 times faster than CodeRAG's 3.732 s. Much of CodeRAG's cost was attributed to LLM reranking. AIRCoder concentrates the model call upstream, extracting a few names once and reusing them across the candidate pool.

Dependency extraction still dominates search. The dependency model consumes 0.320 s of the seed paper's 0.366 s retrieval average, or 87.4% [26]. AIRCoder also runs retrieval on every query. Repoformer found ES gains on no more than one fifth of cases in its RepoEval comparisons [31]. If a search can be bypassed safely, the saving reaches the generator as well as the dependency model.

If retrieval remains useful, the narrower intervention is to replace the dependency extractor.

The useful mixture depends on the input. The seed paper's Table 4 reports 36.68% Code EM with Qwen and 36.88% with DraCo on CrossCodeEval Python, but 49.77% and 47.86% on RepoEval API. The same table shows that training the fusion module on the other benchmark reduces Code EM. Appendix D finds that dependency evidence receives more weight on CrossCodeEval, while dense structure is more prominent on RepoEval [26]. A feature can matter in one ranking regime or cursor setting and be less valuable in another. Sparse structural matching has the additional constraint that comparable parts of a repository should have comparable names.

Reproducibility has a concrete boundary. Table 3 records what is available at the pinned AIRCoder release [27]. The public RRF driver uses a stable ordinal rank and leaves out dense structure. The paper describes eight features. Its appendix describes ten training epochs, whereas the released training script uses five. In the public script, subtracting a target vector of shape B from predicted gaps of shape $B\times1$ can broadcast across candidate pairs. Our fit uses a vector of B predicted gaps, implementing Equation 3 for corresponding pairs. The original Table 4 training inputs and checkpoints would be needed to determine which details were used for those published runs. The reduced comparison regenerates supervision and reuses the public architecture.

4 Study Design and Experimental Protocol

Each test fixes the Python task, visible prefix, and generator. The policies differ in how evidence from other files reaches the prompt.

4.1 Interventions and Policy Definitions

Table 4 defines the policies and their short labels. Changing the extraction route leaves seven retrieval features and the prompt builder in place. Skipping search bypasses all eight features and sends only the visible prefix to the generator.

Prefix rule. The rule reads the full visible input for Python import statements and gathers their aliases. It tokenizes identifiers in the last four nonempty, non-comment lines. If any of those identifiers is an alias of a relative import or of a module whose root is outside Python's standard library, it keeps retrieval active. Otherwise it skips only if the last such line contains a whole-word `return`, `raise`, `yield`, `assert`, `self`, `cls`, `True`, `False`, or `None`, a square bracket or brace, a single or double quote, or any digit. With neither an import match nor a cue, it searches. The scanner is lexical. A keyword in a string can satisfy its test, and a digit or receiver name can accompany a missing API.

Learned gate replay. The prefix-only Repoformer comparison uses the publicly released personal reproduction checkpoint, `repoformer-1b`, with the original token layout [30, 32]. It formats the source-anchored prefix in a FIM template with an empty suffix and thresholds the next-token probability of the retrieval marker at $p > 0.2$ [31]. Each decision selects one of two precomputed completions for that exact task. A separately recorded sensitivity replay supplies the anchored source text after the full target span. It changes the gate input only. The comparison holds the downstream prompt and generation fixed within each branch.

Reduced adaptive fit. We fit a separate checkpoint for each benchmark, using development tasks with independent task IDs. We probe each feature's top candidate and use stable RRF to refill to eight distinct candidates where possible. Each candidate gets one single-snippet completion from StarCoder2-3B. Its released Code ES supplies the label in Equation 3. We use the released pair sampler with a 0.7 high-score threshold, seed 101, Adam at 10^{-3}, batches of 32, and five epochs. We select the checkpoint with highest validation top-one ES, breaking ties by earliest epoch. At evaluation we feed both extractors the same query weights and shared features and regenerate prompts from their respective rankings.

4.2 Benchmarks, Controls, and Statistics

Paired cohorts. The study uses the official CrossCodeEval Python and RepoEval API releases [2, 23]. CrossCodeEval has 192 tasks, four per repository across 48 repositories held out from development. RepoEval has 192 tasks from eight repositories, 24 per repository. Its development and evaluation task IDs are disjoint within the same repository set. We use the CodeGen context edition for all paired RepoEval arms.

Scope guard and prompt construction. All retrieval arms rank chunks from pinned snapshots. A guard excludes the active file and its identical decoded copies, including their entries in DraCo's module resolver. A scope test verified that the guard still allows a separate provider to resolve. Chunks carry path headers. The prompt builder considers candidates in rank order, retains those that fit, and places the highest-ranked retained snippets closest to the cursor. All arms give the same visible text to the completion model. Reference text enters only in scoring and training-label generation on the separate development tasks.

Two exact-match measures. Code EM uses AIRCoder's released evaluator. It strips blank and comment-only lines, keeps as many generated lines as the cleaned target has, then joins and truncates the prediction to the reference character length. Strict EM compares those same lines without character truncation. Code ES uses the released, character-trimmed edit similarity. Code EM and Strict EM agree in the paired policy results, so one column represents both in those tables. Their small differences in the full in-file baseline remain visible.

Baseline scale. Table 6 puts the paired results in context. The full CrossCodeEval in-file run yields 6.38% Code EM, compared with 6.60% in the seed paper. The RepoEval variants yield 26.88% and 26.94%, compared with the published 26.06%. These comparisons check the scale of unassisted quality. Every retrieval lift below is computed against the in-file arm on the corresponding paired tasks.

Paired outcome and uncertainty. For any two policies, a gain is an Inexact-to-Exact change and a loss is the reverse. Let Y_i^R, Y_i^L be the retrieval and in-file exactness indicators for task i, and $g_i = 1$ if a gate runs retrieval. The quality delta against always retrieving is

$$\Delta_{\text{gate}} = \frac{1}{n}\sum_{i=1}^{n}(1-g_i)(Y_i^L - Y_i^R). \tag{4}$$

A task that agrees across branches still contributes to the search budget. It contributes zero to the quality delta. Intervals use percentile bootstrap resampling of repositories, with 10,000 draws for

Table 3: Public release audit at commit `c3c344d674ba`. Missing counts refer to the exact artifacts needed to rerun the published learned comparison.

