VITAL-RAG: OBJECT-LEVEL CONTEXT ALLOCATION FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Repository-level coding agents depend on retrieved code, but relevant evidence may still fail to reach the coding model's input. Multiple views of one object can consume several context positions, while sequential packing can discard evidence from objects already selected. We study this retrieval-to-context gap and introduce VITAL-RAG, an object-level context-allocation system for coding agents. Controlled comparisons on 16,490 RepoBench tasks show that object grouping provides a substantial structural gain: full-source evidence containment increases from 24.99% to 39.16%. With candidates fixed task by task, the system's rendering configuration raises complete rendered-unit availability from 39.59% to 63.67% while using 35.63% fewer evidence tokens. Packing gains grow from 1.29 to 14.13 points as the candidate limit increases from five to twenty on a disjoint set of 14,442 tasks. Under a shared renderer and a twelve-object limit, cost-aware selection improves rendered-unit availability by 3.58 points over rank-only selection on 13,121 held-out tasks. A complete two-candidate RepoExec workflow using VITAL-RAG context construction reaches 65.63% Pass@1 with Claude versus 54.65% for the strongest local style control evaluated here; this complete-workflow result includes post-generation output selection. A 100-task controlled generation probe does not establish a statistically reliable code-accuracy gain from allocation alone. Together, these results separate evidence discovery, evidence delivery, and generated-code correctness in repository-level coding agents.

1 INTRODUCTION

Repository-level coding tasks require evidence distributed across definitions, interfaces, imports, and helper routines (Liu et al., 2024b; Ding et al., 2023; Jimenez et al., 2024). Retrieval identifies potentially useful code, but the model receives only the context constructed from it. A relevant helper can therefore be retrieved and still disappear from the final input when earlier snippets consume the available tokens.

This gap arises at two distinct stages. First, the allocation unit matters: a body, a signature, and a compact lexical view may describe the same object but compete as three candidates. Second, selected objects still compete for tokens. Even after redundant views are grouped, their ordering and rendering determine whether useful evidence survives. Figure 1 illustrates why improving retrieval alone does not resolve either problem.

The distinction matters for evaluating repository assistants. Retrieval scores characterize candidates before prompt construction, while selected recall only records whether a gold candidate is admitted. Neither guarantees that its defining source span appears in the final prompt. A downstream model can fail even when candidate discovery succeeds, and an apparent retrieval improvement can be hidden by prompt construction. We therefore measure both candidate admission and the evidence actually emitted under a fixed budget, and evaluate generation accuracy separately.

We investigate these stages in sequence. A controlled comparison using the same retrieval scorer on 16,490 tasks shows that object grouping raises full-source evidence containment by 14.17 percentage points. With those objects fixed, completeness-first packing provides a further 1.28-point gain. Adding a query-linked companion yields a smaller additional gain. However, the effect of packing grows substantially with candidate count: on a disjoint set of 14,442 tasks, its gain reaches 14.13 points when the candidate limit reaches twenty. Thus object grouping addresses representation

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

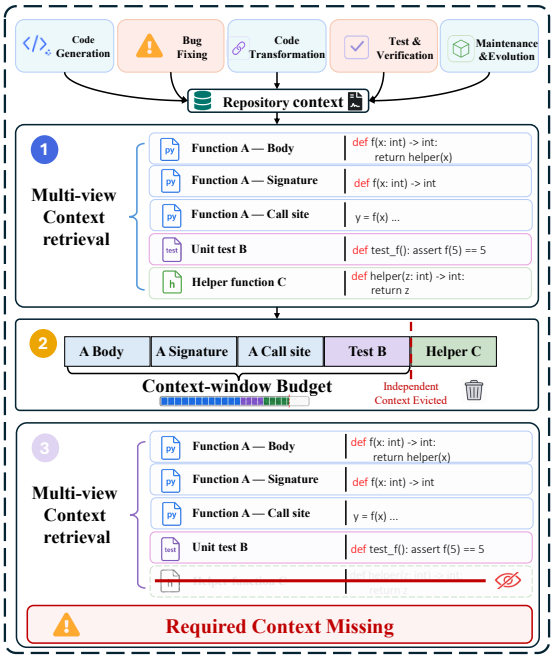


Figure 1: The retrieval-to-context gap. Repeated views occupy positions and tokens that could preserve an independent helper. The schematic illustrates the failure; Appendix A.3 specifies the evaluated view construction.

redundancy, while allocation becomes increasingly consequential as more objects compete for the same context budget.

These observations motivate VITAL-RAG. It groups views into code objects, then selects and orders them using relevance, file coverage, and estimated token cost before constructing bounded context. When the selected candidates are held fixed for every task, its bounded rendering raises evidence availability from 39.59% to 63.67% while reducing mean tokens by 35.63%. At a twelve-object limit, cost-aware selection adds 3.58 points over rank-only selection on 13,121 held-out tasks at nearly unchanged selected recall. An end-to-end two-candidate workflow also achieves higher executable-task success than the evaluated local controls, but includes post-generation output selection. These comparisons locate the gains at different stages of the same repository coding system.

Our primary contribution is an empirical decomposition of the post-retrieval gap between discovered repository evidence and the evidence actually delivered to a coding model. The diagnosis separates grouping from packing; the fixed-candidate and held-out comparisons test two system components; executable tasks assess the complete workflow. We report these stages with their respective metrics because evidence retention and code correctness are distinct outcomes.

The paper follows this progression. Section 2 defines the allocation problem and measurements. Section 3 isolates the two sources of loss and their scale dependence. Section 4 derives the allocation policy from these observations, and Section 5 tests fixed-candidate rendering, cost-aware selection, and the complete workflow. The appendices supply implementation details, full statistics, and task-specific protocols.

2 The Context-Allocation Problem

2.1 From Retrieved Views to Model Input

For a query q, let $\mathcal{V}_q$ be the ranked retrieved views, $\mathcal{E}_q$ their canonical code objects, $\mathcal{A}_q$ the ordered selected objects, and C_q the resulting context. The evidence path is

$$\mathcal{V}_q \longrightarrow \mathcal{E}_q \longrightarrow \mathcal{A}_q \longrightarrow C_q \longrightarrow y_q,$$

where y_q is generated code. Retrieval supplies views; grouping identifies which views refer to the same object; selection assigns object positions; context construction assigns their realized token use. We study the middle stages under a global evidence budget B, an object-count limit K, and a per-object cap. Correct generation is a separate outcome at the end of this path.

This formulation gives two controlled questions. Does grouping views improve what enters the context? Once the candidate objects are fixed, does packing change how much evidence survives? Section 3 answers these before introducing the cost-aware policy.

An allocation can fail at either boundary without a retrieval error. Suppose two highly ranked surfaces describe the same helper. If selection spends two of five positions on them, a distinct dependency may remain outside the selected set. Alternatively, suppose all relevant objects are selected, but a sequential renderer spends most of the budget on early bodies. A later object's signature might be written while the source span needed to use it is omitted. The first failure is a coverage problem over objects; the second is a feasibility problem over tokenized evidence. Their remedies need not be the same, which is why the comparisons below change one boundary at a time.

2.2 Measure Selection and Evidence Retention

Selected recall records whether the labeled gold candidate enters $\mathcal{A}_q$. Evidence retention additionally examines what is written into C_q. We report full-source containment for the grouping and packing diagnosis: the complete original gold snippet must occur in the context after line-ending normalization and trimming. We report complete rendered-unit availability for the shared-renderer policy comparison: a snippet-derived unit from the selected gold candidate must be emitted without further truncation.

These measurements track different points in context construction; a rendered unit may be a query-centered local window. We retain the metric attached to each experiment and make comparisons within its stated protocol. Appendix A.2 gives the exact matching definitions. Generation is evaluated separately by output matching and repository tests.

The distinction prevents two misleading inferences. A higher selected recall can coexist with lower availability when admission creates more token competition. Conversely, higher availability need not imply higher code accuracy: the generator must attend to the supplied unit and use it correctly. Thus the controlled evidence tests a necessary intermediate step, not an end-to-end performance guarantee. The five- and twelve-object policies in Section 5.2 share a renderer, making the comparison of selected recall and availability particularly informative within that protocol.

2.3 Evaluation Scope

The primary diagnosis uses all 16,490 eligible RepoBench v1.1 cross-file-first tasks. Its candidate-scale confirmation uses 14,442 tasks outside a 2,048-task exploratory subset. The policy evaluation uses a separate repository-based split with 13,121 held-out tasks and settings fixed before test evaluation. Unless varied explicitly, evidence budgets are 4,096 tokens with a 1,024-token object cap.

Strict-containment contrasts use 10,000 repository-cluster bootstrap resamples. The policy comparison also reports repository-cluster intervals. Task populations, token accounting, and the relation between these splits are detailed in Appendix A.1. The same-scorer decomposition, candidate-scale comparison, and held-out policy evaluation serve different questions rather than forming a single pooled leaderboard.

Two splits deliberately answer different threats to validity. The complement of the exploratory tasks checks whether the candidate-scale pattern survives an untouched portion of the same benchmark. The repository-held-out split checks a fixed policy on repositories not used to choose its settings. Neither split turns the other experiment's outcome into a directly comparable percentage: the former tests full original-source containment under alternative packers, while the latter tests complete rendering-unit availability under one shared renderer. This separation is maintained throughout the tables and discussion.

Selection and packing	Containment (%)	Gain (pp)	95% CI (pp)
Views, sequential	24.99	Reference	–
Objects, sequential	39.16	+14.17	$[13.47, 14.87]$
Objects + companion, sequential	39.36	+0.20	$[0.04, 0.36]$
Objects, completeness-first	40.44	+1.28	$[1.09, 1.47]$
Objects + companion, completeness-first	40.69	+0.25	$[0.07, 0.43]$

Table 1: Separating grouping, packing, and companion selection on 16,490 tasks. All rates measure complete original-snippet containment. Grouping compares rows 2 and 1; packing compares rows 4 and 2; companion gains compare rows 3 and 2, and rows 5 and 4. Appendix A.3 defines the controls.

3 Where Retrieved Evidence Is Lost

Before designing an allocation policy, we isolate its target. The following comparisons keep the retrieval scorer and evidence budget fixed, first changing the unit of selection and then changing how selected objects are packed. This order distinguishes coverage gained through grouping from evidence retained after selection.

3.1 Grouping and Packing Address Different Losses

The first comparison asks whether repeated views crowd out independent objects. The second asks whether those objects, once selected, retain their evidence under the token budget. Table 1 evaluates both with the same full-source containment rule.

Grouping improves containment from 24.99% to 39.16%, a 14.17-point gain with interval $[13.47, 14.87]$. This is the principal structural effect: views of one object no longer consume several selection positions. Holding the five selected objects fixed, completeness-first packing adds 1.28 points $[1.09, 1.47]$. The latter gain isolates context construction as a further source of loss after object selection.

These two increments are not interchangeable. Grouping recovers an opportunity for a different object to enter the prompt; packing recovers source text from objects that have already won a position. In particular, the packing contrast keeps the selected candidates fixed task by task, so it can be attributed to how those candidates are rendered rather than to a different retrieval ranking. This is the appropriate local attribution for the five-object setting, not a claim that the same packing rule dominates at every candidate count.

A query-linked companion adds 0.20 points $[0.04, 0.36]$ with sequential packing and 0.25 points $[0.07, 0.43]$ with completeness-first packing. Its small positive effect supports local refinement alongside the larger grouping benefit. These comparisons suggest preserving object-level coverage while remaining flexible about nearby evidence, rather than making a fixed 4+1 partition central to the method.

Appendix B.3 checks the grouping mechanism from another direction: adding exact duplicate views changes the selected set less often after object grouping, although BM25 scoring remains sensitive to input multiplicity. The next question is whether the modest packing gain at five objects understates the problem at larger candidate counts.

3.2 Token Competition Grows with Candidate Count

We answer that question on 14,442 tasks excluded from the exploratory analysis, holding the global budget at 4,096 tokens and the object cap at 1,024. Figure 2 varies only the object limit and packing rule.

At five objects, sequential and completeness-first packing reach 39.03% and 40.31% containment; the paired gain is 1.29 points $[1.08, 1.50]$. At ten objects, the rates are 45.61% and 56.40%, a 10.79-point gain $[10.11, 11.47]$. At twenty objects, they are 49.88% and 64.01%, a 14.13-point gain $[13.23, 14.98]$. Contrasts use unrounded rates; Appendix B.1 reports the full table.

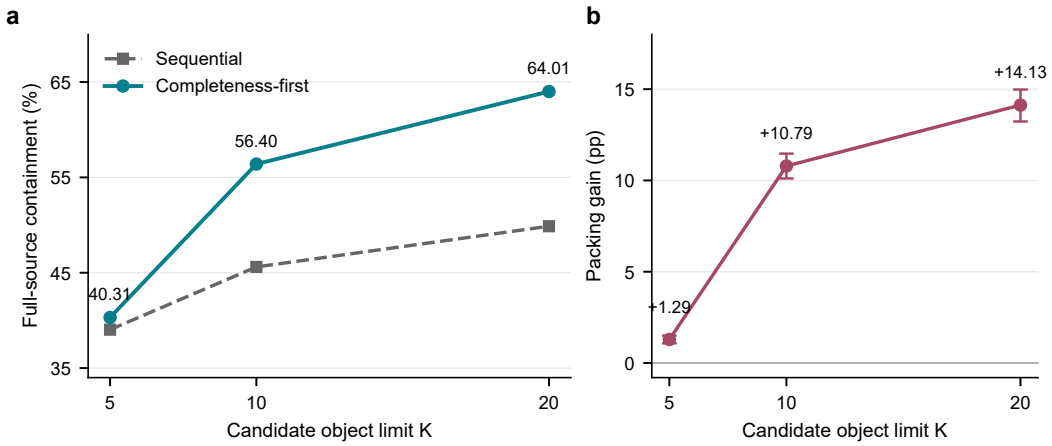


Figure 2: Candidate-scale confirmation on 14,442 tasks disjoint from the exploratory subset. Left: full-source containment with sequential and completeness-first packing at fixed $B = 4096$ and object cap 1,024. Right: paired gains with repository-cluster 95% confidence intervals. Appendix B.1 provides the numerical table. At $K = 20$, rank-valued exact knapsack and shortest-first reach 65.50% and 65.78%, respectively, above completeness-first's 64.01%.