Item checked	Pinned public release	Study treatment
Realized split files, oracle labels, checkpoints	4 split files, 2 oracle files and 2 selected checkpoints missing (100% of the 8 checked files)	Generate new oracle labels on independent development splits and train new checkpoints
RRF signal list	7 of 8 scores active (87.5%), omitting dense structure	Restore dense structure and average ties in each feature
Distributed RRF example	1,000-token chunks and an initial 200-row input slice	Use syntax-aware chunks capped at 800 tokens in the paired study
Adaptive training script	Five epochs and test-oracle EM checkpoint tracking. Literal tensor shapes permit cross-pair broadcasting	Train for five epochs, score corresponding pairs, select by validation top-one ES

Table 4: Canonical policy names and their search behavior. The last columns give repository searches out of 192 per benchmark. Repoformer counts describe the prefix-only replay.

Policy	Short label	Search and extraction rule	CrossCodeEval	RepoEval API
In-file only	In-file	Use the visible in-file prefix with no search.	0 (0%)	0 (0%)
Always Qwen-RRF	Qwen-RRF	Always search with RRF and a Qwen3-8B dependency call.	192 (100%)	192 (100%)
Always DraCo-RRF	DraCo-RRF	Always search with RRF and static dependency extraction.	192 (100%)	192 (100%)
No-Dependency RRF	No-Dep RRF	Always search with RRF and an all-zero dependency column.	192 (100%)	192 (100%)
Static-First RRF	Static-First	Use DraCo if a returned name matches an eligible chunk. Otherwise call Qwen.	192 (100%)	192 (100%)
Prefix-Gated Qwen-RRF	Prefix Gate	Use the prefix rule to select In-file or Qwen-RRF.	166 (86.46%)	107 (55.73%)
Repoformer-Gated Qwen-RRF	Repoformer (replay)	Replay the released 1B gate to select In-file or Qwen-RRF.	183 (95.31%)	120 (62.50%)
Prefix-Gated Static-First	Gate + Static-First	Use the same prefix rule to select In-file or Static-First.	166 (86.46%)	107 (55.73%)
Learned Fusion (Qwen / DraCo)	Learned + Qwen / DraCo	Always search with a frozen adaptive ranker. Swap only its dependency feature.	192 (100%)	192 (100%)

Table 5: Shared inference controls and warm timing protocol. Revisions are abbreviated from the recorded model pins.

Component	Setting
Generator	StarCoder2-3B, `733247c55e3f`, greedy BF16, 48 new tokens at most
Dependency	Qwen3-8B, `b968826d9c46`, or DraCo's Python resolver
Embeddings	UniXcoder, `5604afdc964f`
Prompt budget	Up to 1,000 prefix tokens and ten ranked 800-token chunks under an 8,000-token total cap
Source guard	Exclude the active file and exact decoded copies from candidates and static module lookup. Score after generation.
Warm passes	48 tasks per benchmark, randomized policy order, loaded models and indexes, one request at a time
Runtime	AMD MI300A, ROCm PyTorch 2.8.0, Transformers 4.56.0

the fixed-fusion study and 5,000 for the adaptive study. Each draw keeps a project's paired task outcomes together.

Table 6: Full in-file sanity check with StarCoder2-3B. The published references are seed-paper Table 1 Code EM percentages for the corresponding benchmark [26]. The paired comparison uses its own task-matched in-file baseline.

Source	N	Code EM	Strict EM
CrossCodeEval Python	2,665	170 (6.38%)	169 (6.34%)
RepoEval API, CodeGen	1,600	430 (26.88%)	427 (26.69%)
RepoEval API, Codex	1,600	431 (26.94%)	427 (26.69%)

A count-matched random reference. For a uniform selection of k tasks to skip, each task falls on the in-file branch with probability k/n. If $M_R = \sum_i Y_i^R$ and $M_L = \sum_i Y_i^L$, the expected number of exact completions is

$$\mathbb{E}[M_k] = M_R + \frac{k}{n}(M_L - M_R). \tag{5}$$

This reference answers what a gate would retain if it spent the same search budget without inspecting the task. It uses the observed branch outcomes, so it accounts for searches that help, hurt, or leave the score unchanged.

Warm latency. Timing uses one task per CrossCodeEval repository and six per RepoEval repository. Each policy runs with live extraction and no reuse of another policy's work. The warm end-to-end timer starts before routing and stops after generation. It includes prompt construction and excludes scoring. The adaptive study separately records extractor component latency. In all 576 warm observations, the final prompt hash equals that of the corresponding quality run. The study reads accuracy from the paired generation pass and latency from the warm pass.

5 Empirical Results

5.1 Skipping Retrieval

Repository search pays off on both benchmarks. Always Qwen-RRF raises exact matches from 8 to 64 on CrossCodeEval and from 49 to 72 on RepoEval. The changes come from 58 gains and two losses on CrossCodeEval, and 27 gains and four losses on RepoEval. Net lift is 29.17 pp (95% cluster interval [21.88, 36.46]) and 11.98 pp ([4.69, 20.83]). The cross-file tasks give the gate a harder decision. At the same skip count, uniform selection would remove more useful searches there. RepoEval offers more opportunities to skip harmless searches, but the gate must identify them.

The prefix rule lands near random skipping on RepoEval. At 85 skipped searches, Prefix-Gated Qwen-RRF records 62 exact matches (32.29%). Uniform skipping at that same budget gives an expectation of 61.82 matches (32.20%). It avoids three harmful searches and gives up thirteen helpful ones. On CrossCodeEval it gives up three and gains none while skipping 26 searches. Its 31.77% exceeds the 29.38% random expectation there. The policy is conservative by design on the cross-file benchmark. Recent relative or non-standard-library imports retained search on 150 CrossCodeEval tasks but on only 51 RepoEval tasks. This rule misses the important distinction between a locally bound identifier and a call whose spelling or arguments are defined by a repository convention. On RepoEval, the last visible statement can have a local cue even when the next line needs a cross-file API.