The growing gain establishes that allocation becomes consequential precisely when a larger candidate set offers more evidence than the context can retain. Rank-valued exact knapsack and shortest-first achieve 65.50% and 65.78% at twenty objects, respectively. Thus useful simple alternatives also address this bottleneck; the evidence supports optimizing allocation, rather than privileging one packing rule. Appendix B.2 examines the complementary budget dimension and shows that larger global windows help when object caps also increase.

The pattern is not explained by a change in tokenizer or total allowance: both are fixed while the number of eligible objects increases. More candidates improve the chance of selecting the source, but they also create more ways for an early long object to consume the shared budget. At $K = 20$, the 14.13-point packing contrast is more than ten times its $K = 5$ value. This is an empirical consequence of the stated portfolio and renderer, not a universal scaling law; nevertheless it identifies a regime where treating retrieval rank as a prompt order is particularly costly.

Global budget and local caps interact. With five canonical objects, expanding only the global allowance leaves full-source containment near 40.45%, whereas scaling the per-object cap with it raises containment to 56.26% at 16K. The flat fixed-cap curve diagnoses a local bottleneck; it does not show that larger context windows are intrinsically unhelpful. Appendix B.2 gives the full comparison.

Repeated views have a second effect before selection. On 2,048 hash-selected tasks, adding exact copies changes the top-k object set in 14.50% of tasks when raw views compete directly, versus 3.76% when BM25 scores are collapsed to an object maximum. Grouping protects selection positions, but it is not exactly copy-invariant because BM25 document frequencies were computed before grouping. Pre-scoring exact deduplication yields zero set and order changes in this perturbation (Appendix B.3). The main grouping gain and this stress test are complementary: one measures evidence retained, the other checks sensitivity to multiplicity of equivalent surfaces.

3.3 Implications for Allocation

Together, these results motivate two requirements: assign positions to distinct objects, and account for token competition when choosing and ordering them. Relevance alone determines which objects look useful; estimated cost and source coverage help decide which combination is practical within the context. The next section implements these requirements in a no-gold policy, whose effect is then tested with context construction held fixed.

4 VITAL-RAG

The diagnosis in Section 3 suggests treating object identity and token cost as separate allocation concerns. VITAL-RAG follows this structure: merge repeated views, select an ordered set of objects, and construct context under explicit token limits. It uses only the query and retrieved candidates, without gold evidence or generation outcomes.

4.1 Allocate Positions to Objects

Each object receives one representative and the score of its strongest retrieved view. This preserves the object-level coverage benefit in Section 3.1 while allowing a larger candidate pool than the original five-object setting. For the evaluated policy, the first 50 distinct candidate IDs are merged by normalized path and identifier, and at most twelve objects are selected. Appendix A.3 gives the view construction, identity keys, and control definitions.

The grouping key represents a pragmatic repository object: it prevents multiple surfaces with the same file and identifier from spending several positions, while leaving distinct identifiers in one file eligible. This is deliberately narrower than semantic equivalence across files or aliases. The policy relies on the existing ranked candidate pool rather than changing the retriever, so the later rank-only comparison asks about allocation of discovered objects, not a newly learned retrieval model.

4.2 Balance Relevance, Coverage, and Cost

Selecting more objects creates the token competition measured in Section 3.2. We therefore rank candidates by a utility that combines relevance with file novelty and estimated rendering cost. Let a_i be relevance, $\widehat{c}_i$ estimated tokens, and S the objects selected so far. The next object maximizes

$$u_i(S) = \frac{\nu_i(S) a_i}{(\max(32, \widehat{c}_i)/256)^{0.5}}, \qquad \nu_i(S) = \begin{cases} 1.00, & \text{path}_i \notin P(S), \\ 0.25, & \text{path}_i \in P(S), \end{cases} \tag{1}$$

where $P(S)$ contains the files already represented. Relevance combines retrieval rank, retrieval score, and lexical query overlap. Cost estimates include the object's descriptor and a capped snippet length. Their exact definitions and fixed coefficients appear in Appendix A.4.

After each selection, file novelty is recomputed for the remaining objects. The same-file penalty is soft: a relevant companion can still be selected, but it must compete on utility rather than receive a reserved slot. Thus the modest companion benefit in Section 3.1 motivates retaining local flexibility without making a fixed 4+1 partition the method's organizing rule. The policy's combined effect is tested in Section 5.2; that comparison does not separately attribute gains to individual utility terms.

Each term responds to a measured pressure. Retrieval rank and lexical overlap keep the choice anchored to likely relevance; the novelty factor avoids filling the portfolio with one source file; the square-root cost denominator discourages expensive objects without automatically preferring only short snippets. We use a greedy ranking rule for efficient sequential object allocation. It can select a costly object when its relevance and file coverage warrant the cost, and its actual token use is determined later by the renderer. All coefficients are fixed before evaluation on the held-out split. We do not tune them on the test set.

4.3 Construct Bounded Context

The estimated cost guides selection and ordering; a shared context-construction procedure enforces the actual token budget. In selection order, it writes the object's path, identifier, and signature, followed by a query-centered snippet window. The evaluated setting allows at most 4,096 `cl100k_base` tokens overall and 1,024 per object, stopping when the global limit is reached. Appendix A.5 specifies this procedure and distinguishes estimated costs from enforced limits.

This separates two interventions. Section 3.2 holds candidate selection fixed and changes packing to diagnose context loss. The next section instead holds context construction fixed and changes object selection and ordering. The latter comparison tests whether the proposed policy can improve evidence availability beyond simply increasing the object limit.

This difference is also why the policy should not be described as dynamically assigning exact token shares to all twelve objects. Its explicit cost estimate influences which objects enter and in what order; the fixed renderer then writes descriptors and bounded query-centered windows sequentially. Because it can exhaust the shared allowance before the last object, ordering remains consequential even when the same number of candidates is selected. Joint optimization of token shares is outside the scope of the present comparison; the experiment instead asks whether cost-aware selection improves the present shared procedure.

5 Evaluating Cost-Aware Allocation

Section 3 established that object grouping and packing matter. We evaluate the system at three points: evidence retained with fixed candidates, the added value of cost-aware object selection, and code produced by a complete repository workflow. The first two tests isolate context mechanisms; the last includes generation and output selection.

5.1 Evidence Retention with Fixed Candidates

Table 2 intentionally uses a different evidence unit from Table 1: its absolute availability values and gains are therefore not directly comparable with the full-source containment results in Section 3. It compares two rendering configurations on the 16,490 eligible RepoBench tasks. Candidates are identical task by task, so the 24.08-point availability increase isolates context construction. Mean evidence length falls from 2,720 to 1,751 tokens, with gains in both Java and Python. Appendix D.1 specifies the five-object controls.

Tasks	**Base (%)**	**Full (%)**	**Gain (pp)**	**Tokens B/F**
All (16,490)	39.59	63.67	+24.08	2,720/1,751
Java (8,556)	37.60	63.69	+26.09	2,798/1,765
Python (7,934)	41.73	63.65	+21.92	2,637/1,736

Table 2: Fixed-candidate rendering comparison on RepoBench. Rates are complete rendered-unit availability at 4K, not full-source containment. Base and Full select identical candidates task by task; mean evidence tokens fall by 35.63% overall (unrounded means).

5.2 Availability Beyond Candidate Recall

The comparison fixes the initial retrieved pool, token limits, and context-construction procedure. A five-object configuration provides a reference point. Rank-only selection expands the portfolio to twelve objects; VITAL-RAG uses the same limit with cost-aware selection and ordering. Settings were fixed before evaluating the 13,121-task repository-held-out split (Appendix A.1).

Policy	**Selected recall (%)**	**Availability (%)**	**Mean tokens**
Five-object reference	64.27	63.72	1,743
Rank-only ($K = 12$)	88.27	82.47	2,523
VITAL-RAG ($K = 12$)	88.42	86.05	2,266

Table 3: Object-selection policies on 13,121 repository-held-out tasks. All share the initial top-50 candidate pool and context-construction procedure. Availability refers to complete rendered units at a 4K evidence budget (Appendix A.2); object limits distinguish the five-object reference from the two twelve-object conditions.

Expanding the object limit substantially increases selected recall, from 64.27% to 88.27% for rank-only selection. The critical comparison then holds that limit fixed. VITAL-RAG changes selected recall by only 0.145 points, with interval $[-0.424, 0.704]$, while increasing availability by 3.58 points $[2.92, 4.27]$. The paired availability outcomes contain 1,131 wins and 661 losses. Mean evidence tokens fall from 2,523 to 2,266.

This is the policy-level answer to the diagnosis: two selections with nearly the same aggregate ability to admit gold candidates can differ materially in what reaches the model input. Because the renderer is shared, the gain belongs to the combined selection-and-ordering policy. The comparison does not isolate each relevance, novelty, or cost term separately.

The availability improvement remains 3.72 points $[3.05, 4.44]$ after excluding all pilot tasks. It is approximately 3.76 points for Python and 3.42 for Java. Appendix B.4 collects these contrasts and explains their relation to the primary result. Each comparison uses its own stated evidence metric; the full matching definitions remain in Appendix A.2.

5.3 A Controlled Generation Probe

This controlled probe tests whether higher evidence availability translates into higher generation accuracy. On 100 Python tasks not selected by outcome, the same Qwen3-8B Q4 backend, prompt shell, and temperature-zero setting produce 8% Exact with the five-object reference, 6% with rank-only twelve-object selection, and 10% with VITAL-RAG. The latter's 2-point advantage over the reference has paired interval $[-2, +6]$. Allocation changes model outputs, but this interval does not establish a statistically reliable allocator-only gain. We therefore treat evidence availability as an intermediate systems metric rather than a surrogate for code correctness. Appendix C gives the full result and newly available-evidence subset.

5.4 Complete Workflow on Executable Tasks

The RepoExec comparison evaluates the complete generation-and-selection workflow, not the allocator in isolation. The workflow generates two candidates and applies the static selector described in Appendix E.5. Table 4 reports this workflow on 355 RepoExec tasks per backend. Its selected output has the highest observed Pass@1 among the evaluated local controls for each backend. For Claude, the paired gain over the strongest control is 11.05 points, with 47 wins, 8 losses, and 95% interval $[7.08, 15.01]$ on 353 paired tasks. GPT and Qwen paired intervals include zero (Appendix E.6). Control construction is specified in Appendix E.

Method	**GPT-5.4**	**Claude 4.6**	**Qwen3-8B**
CodeRAG-style	52.11	53.24	14.93
GraphCoder-style	52.39	54.65	21.13
RepoScope-style	55.49	53.80	15.21
VITAL-RAG workflow (2 candidates + selector)	**57.75**	**65.63**	**21.69**

Table 4: RepoExec Pass@1 (%) over 355 tasks per backend. VITAL-RAG uses two generated candidates and a static output selector; comparators are local style controls. Only the Claude paired gain over its strongest control has a confidence interval excluding zero. This comparison evaluates the complete generation-and-selection workflow and should not be interpreted as an isolated estimate of the twelve-object allocation policy. Appendix E.2 defines the controls.

Together, these results trace the system from object coverage to retained evidence and tested code. The fixed-candidate rendering and held-out selection comparisons directly support context allocation; the RepoExec gain is a complete-workflow result and should not be attributed to the allocator alone.

6 Related Work

Repository evidence and retrieval. RepoBench and CrossCodeEval make cross-file completion measurable by requiring code outside the immediate file (Liu et al., 2024b; Ding et al., 2023). Repository-level prompt construction and iterative retrieval-generation show how imports, related definitions, and feedback from generation can inform what is fetched (Shrivastava et al., 2023; Zhang et al., 2023). Repoformer learns when retrieval is worthwhile, while RLCoder learns a retriever without requiring labeled retrieved snippets (Wu et al., 2024; Wang et al., 2025b). Dataflow guidance, graph-based expansion, and repository-structure-aware retrieval provide complementary

routes to discovering dependencies (Cheng et al., 2024; Liu et al., 2024c; Phan et al., 2025; Ouyang et al., 2025). CoRet trains retrieval for editing rather than line completion (Fehr et al., 2025). These methods improve the candidate pool; our controlled tests ask what happens after it has been scored and before its evidence reaches the model.

Richer contexts and downstream workflows. CodeRAG targets relevant and necessary knowledge, RepoScope organizes call-chain-aware views, AIRCoder integrates multiple retrieval dimensions, and late code chunking changes the representation of retrieved code (Zhang et al., 2025; Liu et al., 2026; Shi et al., 2026; Oh & Lee, 2026). SpecAgent anticipates useful future context, whereas Think-Search-Patch organizes search within a repair workflow (Ma et al., 2026; Xiong et al., 2025). RepoClassBench, RepoExec, SWE-bench, and OpenHands assess progressively broader class, executable, issue-repair, and agent settings (Deshpande et al., 2024; Hai et al., 2025; Jimenez et al., 2024; Wang et al., 2025a). These settings motivate evaluation both at the evidence stage and in complete coding workflows.

Selecting evidence under a limit. RAG conditions generation on external material; MMR and submodular objectives balance relevance, redundancy, and coverage (Lewis et al., 2020; Carbonell & Goldstein, 1998; Lin & Bilmes, 2011). RAPTOR, RankRAG, and RepoShapley use hierarchical summaries, joint ranking and generation, and interaction-aware chunk values (Sarthi et al., 2024; Yu et al., 2024; Huo et al., 2026). Long-context position effects and retrieval-preference optimization further show that including evidence is not identical to using it well (Liu et al., 2024a; Yan et al., 2025). Unlike general-purpose diversity objectives, VITAL-RAG targets repository-specific post-retrieval loss by separating object-position redundancy from token-retention loss under a fixed renderer.

7 Limitations

The findings are specific to the stated RepoBench candidates, view construction, tokenizer, budgets, and path-and-identifier grouping; richer object definitions remain future work. Appendix B.3 shows that grouping after BM25 does not remove every scoring effect of duplicate views. The allocator is a greedy heuristic whose settings were fixed before held-out evaluation. Evidence retention is distinct from generation quality: the 100-task probe is inconclusive (Appendix C), and RepoExec evaluates the complete workflow, including output selection and documented recovery protocols (Appendices D and E).

8 Conclusion

Repository retrieval loses evidence when repeated views consume object positions and selected objects compete for tokens. VITAL-RAG reduces this retrieval-to-context gap through object grouping and bounded, relevance- and cost-aware allocation; consistent code-accuracy gains remain to be established.