Repoformer keeps more matches at a different point on the budget curve. In the prefix-only replay Repoformer retains 63 CrossCodeEval matches while skipping nine searches and 68 RepoEval matches while skipping 72. On RepoEval it avoids three harmful searches and misses seven helpful ones. It performs 13 more searches than the prefix rule and returns six more exact matches. At 72 skips, uniform choice would yield 33.01% Code EM, compared with the replay's 35.42%. This suggests useful selectivity at the measured point. The two gates still need a matched-budget sweep to compare them directly.

Right context moves the gate's operating point. With the same threshold and true source suffix, Repoformer makes 164 searches on CrossCodeEval and 94 on RepoEval. Its replay keeps 58 and 63 matches, respectively. That is five fewer matches on each benchmark, with correspondingly fewer searches. The gate's prefix-only median prefill is 9.7 ms on CrossCodeEval and 16.6 ms on RepoEval. These are gate costs. The quality lever here is the change in branch decisions. A suffix changes how often the gate searches at a fixed threshold. An editor with FIM input needs a threshold calibrated on that input.

5.2 Replacing the Dependency Extractor

Keep the search and shorten the expensive branch. Static-First RRF spends 96 fewer calls on CrossCodeEval (50.0%) and 38 fewer calls on RepoEval (19.8%). Each cohort gives up one exact match relative to Always Qwen-RRF. The cross-file context remains available because Static-First still searches every time. Prefix-Gated Qwen-RRF uses only 107 Qwen calls on RepoEval and loses ten matches. Prefix-Gated Static-First cuts Qwen calls further, to 75, and loses eleven. In this cohort, the most aggressive savings carry a larger quality cost than changing the extractor alone.

A dependency score can hurt a strong ranking. On CrossCodeEval, No-Dependency RRF finishes with 66 exact matches, ahead of both 64-match always-on extractors and 63-match Static-First. Against Qwen-RRF it gains four and loses two, a +1.04 pp point estimate (95% cluster interval [−1.56, 3.65]). On RepoEval it ties DraCo-RRF at 69 exact matches and gives nearly identical edit similarity, 65.14% versus 65.13%. This is a zero-extraction-cost policy in both cases. It still assembles a retrieved prompt from text and structure.

The RRF mechanism explains why the result is plausible. A positive dependency score can move a chunk ahead even when its textual and structural evidence is weak. If a name is plausible but points to a less useful signature, giving its rank an equal vote can crowd out a better example. An all-zero column cannot change relative rank. The paired experiment holds fusion fixed. In the seed paper, removing the feature from the learned ranker instead reduced CrossCodeEval Code EM from 36.68% to 32.25% [26, Table 4]. Section 5.3 shows what changes when the dependency feature receives a query-specific weight.

A fast component leaves a long prompt. In the adaptive experiment, median extraction drops from 408.8 to 3.0 ms on CrossCodeEval and from 452 to 20 ms on RepoEval. The component p95 drops from about 2,766 ms to 7 and 36 ms. The separately timed fixed-RRF requests are a different measurement. Their warm wall-time medians move from 1.99 to 1.58 s with Always DraCo-RRF on CrossCodeEval and from 2.09 to 1.63 s on RepoEval. Static-First reaches 1.72 s on CrossCodeEval. On RepoEval its median stays at 2.09 s, even as the p90 falls from 4.31 to 3.48 s.

Figure 2 shows why these effects differ. Always Qwen-RRF needs about 20 ms and 45 ms median for ranking and prompt construction, while median generation is roughly 1.56 s and 1.58 s. In-file generation takes 0.67 s and 0.68 s. The 8,000-token limit is a cap. Actual Always Qwen-RRF quality prompts average 6,002 and 6,490 tokens. Generation from these longer prompts is the main cost remaining after a dependency call is removed. On RepoEval, Prefix Gate takes the in-file branch often enough to drop wall-time p50 to 1.10 s and p90 to 2.27 s. Table 7 shows the price in exact matches.

5.3 Learned Fusion and Task-Level Churn

The extractor interacts with the weights. Learned Fusion ties at 65 CrossCodeEval matches with either extractor. On RepoEval it reaches 76 with Qwen and 72 with DraCo. These are the same directional patterns as the published switch in seed-paper Table 4, which produces a small CrossCodeEval change and a lower RepoEval point estimate with DraCo. Table 9 gives the reduced checkpoints and

Table 7: Retrieval routing in the 192-task paired cohorts. Code EM and Strict EM agree. Each gain or loss is relative to Always Qwen-RRF. The random row is Equation 5 evaluated at the Prefix Gate's skip count.

	CrossCodeEval Python			RepoEval API		
Policy	Code EM	Searches skipped	Gains/losses	Code EM	Searches skipped	Gains/losses
In-file	8 (4.17%)	192 (100%)	2/58	49 (25.52%)	192 (100%)	4/27
Qwen-RRF	64 (33.33%)	0 (0%)	–	72 (37.50%)	0 (0%)	–
Prefix Gate	61 (31.77%)	26 (13.54%)	0/3	62 (32.29%)	85 (44.27%)	3/13
Uniform at Prefix Gate budget	56.42 (29.38%)	26 (13.54%)	–	61.82 (32.20%)	85 (44.27%)	–
Repoformer (replay)	63 (32.81%)	9 (4.69%)	0/1	68 (35.42%)	72 (37.50%)	3/7
Repoformer (replay), source suffix	58 (30.21%)	28 (14.58%)	1/7	63 (32.81%)	98 (51.04%)	3/12

Table 8: Dependency interventions under fixed RRF. Both measures of exact match give the Code EM count shown. Calls are the number of Qwen3-8B requests per 192 tasks. ES gives released Code Edit Similarity.