REFERENCES

Jaime G. Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 335–336. ACM, 1998. doi: 10.1145/290941.291025. URL https://doi.org/10.1145/290941.291025.

Wei Cheng, Yuhan Wu, and Wei Hu. 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)*, pp. 7957–7977. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.431. URL https://aclanthology.org/2024.acl-long.431/.

Ajinkya Deshpande, Anmol Agarwal, Shashank Shet, Arun Iyer, Aditya Kanade, Ramakrishna Bairi, and Suresh Parthasarathy. Class-level code generation from natural language using iterative, tool-enhanced reasoning over repository, 2024. URL https://arxiv.org/abs/2405.01573.

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. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46701–46723. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets_and_Benchmarks.html.

Fabio James Fehr, Prabhu Teja S, Luca Franceschi, and Giovanni Zappella. CoRet: Improved retriever for code editing. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 775–789, 2025. doi: 10.18653/v1/2025.acl-short.62. URL https://aclanthology.org/2025.acl-short.62/.

Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1496–1524, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-naacl.82. URL https://aclanthology.org/2025.findings-naacl.82/.

Yu Huo, Kun Zeng, Siyu Zhang, Yuquan Lu, Cheng Yang, Yifu Guo, and Xiaoying Tang. RepoShapley: Shapley-enhanced context filtering for repository-level code completion. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 10390–10412. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.505. URL https://aclanthology.org/2026.findings-acl.505/.

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? In *The Twelfth International Conference on Learning Representations*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

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

Hui Lin and Jeff A. Bilmes. A class of submodular functions for document summarization. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 510–520. Association for Computational Linguistics, 2011. URL https://aclanthology.org/P11-1052/.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024a. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*. OpenReview.net, 2024b. URL https://openreview.net/forum?id=pPjZIOuQuF.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 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*, pp. 570–581. Association for Computing Machinery, 2024c. doi: 10.1145/3691620.3695054. URL https://doi.org/10.1145/3691620.3695054.

Yang Liu, Li Zhang, Fang Liu, Zhuohang Wang, Donglin Wei, Zhishuo Yang, Kechi Zhang, Jia Li, and Lin Shi. RepoScope: Leveraging call chain-aware multi-view context for repository-level code generation. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, 2026. URL https://conf.researchr.org/details/icse-2026/icse-2026-research-track/169/RepoScope-Leveraging-Call-Chain-Aware-Multi-View-Context-for-Repository-Level-Code-G.

George Ma, Anurag Koul, Qi Chen, Yawen Wu, Sachit Kuhar, Yu Yu, Aritra Sengupta, Varun Kumar, and Murali Krishna Ramanathan. SpecAgent: A speculative retrieval and forecasting agent for code completion. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, 2026. doi: 10.18653/v1/2026.acl-long.786. URL https://aclanthology.org/2026.acl-long.786/.

Seungmin Oh and Eunseok Lee. Late code chunking: A code chunking strategy for repository-level code completion. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 780–786. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-short.64. URL https://aclanthology.org/2026.acl-short.64/.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*. OpenReview.net, 2025. URL https://openreview.net/forum?id=dw9VUsSHGB.

Huy Nhat Phan, Hoang Nhat Phan, Tien N. Nguyen, and Nghi D. Q. Bui. RepoHyper: Search-expand-refine on semantic graphs for repository-level code completion. In *Proceedings of the IEEE/ACM International Conference on AI Foundation Models and Software Engineering*, pp. 14–25, 2025. URL https://conf.researchr.org/details/forge-2025/forge-2025-papers/14/RepoHyper-Search-Expand-Refine-on-Semantic-Graphs-for-Repository-Level-Code-Completi.

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

Chuanqi Shi, Miao Gao, and Zhiqiang Gao. 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)*, pp. 25458–25470. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.1166. URL https://aclanthology.org/2026.acl-long.1166/.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31693–31715. PMLR, 2023. URL https://proceedings.mlr.press/v202/shrivastava23a.html.

Aditya Bharat Soni, Boxuan Li, Xingyao Wang, Valerie Chen, and Graham Neubig. Coding agents with multimodal browsing are generalist problem solvers. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 6052–6069. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-eacl.318. URL https://aclanthology.org/2026.findings-eacl.318/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*. OpenReview.net, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. RLCoder: Reinforcement learning for repository-level code completion. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, 2025b. URL https://conf.researchr.org/details/icse-2025/icse-2025-research-track/24/RLCoder-Reinforcement-Learning-for-Repository-Level-Code-Completion.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 53270–53290. PMLR, 2024. URL https://proceedings.mlr.press/v235/wu24a.html.

Bojian Xiong, Yikun Lei, Xikai Liu, Shaowei Zhang, Pengyun Zhu, Yan Liu, Yongqi Leng, Ling Shi, Meizhi Zhong, Yurong Zhang, Yan Gao, Yiwu, Yao Hu, and Deyi Xiong. Thinksearch-patch: A retrieval-augmented reasoning framework for repository-level code repair. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1555–1566, 2025. doi: 10.18653/v1/2025.emnlp-industry.109. URL https://aclanthology.org/2025.emnlp-industry.109/.

Shi-Qi Yan, Quan Liu, and Zhen-Hua Ling. Rpo: Retrieval preference optimization for robust retrieval-augmented generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5228–5240. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.261. URL https://aclanthology.org/2025.acl-long.261/.

Yue Yu, Wei Ping, Zihan Liu, Boxin Wang, Jiaxuan You, Chao Zhang, Mohammad Shoeybi, and Bryan Catanzaro. RankRAG: Unifying context ranking with retrieval-augmented generation in llms. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/db93ccb6cf392f352570dd5af0a223d3-Abstract.html.

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

Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. 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*, pp. 23278–23288. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.1187. URL https://aclanthology.org/2025.emnlp-main.1187/.

ORGANIZATION OF THE APPENDICES

The appendices follow the argument in the main text. Appendix A supplies the populations, measurements, and method details for Sections 2 and 4. Appendix B extends the diagnosis in Section 3 and the policy comparison in Section 5.2. Appendix C gives the complete generation study discussed in Section 5.3.

Appendices D and E provide task-level evaluations and their specific protocols. Appendix F describes the supporting records. Each result appendix points back to the claim it supports so that supplementary analyses can be read alongside the relevant main-text comparison.

A EXPERIMENTAL AND METHOD DETAILS

This appendix supports the problem definition in Section 2 and the allocation policy in Section 4. It fixes the evaluation populations, evidence-matching rules, and implemented selection procedure before the additional result tables in Appendix B.

A.1 DATA, SPLITS, AND TOKEN ACCOUNTING

The RepoBench source contains 16,755 rows. Exact-row deduplication leaves 16,741 tasks; common eligibility screening yields 16,490 tasks from 2,798 repositories, including 8,556 Java and 7,934 Python tasks. The decomposition in Section 3.1 uses this complete eligible set. The candidate-count study in Section 3.2 uses the 14,442-task complement of 2,048 hash-selected exploratory tasks; these two subsets are disjoint and cover the complete set.