	CrossCodeEval Python			RepoEval API		
Policy	Code EM	ES (%)	Qwen calls	Code EM	ES (%)	Qwen calls
Qwen-RRF	64 (33.33%)	67.74	192 (100%)	72 (37.50%)	65.68	192 (100%)
DraCo-RRF	64 (33.33%)	67.31	0 (0%)	69 (35.94%)	65.13	0 (0%)
No-Dep RRF	66 (34.38%)	68.46	0 (0%)	69 (35.94%)	65.14	0 (0%)
Static-First	63 (32.81%)	67.54	96 (50.00%)	71 (36.98%)	65.59	154 (80.21%)
Gate + Static-First	59 (30.73%)	65.69	83 (43.23%)	61 (31.77%)	63.32	75 (39.06%)

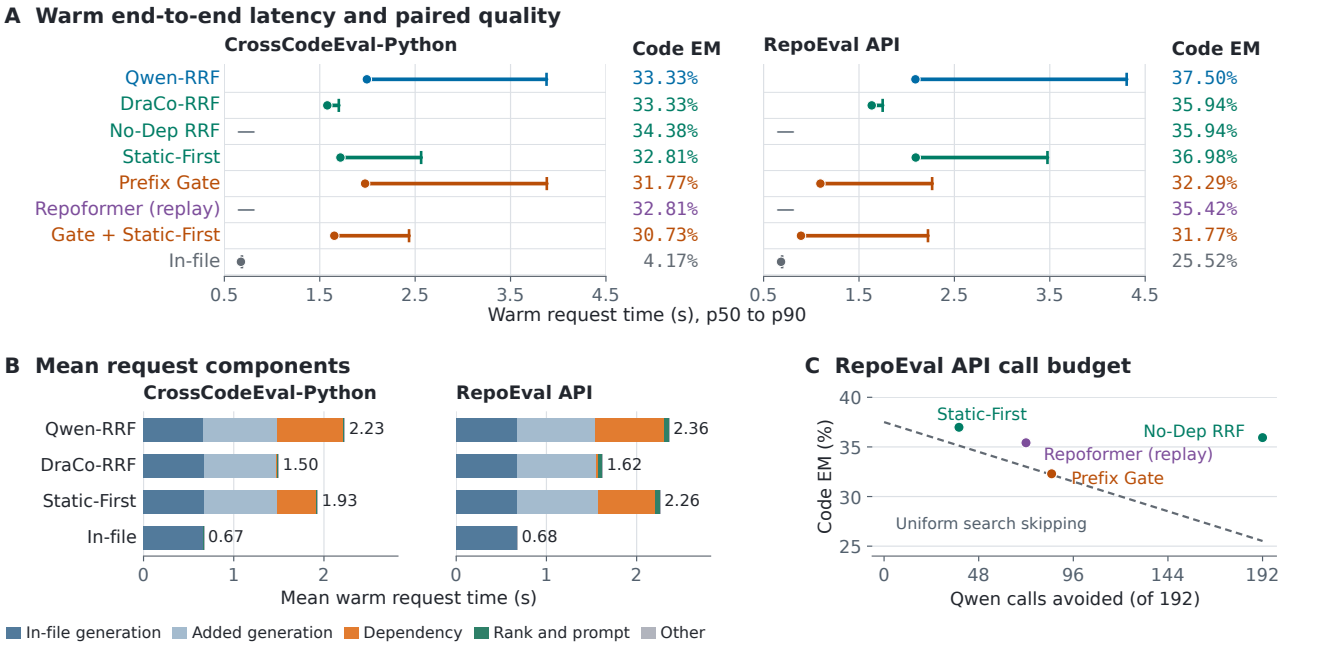


Figure 2: Quality, warm latency, and Qwen-call budget. (A) Code EM is measured in the paired cohorts. The p50 and p90 warm end-to-end times come from separate 48-task runs. Dashes mark policies without warm measurements. (B) Stacked per-request mean components decompose measured wall time. Extra generation time is relative to matched in-file prompts. The residual covers time outside the component timers. (C) The RepoEval dashed curve gives the uniform search-skip expectation at each Qwen-call budget. Static-First still searches every time.

paired intervals. In these conditions, Qwen keeps more exact RepoEval matches under both kinds of fusion, with a larger observed gap in the learned ranker. The learned result shows that a dependency signal can still matter inside a query-conditioned mixture. The

substitution changes which names match. The query-conditioned weights stay fixed.

Table 9: Reduced learned fusion. Holdout size is 192 per benchmark. Change [95% CI] is Learned + DraCo minus Learned + Qwen in percentage points.

Measure	CrossCodeEval	RepoEval API
Train/validation tasks	64/16	100/24
Selected epoch	3	2
Validation ES	70.12%	71.74%
Learned + Qwen	65 (33.85%)	76 (39.58%)
Learned + DraCo	65 (33.85%)	72 (37.50%)
Gains/losses	7/7	2/6
Change [95% CI]	0.00 [−3.65, 4.17]	−2.08 [−6.25, 2.08]

Exact-match parity can hide a changed suggestion. The two 65-match totals on CrossCodeEval contain seven gains and seven losses. The dependency vector changes on 144 tasks, the retained prompt on 128, and exactness on 14. On RepoEval those counts are 169, 120, and eight. A feature can alter the ranking while preserving the prompt. A changed prompt can preserve exactness. Figure 3 shows the task outcomes that disappear when only net totals are reported.

A checkpoint learns the signal distribution it sees. In the adaptive evaluation, Learned + Qwen yields at least one positive candidate on 155 CrossCodeEval and 173 RepoEval tasks. The DraCo replacements yield positive candidates on 96 and 38. The fixed checkpoint continues to apply the same weight to a positive or absent dependency feature even though static hits appear much less often. The larger RepoEval gap is consistent with this distribution shift. Training with static and hybrid signals would test whether matching training and serving distributions recovers some of the lost matches. A prompt-level target would additionally account for the contribution of the other retained chunks, which single-candidate ES cannot capture.

5.4 Cursor Fragments and Prompt Utility

Fewer static resolutions yield four more exact matches. We remove the visible last-line fragment only from DraCo's input, keeping its indentation and all preceding text. The original generator prefix, seven other features, chunk budget, and StarCoder2-3B settings are unchanged. Of 192 CrossCodeEval tasks, 43 lose a static resolution and 24 gain one. The total falls from 96 to 77 (Table 10). The symbol list changes on 105 tasks. The follow-up reconstructs all 192 original prompt hashes before changing the dependency vector. The intervention changes 56 retained prompts and 38 raw outputs. Code EM rises from 64 (33.33%) to 68 (35.42%), with four gains and zero losses.

Table 10: DraCo resolution before and after removing the visible cursor fragment. Each cell is a count and percentage of all 192 CrossCodeEval tasks.

Original	Now resolved	Now unresolved	Total
Resolved	53 (27.60%)	43 (22.40%)	96 (50.00%)
Unresolved	24 (12.50%)	72 (37.50%)	96 (50.00%)
Total	77 (40.10%)	115 (59.90%)	192 (100%)

More static hits did not yield more matches in this intervention. This is a direct counterexample to using the overall resolution rate as a proxy for completion quality. A fragment can activate an eligible signature that pulls a less useful chunk into context. Removing it can shift the resolver to a different line, or can let text and structure decide the ranking on their own. Either path can help or hurt. The operational question is what enters the prompt and whether the generator can use it.

A shared file can support different answers. Figure 4 illustrates two points along the dependency-to-prompt path. For `self.http.`, the Qwen route ranks a chatroom chunk first and emits a `get_user` call. The DraCo route includes a definition of `get_me` at rank eight and completes the reference call. In the Vizier test, the same `core_test.py` chunk appears at ranks four and ten. The completions differ at the method name. A file-level hit can hide differences in the amount, order, or content of the rest of the evidence, so the shared chunk alone cannot account for the flip. The two cases show why a useful confidence check needs finer evidence than a path match.

6 Discussion, Proposed Architecture, and Broader Context

6.1 Retrieval Need, Extraction Need, and Prompt Utility

The cheapest useful evidence. CrossCodeEval and RepoEval reward searching more often than they penalize it, but they offer different opportunities for skipping. The whole-search gate must recognize edits for which the generator has enough evidence from the current file. An extraction policy has a narrower job. Static-First RRF keeps text and structure available, cuts the dependency budget, and gives up one match on each benchmark. On CrossCodeEval, No-Dependency RRF shows how far those seven features can go on their own. The eleven-match loss with Gate + Static-First on RepoEval illustrates what happens when the broader search decision removes evidence that a cheaper extraction route could still use.

The right unit of analysis is the retained prompt. A positive dependency feature measures whether a name matched an eligible signature. The cursor intervention shows why rewarding this event alone can send a system in the wrong direction. Searching or resolving more can change the candidate ranking without improving the evidence that reaches generation. A different snippet can supply a useful definition. Prompts that share a snippet can still yield different API names. Opposing effects on individual suggestions can cancel in aggregate.

Component cost and user delay answer different questions. DraCo lowers dependency-call latency sharply. The warm wall-time advantage is smaller because generation from multi-file prompts remains. Dropping a Qwen call can still improve the tail even when the median is unchanged, as Static-First does on RepoEval. An interactive system should account for the generation cost of the prompt it is about to construct as well as the likely benefit of another lookup. The query-specific ranker's weights tell us how it combines candidates. They do not tell us whether its downstream gain justifies the time.

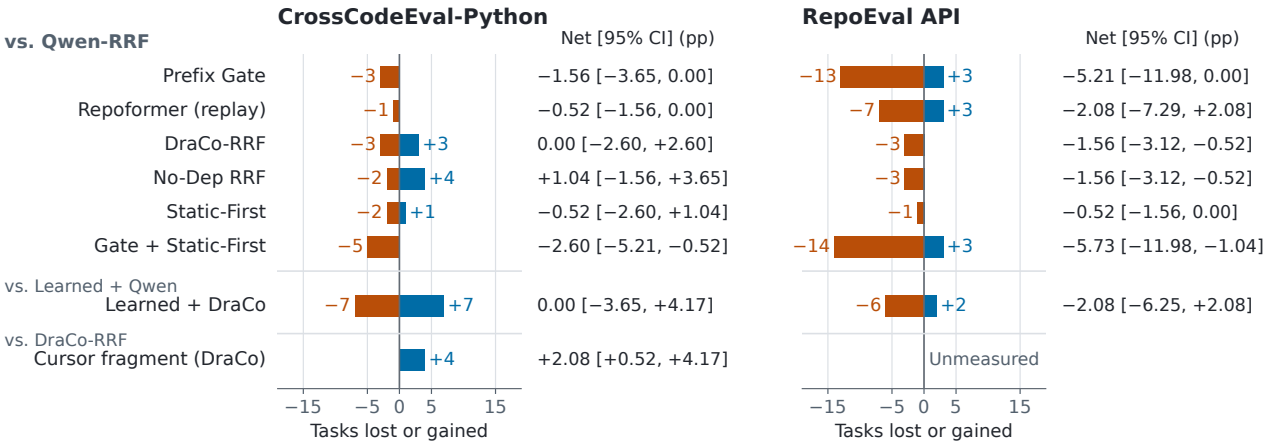


Figure 3: Exact matches gained and lost on the 192-task paired cohorts. The reference is Always Qwen-RRF except for Learned + DraCo (relative to Learned + Qwen) and the cursor-fragment intervention (relative to original Always DraCo-RRF). Delta annotations give net percentage points with a 95% repository-cluster interval. The cursor intervention was evaluated only on CrossCodeEval.

A. CrossCodeEval `kick/client.py`

Visible prefix `data = await self.http.`

Reference suffix `get_me()`

Always Qwen-RRF **EM 0**

Rank 1 in `kick/chatroom.py`

```
class PartialChatroom:
```

Output `get_user(self.http.username)`

Always DraCo-RRF **EM 1**

Rank 8 in `kick/http.py`

```
def get_me(self) -> Response[ClientUserPayload]:
```

Output `get_me()`

B. RepoEval API `feature_mapper_test.py`

Visible prefix `problem = vz.ProblemStatement()`

Reference method `add_int_param('i1', 0, 10)`

Always Qwen-RRF **EM 1**

Rank 4 in `core_test.py`

```
class TrialToArrayConverterTest(absltest.TestCase):
```

Output method `add_int_param('i1', 0, 10)`

Always DraCo-RRF **EM 0**

Same `core_test.py` chunk at rank 10

```
class TrialToArrayConverterTest(absltest.TestCase):
```

Output method `add_integer_param('i1', 0, 10)`

Figure 4: Two measured completions with opposite outcomes after swapping extractors. Ranks describe retained chunks. The two RepoEval outputs share the displayed chunk, which is ranked differently in their respective prompts. Source excerpts are from kick.py (© 2023 cibere, MIT) and Google Vizier (© 2023 Google LLC, Apache-2.0) [8, 23]. The companion site shows the In-file result and full comparison for both.