The policy comparison in Section 5.2 uses a separate split based on repository-ID hashes, with 13,121 held-out tasks after development settings were fixed. This split is not the 14,442-task complement. Its pilot-excluded and language-specific results are reported in Appendix B.4. All stated evidence budgets use `cl100k_base`; unless explicitly varied, the global budget is 4,096 and the object cap is 1,024 tokens. The strict-containment comparisons use 10,000 repository-cluster bootstrap resamples. Reported percentage-point contrasts are computed before the displayed rates are rounded.

A.2 EVIDENCE MATCHING DEFINITIONS

Let g_q be the original gold snippet, C_q the final context, and A_q the selected candidate set. Define norm to standardize line endings and remove only leading and trailing whitespace. Full-source containment is

$$\text{Containment} = \frac{100}{N} \sum_q \mathbf{1}[\mathrm{norm}(g_q) \text{ is a contiguous substring of } \mathrm{norm}(C_q)]\,. \tag{2}$$

We call this *full-source evidence containment*. A query-centered fragment alone does not satisfy it.

The frozen renderer exposes a different proxy. Let $U_q(g_q)$ denote snippet-derived rendering units for the gold candidate. Its *complete rendered-unit availability* is

$$\begin{aligned}\text{Availability} = \frac{100}{N} \sum_q \mathbf{1}\big[&g_q \in A_q \wedge \\ &\exists u \in U_q(g_q) : u \text{ is emitted without further truncation}\big].\end{aligned} \tag{3}$$

A unit can be a query-centered window of the original snippet. Completeness in this availability measure refers to that unit, not the original source. Selected recall measures only whether the gold candidate enters A_q. These definitions share the underlying gold candidate but not the evidence unit or matching rule. Their absolute values must not be pooled or directly subtracted. Both are evidence-level measurements, not code-accuracy metrics.

These are the formal definitions underlying the two measurements introduced in Section 2.2. Each result table identifies which one it reports; additional task analyses are in Appendix D.

A.3 Retrieved Views and Selection Controls

The evaluated expansion constructs three views per source snippet: its full text; its path, identifier, and first extracted signature line; and compact tokenized text with normalized case. Task-local BM25 uses $k_1 = 1.2$ and $b = 0.75$. Tokenization splits camelCase and snake_case; equal scores are ordered by original candidate position, path, identifier, and view type. An object inherits its strongest view score.

The evaluated object key is $(\mathrm{normpath}(\mathrm{path}), \mathrm{casefold}(\mathrm{identifier}))$. In the archived five-object run, the normalized source-provenance component was stored under the internal field name `fiber_id`; the twelve-object implementation computes the normalized path directly. View ranking can assign several positions to the same object. Object ranking instead selects the strongest representative of each distinct key in retrieval order. The five-object companion control takes four ranked objects and a non-core object from a represented source with lexical query overlap, falling back to the strongest remaining global object when needed.

These are the selection interventions compared in Section 3.1. Their separation from packing is essential: grouping changes which objects can receive positions, whereas the packing comparison changes context construction after selection. The exact-copy perturbation in Appendix B.3 additionally tests the effect of adding views before BM25 scoring.

A.4 Cost-Aware Selection Details

For the policy in Section 4.2, the first 50 distinct candidate IDs from the three-view ranking are merged into canonical objects. This initial pool is shared with rank-only selection and the five-object reference. The object limit is twelve for both the rank-only and cost-aware conditions.

Let r_i be an object's zero-based canonical rank, s_i its retrieval score, Q the set of unique query tokens, and T_i the unique tokens from its path, identifier, and first 1,200 snippet characters. Its relevance is

$$o_i = \frac{|Q \cap T_i|}{\max(1, |Q|)}, \tag{4}$$

$$a_i = \frac{1}{r_i + 2} + 0.15 \log(1 + \max(0, s_i)) + 0.30 o_i. \tag{5}$$

Let ℓ count `cl100k_base` tokens. The estimated cost is the token length of the path, identifier, and signature (or identifier fallback), plus a capped full-snippet cost:

$$\widehat{c}_i = \ell(\mathrm{path}_i + \mathrm{identifier}_i + \mathrm{signatureOrIdentifier}_i) + \min(700, \ell(\mathrm{fullSnippet}_i)). \tag{6}$$

These quantities enter Eq. 1. File novelty is 1.00 for an unrepresented file and 0.25 otherwise, corresponding to a 0.75 source penalty; the cost exponent is 0.5. Select the remaining object with the greatest current utility, append it to the ordered set, recompute novelty, and repeat until twelve objects have been chosen or the pool is exhausted. Neither gold labels nor generated outputs enter the decision.

For artifact correspondence only, `SOAG-5`, `Canonical-12`, and `Budgeted-12` identify the five-object reference, rank-only twelve-object control, and VITAL-RAG twelve-object policy, respectively. These are archive labels for the descriptive conditions used in Table 3.

A.5 Context Construction Details

Section 4.3 uses the same fixed context-construction procedure for all three policy conditions. It writes objects in selection order, first emitting the path, identifier, and signature, then a query-centered snippet window within the remaining object allowance. The global 4,096-token budget and per-object 1,024-token cap are actual limits; the selection cost is only an estimate. Rendering stops when the global budget is exhausted.

There is no cross-object reoptimization of token shares. The implementation's 128-token constant is used for a statistics field, not an enforced minimum allocation. This distinction explains why selection order can affect evidence retained by the shared procedure. The cap-scaling experiment in Appendix B.2 tests the separate effect of changing local and global allowances together.

B Additional Evidence and Sensitivity Analyses

These analyses extend the two main findings: the allocation loss diagnosed in Section 3, and the policy improvement in Section 5.2. Candidate count, token caps, and duplicate views test the scope of the diagnosis; the final subsection tests whether the policy benefit persists across restricted subsets.

B.1 Effect of Candidate Count

Section 3.2 uses the growing packing advantage to motivate cost-aware allocation. Table 5 provides the exact values behind Figure 2, keeping the same comparison and evaluation population.

K	Sequential	Completeness-first	Gain (pp)	95% CI (pp)
5	39.03	40.31	1.29	$[1.08, 1.50]$
10	45.61	56.40	10.79	$[10.11, 11.47]$
20	49.88	64.01	14.13	$[13.23, 14.98]$

Table 5: Numerical values for Figure 2. Rates are full-source containment percentages. All rows use the 14,442 tasks outside the exploratory subset, $B = 4096$, and cap 1,024. Paired contrasts precede rounding of displayed rates.

The 2,048 exploratory tasks and 14,442 remaining tasks are disjoint and together cover the 16,490-task population. At $K = 20$, rank-valued exact knapsack and shortest-first achieve 65.50% and 65.78% full-source containment. These controls support the allocation problem identified in Section 3.2 while showing that completeness-first is not the strongest packing rule in this setting.

B.2 Coupling Global and Per-Object Budgets

The fixed-cap and scaled-cap conditions use the same five-object portfolio and original-source containment test; only the global and per-object allowances differ. Paired contrasts are calculated before rounding and use the repository-cluster resampling described in Appendix A.1.

Global budget	Fixed cap=1K	Scaled cap	Containment	Gain (pp)
4K	40.44	1K	40.44	0.00
8K	40.45	2K	49.11	8.67
16K	40.45	4K	56.26	15.81

Table 6: Global/local budget coupling with five canonical objects. Rates are full-source containment percentages. The 16K gain has repository-cluster 95% CI $[14.96, 16.69]$ points.