6.2 A Rank-Shift and LSP-Verified Confidence Router

The measured policies suggest a four-stage architecture (Figure 5). It makes the gate decide whether a search is worth the request budget, then tests whether a static hit adds anything to the cheapest available prompt. The final branch uses a learned fusion module trained on the extractor signals it will actually receive. Table 11 compares the decision points with existing designs.

1. Calibrate the search gate on both sides of the cursor. Let $q = (\ell, r)$ contain the visible prefix and suffix. Use paired development completions to fit $\widehat{u}_{\text{search}}(q)$, an estimate of the quality difference between the cheap retrieved route and the in-file route for that query. Calibration should measure how reliable the estimate is for given budgets and cursor types. One decision rule searches when predicted uplift exceeds $\lambda\,\widehat{t}(q)$, where $\widehat{t}(q)$ is the extra request time and λ sets the time-quality trade-off. A ladder of fixed λ values gives complete budget curves. It also lets a team choose a search rate without changing the downstream scoring definition. The measured suffix sensitivity motivates training with the right-context availability the editor will have.

2. Verify a static binding against the index. Use the Language Server Protocol (LSP) to ask for the definition of a visible receiver or imported symbol and map the returned file and span to an eligible indexed chunk [22, 25]. An existing semantic index such as SCIP can provide definition identities where available. Use exact binding and provenance rather than a loose substring to mark an eligible static hit. Refresh or reject stale identities after edits. The same interface can reuse project services for Python, TypeScript, Java, C#, and Rust. Warm servers and indexes amortize setup. Measure coverage and startup for each provider.

3. Measure changes that survive the prompt budget. Construct P_0 from the seven text and structure signals and P_s from those same

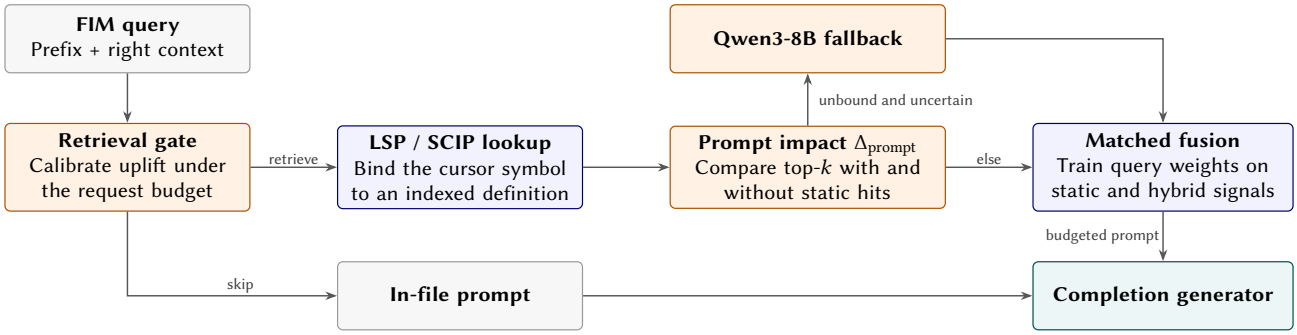


Figure 5: Proposed confidence router. A prefix-and-suffix gate first decides whether to retrieve. On the search branch, a language server or SCIP can bind a cursor symbol to indexed evidence. A rank-shift check compares the budgeted prompts with and without that evidence. Qwen fallback requires both an unbound query and uncertain cheap ranks. The final fusion is trained with its serving extractors.

Table 11: Design comparison. The confidence router is a proposal. Static-First RRF is the measured fixed-fusion policy.

System	When it searches	How it handles dependencies	What determines the final evidence
AIRCoder	Every request	LLM predicts likely names	RRF or query-conditioned weighting of eight inputs
Repoformer	Learned gate before lookup	No separate dependency call in the evaluated lexical path	Retrieved similar snippets if the gate opens
DraCo	Static search from visible code	Python graph and dataflow traversal	Reachable definitions and prompt budget
Static-First RRF	Every request	DraCo signature hit, else Qwen	Fixed RRF over text, structure, and the selected signal
Proposed router	Calibrated FIM uplift under a latency budget	Indexed binding check, with model fallback for unbound and ambiguous queries	Rank shift after packing and extractor-matched learned fusion

signals plus the static dependency vector. Compare retained chunks and order after packing, not just the raw top ranks. For each eligible chunk c, let $a_P(c) = 1/r_P(c)$ at retained rank $r_P(c)$ and zero when absent. An order-sensitive score is the normalized difference

$$\Delta_{\text{prompt}} = \frac{\sum_c |a_{P_s}(c) - a_{P_0}(c)|}{\sum_c \max\{a_{P_s}(c), a_{P_0}(c)\}}. \tag{6}$$

Set the score to zero if both prompts retain nothing. A zero implies the same ordered chunks under a fixed prompt builder. A high score identifies a change for the generator to use or ignore. For a verified binding, use Δ_{prompt} and the binding evidence to decide whether to pass the static or zero vector to final fusion. The probe prompts estimate impact. Calibrate the decision on completions from the served ranker.

An unbound query has an all-zero static vector, which forces $\Delta_{\text{prompt}} = 0$. The fallback decision therefore needs a separate cheap ambiguity signal. For example, measure how much the text and structure lists disagree about candidates near the retention cutoff, and use the rank margin around that cutoff as a confidence feature. Call Qwen only when the query has no eligible binding *and* the cheap ranking is sufficiently uncertain that an additional signal could plausibly change what fits. For a tight and consistent ranking, use the zero vector. Training labels can come from paired cheap and Qwen prompts on development repositories, with benefit assessed after packing and generation. The rank-shift score measures available static evidence. The ambiguity estimate addresses the hypothetical effect of an extraction that has yet to run.

4. Train fusion on the signals used at serving time. Fit query-conditioned weights on static, Qwen, and hybrid dependency inputs from the intended serving policy. Include all-zero vectors and verified hits that fail to survive packing. Sample alternative retained prompts from the same queries, then add supervision from generation on those prompts. This tests whether a defined API helped because the generator saw it, and whether the same answer already followed from text and structure. Select the ranker and thresholds on development repositories and evaluate once on new repositories. The primary outputs should be paired quality, call budget, prompt changes, and warm and cold request latency at matched budgets.

6.3 Long Context, Coding Agents, and FIM Editing

A wider window still needs context selection. Even a 128K- or 1M-token window can leave a useful example buried far from the cursor. Long Code Arena evaluates models on project-level tasks where this problem is exposed [5]. Lost in the Middle shows that access to a long input can be sensitive to where relevant evidence appears [18]. Its results concern different tasks, but motivate testing placement explicitly for code. In the warm experiment, moving from in-file prompts to retrieved prompts already adds about nine tenths

of a second to median generation at the measured model size. A larger input limit alone cannot control that cost. A short, well-placed definition or call site can be the better use of the editor's time.

Inline completion and coding agents. SWE-bench evaluates repairs to real repository issues [13]. SWE-agent explores files, edits code, and runs tools iteratively [34]. Those tool calls are part of a longer task with a persistent objective. An inline completion is a single edit on a moving cursor and can go stale before the next keystroke. Its retrieval loop needs to be cheaper, and a definition lookup already maintained by the editor is especially attractive. The two settings share a need for grounded information but put different prices on acquiring it.

Executable outcomes can improve the training target. A FIM benchmark can preserve the existing right side of an edit and measure whether the filled file parses and binds to the intended library or project symbol [4]. DevEval shows how repository-level evaluation can use existing tests [15]. Combining these ideas would let a benchmark record reference agreement, valid alternatives, and failures that differ by only one identifier. The gate could then optimize a utility closer to what the editor needs, such as the probability of a type-correct edit under a time budget.

6.4 Threats to Validity and Reproducibility

Task sampling and transfer. The study covers one completion model and Python line or API tasks. RepoEval contributes eight repository clusters, with development and test separated by task ID. Its intervals describe variation among the sampled repositories and offer weaker evidence about new projects than the held-out CrossCodeEval design. The reduced learned checkpoints use regenerated inputs and much smaller splits than the seed paper. They test extractor substitution with fixed, Qwen-trained weights, so they cannot isolate the benefit of retraining for the static distribution. The No-Dependency CrossCodeEval interval contains zero. The measured difference motivates paired follow-up across more projects. It establishes the ordering of the observed sample.

Prompt editions. The official CodeGen and Codex RepoEval API editions share 1,600 task IDs, but 597 effective model prompts (37.31%) differ after truncation. In the full in-file runs, 170 outputs differ among those 597 tasks. Five become exact and four cease to be exact. Among the 1,003 unchanged prompts, 94 outputs differ without an exact-match flip. The two passes used different batching and GPU allocation, so output differences cannot all be attributed to the prompt. Among the ten released AIRCoder RepoEval examples, the effective prompt agrees with the Codex edition in all ten and with CodeGen in four. The paired comparison uses CodeGen throughout. The benchmark name by itself is insufficient to identify the effective input.

Verbatim targets in other files. A post-scoring audit finds the normalized target string in an actual retained other-file chunk on 36 of 192 RepoEval tasks (18.75%) and on no CrossCodeEval task. Excluding the same 36 tasks from every audited arm leaves 44 Qwen-RRF matches (28.21%) and 38 In-file matches (24.36%) out of 156. The raw gain falls from 11.98 to 3.85 pp. The full paired cohort remains primary. These repeats can reflect genuine patterns in the project, and the filter misses paraphrases or semantically equivalent examples. The audit measures literal exposure in the retained prompt. It does not establish pretraining leakage or assign a causal fraction of the gain to copying.

Serving conditions and gates. The Repoformer experiments combine offline gate decisions with previously generated outcomes. Its prefill time describes the gate alone. A live integration must also budget branch execution and any competition for accelerator capacity. The warm serial timing excludes model loading, indexing, concurrency, and queueing. Counts of calls therefore describe a different resource from request p50 or p90. Uniform skipping gives a descriptive expectation for the chosen budget. The compared gates may use different budgets and their rankings can change at matched operating points.

What the scorer can see. The released scorer and Strict EM agree for these paired results, but a short scoring span can materially increase the apparent success rate. On RepoEval, scoring only the first line gives Always Qwen-RRF 44.79% instead of 37.50% and In-file 32.81% instead of 25.52%. The main scores compare the reference span, including any later lines required by the target. They cannot show whether an alternative completion compiles, passes tests, or would be accepted by a developer. Cases with reusable unit tests and a held-out FIM suffix offer a route to more direct quality measures.

7 Conclusion

AIRCoder demonstrates that repository-level completion can use text, dependencies, and structure as complementary clues. The paired interventions show why those clues should carry separate decisions. A search can be valuable when a second model call adds little. Static-First RRF kept repository context while saving half the Qwen calls on CrossCodeEval and about a fifth on RepoEval, with one fewer exact match in each cohort. Removing the dependency feature altogether produced the best observed fixed-RRF Code EM on CrossCodeEval. The prefix gate made larger RepoEval call savings at a larger quality cost, close to what uniform search skipping would yield.

A static hit gains value when it improves the context the model actually sees. Removing a cursor fragment led to fewer resolutions and more exact matches, and the learned substitution changed fourteen CrossCodeEval exact outcomes despite tying in aggregate. Timing the whole request completes the picture. Swift dependency extraction leaves most of the cost of generating from a retrieved prompt. A confidence router can use a verified binding, a cheap ranking, and the resulting prompt to decide which cost is worth paying. The next question is whether that router can improve executable completion quality at a fixed end-to-end latency budget on repositories it has never seen.