B.3 Exact-Duplicate View Perturbations

Section 3.1 attributes the largest structural gain to grouping views into objects. Here we test a related mechanism by adding exact duplicate views before ranking. The experiment distinguishes protection against repeated object positions from invariance of the retrieval scores themselves.

Policy	Top-k set changed	Order changed	Mean Jaccard
Raw views	14.50%	30.96%	0.9385
Object maximum after BM25	3.76%	26.03%	0.9874
Exact deduplication before BM25	0.00%	0.00%	1.0000

Table 7: Exact-copy perturbation analysis on 2,048 hash-selected tasks. Changes are measured after adding duplicate views to the candidate pool.

This is an exact-copy perturbation, not a test of semantically paraphrased views. The object-maximum condition is applied after BM25 scoring, whereas exact deduplication changes the scorer's input before ranking.

B.4 Additional Policy Comparisons

Section 5.2 reports the primary twelve-object comparison. Table 8 places its availability and selected-recall intervals together and checks whether excluding all pilot tasks changes the conclusion.

VITAL-RAG minus rank-only selection	**Difference (pp)**	**95% CI (pp)**
Rendered-unit availability, held-out	3.58	$[2.92, 4.27]$
Selected recall, held-out	0.145	$[-0.424, 0.704]$
Rendered-unit availability, excluding all pilots	3.72	$[3.05, 4.44]$

Table 8: Paired contrasts for the shared-renderer policy comparison. The held-out comparison contains 13,121 tasks; the pilot-excluded row uses its restricted subset.

The held-out and pilot-excluded rows use different task sets; their intervals should not be pooled. The language-specific differences are approximately 3.76 points for Python and 3.42 for Java.

C Generation Study

Section 5.3 reports the main generation probe. All conditions generate one line with the same backend and decoding settings. The full context and output measures are:

Policy	**Availability (%)**	**Exact (%)**	**Strip-exact (%)**
Five-object reference	68	8	9
Rank-only ($K = 12$)	82	6	9
VITAL-RAG ($K = 12$)	87	10	11

Table 9: The 100-task generation study. Context availability and output matching are separate measurements.

On the 22 tasks for which only VITAL-RAG makes the gold rendering unit available, each policy solves one task. This restricted subset illustrates why newly available evidence cannot be equated with a correct one-line completion.

D Additional Task and Component Analyses

This appendix supplies further selection, class-generation, dependency-coverage, and budget results for the system evaluations in Section 5. Their protocols are specified in Appendix E. Evidence-availability columns use the rendered-unit definition in Appendix A.2, not the full-source measurement of Section 3.

D.1 Five-Object Selection Controls

The screened population in Table 2 contains 16,490 tasks, with 8,556 Java and 7,934 Python tasks from 2,798 repositories. Candidate-entry recall and selected-object counts for the five-object configuration are reported below. The source contains 16,755 rows; exact-row deduplication leaves 16,741 before common eligibility screening.

The Base and Full candidates in Table 2 were verified identical task by task, despite the distinct selection rules defined in Appendix E.1. Token reductions by language are 36.91% for Java and 34.18% for Python, calculated from unrounded means.

Scope	Tasks	R@5	Objects	Avail.@4K	Gap	Tokens
All	16,490	64.16	4.422	39.59	24.57	2,720
Java	8,556	63.97	4.413	37.60	26.37	2,798
Python	7,934	64.37	4.432	41.73	22.64	2,637

Table 10: Five-object selection: candidate entry and rendered-unit availability. The gap is R@5 minus Avail.@4K, not a full-source containment loss.

Control	Method	R@5	Objects	Avail.@4K	Tokens
Identity	View ranking	46.56	2.860	25.08	3,254
	Text deduplication	46.51	2.859	25.10	3,253
Coverage	Object ranking ($K = 5$)	64.10	4.422	39.47	2,734
	File coverage	63.98	4.424	39.42	2,714
	MMR	64.21	4.424	39.41	2,738
	4+1 selection	64.16	4.422	39.59	2,720

Table 11: Selection controls on 16,490 RepoBench tasks, using rendered-unit availability.

D.2 Class Generation

RepoClassBench supplies natural-language-to-class tasks (Deshpande et al., 2024). After unsuccessful generation calls were rerun under the documented recovery protocol, the runs contain 288 tasks; Token-F1 and character similarity are evaluated on 227 reference-bearing tasks. Comparators are locally implemented style controls, not official reproductions of the corresponding published systems (Zhang et al., 2025; Liu et al., 2024c; 2026).

Method	Gpt-5.4 Token-F1/Char	Claude Sonnet 4.6 Token-F1/Char	Qwen3-8B Token-F1/Char
CodeRAG-style	0.5764/0.4220	0.7084/0.5632	0.4865/**0.3479**
GraphCoder-style	0.5138/0.3434	0.6509/0.4673	0.4059/0.2799
RepoScope-style	0.5948/0.4491	**0.7203/0.5700**	**0.5164**/0.3457
VITAL-RAG	**0.6012/0.4561**	**0.7203/0.5700**	**0.5164**/0.3457

Table 12: Auxiliary RepoClassBench proxy results on 227 reference-bearing tasks, after unsuccessful generation calls were rerun under the documented recovery protocol. Claude and Qwen VITAL-RAG outputs reuse RepoScope-style outputs; Qwen includes adjusted recovery settings.

Claude and Qwen select RepoScope-style outputs on all 288 tasks; their matching scores are therefore expected, not independent evidence of an allocator benefit. GPT selects 269 outputs from the allocation configuration and 19 RepoScope-style outputs. Qwen's recovery procedure changed output limits, compactness instructions, and repetition penalties. Appendix E.4 specifies the recovery protocol.

D.3 Repository Execution

RepoExec evaluates generated repository functions with tests (Hai et al., 2025). Table 4 reports the complete workflow in the main text. Pass@1 uses the single selected output over 355 tasks per backend; two candidates were generated, so this is not a one-generation compute budget. The corresponding pass counts for VITAL-RAG are 205, 233, and 77 for GPT, Claude, and Qwen. Missing or unsuccessful generations count as failures. Paired control statistics are in Appendix E.6.

(a) Context-allocation components on RepoBench

Variant	R@5	Avail.@4K	Gap	Tokens
Full	64.16	63.67	0.49	1,751
No quotient	46.54	46.15	0.39	2,061
No rendering	64.16	39.59	24.57	2,720
No structure	46.56	25.08	21.48	3,254

(b) RepoExec dependency coverage: 231 tasks, 33 strict

Allocation	Dep. R	Strict R	All Tok.	Strict Tok.
Canonical 5+0	39.59	16.26	1,594	1,536
VITAL-RAG 4+1	38.94	21.31	1,612	1,485
Refined 3+2	34.16	22.20	1,778	1,725

(c) Context-budget sensitivity on RepoBench

Budget	Full	No rendering	Gain (pp)
1K	43.58	19.33	+24.26
2K	60.13	30.07	+30.06
4K	63.67	39.59	+24.08
8K	63.68	40.86	+22.82

Table 13: Context-allocation analyses under their stated metrics. RepoBench panels use rendered-unit availability, not full-source containment. Panel (b) measures recall of statically resolved dependency entities on the subsets defined in Appendix E.3; it is distinct from Pass@1 and RepoBench rendered-unit availability. Panel (c) reports Full-minus-No-rendering recall gains.