References

[1] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Advances in Neural Information Processing Systems*, Vol. 36. Neural Information Processing Systems Foundation, New Orleans, LA, USA, 32270–32298. doi:10.52202/075280-1401

[2] Amazon Science. 2025. CrossCodeEval benchmark release. GitHub repository, pinned revision `40c68d2b7ca2`. https://github.com/amazon-science/cceval/tree/40c68d2b7ca2a8eae95d901ac80e6a540a84a53d

[3] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. In *Proceedings of the 12th International Conference on Learning Representations*. OpenReview.net, Vienna, Austria, 9112–9141. https://proceedings.iclr.cc/paper_files/paper/2024/hash/25f7be9694d7b32d5cc670927b8091e1-Abstract-Conference.html

[4] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient Training of Language Models to Fill in the Middle. arXiv preprint arXiv:2207.14255. doi:10.48550/arXiv.2207.14255

[5] Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. 2024. Long Code Arena: a Set of Benchmarks for Long-Context Code Models. arXiv preprint arXiv:2406.11612. doi:10.48550/arXiv.2406.11612

[6] 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. 2021. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374. doi:10.48550/arXiv.2107.03374

[7] Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Bangkok, Thailand, 7957–7977. doi:10.18653/v1/2024.acl-long.431

[8] cibere. 2023. kick.py source. GitHub repository, MIT license. https://github.com/cibere/kick.py/tree/03541c0732ad85248a294fb3e96ae617688482f4

[9] 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*. Association for Computing Machinery, New York, NY, USA, 758–759. doi:10.1145/1571941.1572114

[10] 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. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In *Advances in Neural Information Processing Systems*, Vol. 36. Neural Information Processing Systems Foundation, New Orleans, LA, USA, 46701–46723. doi:10.52202/075280-2023 Datasets and Benchmarks Track.

[11] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code Completion by Jointly Modeling In-file and Cross-file Context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. ELRA and ICCL, Torino, Italia, 3433–3445. https://aclanthology.org/2024.lrec-main.305/

[12] 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 (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. doi:10.18653/v1/2022.acl-long.499

[13] 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 *Proceedings of the 12th International Conference on Learning Representations*. OpenReview.net, Vienna, Austria, 54107–54157. https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html

[14] 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. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., Red Hook, NY, USA, 9459–9474. https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[15] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, Bangkok, Thailand, 3603–3614. doi:10.18653/v1/2024.findings-acl.214

[16] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! Transactions on Machine Learning Research. https://openreview.net/forum?id=KoFOg41haE

[17] Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Wei Jiang, Chengpeng Wang, Gang Fan, and Peng Di. 2026. RepoFuse: A Dual-Context Approach to Repository-Level Code Completion at Industrial Scale. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 568–579. doi:10.1145/3803437.3805230

[18] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[19] Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *Proceedings of the 12th International Conference on Learning Representations*. OpenReview.net, Vienna, Austria, 47832–47850. https://proceedings.iclr.cc/paper_files/paper/2024/hash/d191ba4c8923ed8fd8935b7c98658b5f-Abstract-Conference.html

[20] Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-fine Retrieval Based on Code Context Graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 570–581. doi:10.1145/3691620.3695054

[21] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv preprint arXiv:2402.19173. doi:10.48550/arXiv.2402.19173

[22] Microsoft. 2022. Language Server Protocol Specification, Version 3.17. Protocol specification, Go to Definition. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_definition

[23] Microsoft. 2023. RepoEval benchmark and repository snapshots. CodeT GitHub repository, pinned revision `35f54d60b152`. https://github.com/microsoft/CodeT/tree/35f54d60b152cc31d134b788e702878ad613d9f7/RepoCoder

[24] Huy N. Phan, Hoang N. Phan, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. RepoHyper: Search-Expand-Refine on Semantic Graphs for Repository-Level Code Completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, Piscataway, NJ, USA, 14–25. doi:10.1109/FORGE66646.2025.00009

[25] SCIP contributors. 2026. SCIP Code Intelligence Protocol. Protocol schema, version 0.10.0. https://github.com/scip-code/scip/blob/v0.10.0/scip.proto

[26] 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. doi:10.18653/v1/2026.acl-long.1166

[27] Chuanqi Shi, Miao Gao, and Zhiqiang Gao. 2026. AIRCoder implementation. GitHub repository, pinned revision `c3c344d674ba`. https://github.com/chuanky/AIRCoder/tree/c3c344d674ba4c913c173c1c63c49309a49b1bba

[28] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-Level Prompt Generation for Large Language Models of Code. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, Honolulu, HI, USA, 31693–31715. https://proceedings.mlr.press/v202/shrivastava23a.html

[29] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025. RLCoder: Reinforcement Learning for Repository-Level Code Completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, Piscataway, NJ, USA, 1140–1152. doi:10.1109/ICSE55347.2025.00014

[30] Di Wu. 2024. Repoformer-1B personal reproduction checkpoint. Hugging Face, pinned revision `1e074914a656`. https://huggingface.co/xiaowu0162/repoformer-1b/tree/1e074914a656c1d93fc3a4dbdee2a96267eaa48d

[31] 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, Vienna, Austria, 53270–53290. https://proceedings.mlr.press/v235/wu24a.html

[32] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2025. Repoformer implementation. GitHub repository, pinned revision `5b0571318e99`. https://github.com/amazon-science/Repoformer/tree/5b0571318e9918fd2af132c4b35077f9ea331133

[33] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv preprint arXiv:2505.09388. doi:10.48550/arXiv.2505.09388

[34] 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*, Vol. 37. Neural Information Processing Systems Foundation, Vancouver, BC, Canada, 50528–50652. doi:10.52202/079017-1601

[35] 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*. Association for Computational Linguistics, Singapore, 2471–2484. doi:10.18653/v1/2023.emnlp-main.151

[36] Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. 2025. CodeRAG: Finding Relevant and Necessary Knowledge for Retrieval-Augmented Repository-Level Code Completion. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Suzhou, China, 23278–23288. doi:10.18653/v1/2025.emnlp-main.1187

Declaration on the Use of AI

AI assistance in the original literature review covered grammar and ACM LaTeX formatting. For this report and its companion site, OpenAI Codex generated and revised experiment and visualization code, audited source and bibliographic metadata, checked results against frozen artifacts, drafted and substantially rewrote manuscript text, generated figures, and revised the website. The results use the recorded experiments and derived checks.