D.4 Component and Budget Analyses

Panel (a) separates candidate admission from rendered-unit availability: grouping increases candidate coverage, while rendering increases availability of the units that the renderer constructs. The full-source metric decomposition in Table 1 supplies complementary component evidence.

Panel (b) measures recall of statically resolved dependency entities, not execution correctness. The 33-task strict subset increases from 16.26% under 5+0 to 21.31% under 4+1, while overall dependency recall changes from 39.59% to 38.94%. The 3+2 configuration reaches 22.20% strict recall with greater token use. This is a local companion trade-off, not evidence for a universally optimal four-to-one partition.

Panel (c) keeps five slots and the 1,024-token local cap fixed. Its saturation cannot establish that a larger window is ineffective. The new cap-coupling analysis in Table 6 directly shows why expanding the local allowance alongside the global budget changes that conclusion.

E Task Evaluation Protocols

These details support the additional results in Appendix D and the distinction between context allocation and output selection in Section 5.3.

E.1 Selection Controls

The view construction and task-local BM25 scorer follow Appendix A.3. In the five-object comparisons of Appendix D.1, view ranking admits repeated object keys; text deduplication removes exact duplicate view hashes; object ranking keeps the best view for each provenance-and-identifier key. File coverage first assigns positions across files, then fills the remaining positions. MMR balances relevance and Jaccard redundancy with equal weights.

The ungated 4+1 control takes four ranked objects and the first same-source candidate; the query-linked version additionally requires lexical overlap with the query. The full configuration adds bounded context construction. Despite the distinct general selection rules, the Base and Full comparison in Table 2 has task-wise identical selected candidates, supporting its fixed-selection rendering attribution.

E.2 Construction of Comparison Methods

The class and execution tables in Appendices D.2 and D.3 use the following local comparison implementations.

The downstream comparison names denote local context-construction controls. GraphCoder-style arranges metadata, descriptions, imports, and headers as graph-like nodes. CodeRAG-style lexically ranks descriptions or code blocks. RepoScope-style combines local scope, detailed descriptions, and brief descriptions. These controls isolate practical context-organization choices within the evaluated implementation. They do not implement the full published systems, and the reported comparisons are not claims of superiority over official CodeRAG, GraphCoder, or RepoScope reproductions.

E.3 RepoExec Dependency-Coverage Subsets

This subset definition supports the companion-selection analysis in Appendix D.4, rather than the execution outcome in Section 5.3.

Of the 355 RepoExec execution tasks, 231 have at least one repository dependency identified by static resolution: `resolve_dependency_ids` returns a nonempty set. Let D_q be that set and S_q the selected entity set. The per-task metric used in the companion analysis is

$$\text{DependencyRecall}(q) = \frac{|S_q \cap D_q|}{|D_q|}, \qquad |D_q| > 0. \tag{7}$$

It measures selected-entity coverage of resolved dependencies, separately from execution Pass@1 and either of the RepoBench evidence metrics.

The strict subset contains 33 tasks satisfying all of the following recorded conditions: at least two resolved dependencies; available cross-context; different full-context and small-context prompts; dataset coverage of at least 90; and `isContained=false`. The companion analysis evaluates three allocation policies on the 231 eligible tasks, yielding 693 task-policy records. The strict subset is defined by these dependency and context conditions, not by whether the companion gate happened to switch a selection.

E.4 Class-Generation Recovery Protocol

Table 12 uses 288 tasks after unsuccessful generation calls were rerun under the documented recovery protocol, with proxy metrics computed on 227 tasks containing reference class bodies. Those completed calls are included in the displayed metrics, not counted as separate tasks.

In the completed run, Claude and Qwen select RepoScope-style on all 288 tasks, so their VITAL-RAG metrics are identical by construction. GPT selects 269 outputs from the allocation configuration and 19 RepoScope-style outputs. The Qwen completion process reruns unsuccessful calls with extended output limits, an additional instruction to keep the output compact, and changed repetition penalties. Consequently, this table is an auxiliary account of the completed selection workflow, rather than an independent, uniform-protocol test of the context allocator.

E.5 Program Selection

This procedure produces the selected outputs reported in Table 4. It is a separate post-generation decision, as distinguished from context allocation in Section 5.3.

Inputs and decision rules. The RepoExec selector receives two generated code strings and a task ID used for record alignment. Its decision function reads no reference implementation, test outcomes, or execution scores. It derives four static predicates: (i) removal of a specific class-as-receiver call pattern; (ii) successful AST parsing of the alternative when the default fails parsing; (iii) a nonempty set of suspicious names present only in the default, with the alternative passing AST parsing; and (iv) an alternative whose UTF-8 byte length is below half the default's, again guarded by successful parsing. The name heuristic collects AST uses, definitions, and imports, excludes builtins and a module allowlist, and filters logging names and names beginning with an uppercase letter from the set difference. The class-call rule is a pattern check rather than a complete interface verifier.

Policy and candidate configuration. The broad policy checks the class-call rule, then the suspicious-name rule, then compactness, retaining the default otherwise. The high-precision policy checks the class-call rule, syntax repair, and compactness in that order. The recorded GPT and Claude decisions follow the broad policy; Qwen's ledger explicitly records the high-precision policy. GPT uses the current RepoScope-style rerun as candidate A. Claude and Qwen use GraphCoder-style as A. Their allocation configurations supply candidate B. The default is retained on 307 GPT, 268 Claude, and 309 Qwen tasks. Thus the corresponding alternative counts are 48, 87, and 46. These implementation details describe the static selection stage used by the reported complete system.

E.6 Paired Statistics and Aggregation

The paired reports compare task-aligned execution outcomes. Bootstrap intervals use 10,000 paired resamples; sign tests are two-sided. A win means that the selected program passes and the comparator fails, and a loss means the reverse. Table 14 reproduces the stored statistics against the highest-scoring style control for each model in Table 4. Missing paired records are excluded from these comparisons, while Table 4 uses a fixed denominator of 355 per model.

Model	Comparator	Pairs	W/L/T	Gain (pp)	95% CI (pp)
GPT-5.4	RepoScope-style	355	20/12/323	2.25	$[-0.85, 5.35]$
Claude Sonnet 4.6	GraphCoder-style	353	47/8/298	11.05	$[7.08, 15.01]$
Qwen3-8B	GraphCoder-style	350	4/2/344	0.57	$[-0.57, 2.00]$

Table 14: Paired execution statistics. W/L/T denotes wins, losses, and ties. The corresponding sign-test values are 0.2153, $p < 0.0001$, and 0.6875. These are comparisons of the reported complete-system configurations.

For transfer accounting, the comparator is each selector's actual default. GPT's current RepoScope-style default passes 198 tasks, whereas the RepoScope-style row in Table 4 is a separate 197-pass run. Relative to the current default, GPT has 8 wins and 1 loss over 355 paired tasks. Claude and Qwen contribute 47/8 and 4/2 over 353 and 350 paired tasks. These stored paired reports recover 59 wins and 11 losses over their 1,058 common model–task records. The complete fixed-program policy table analyzed below contains 1,065 records and reports the same combined-policy W/L totals. Precision among decisive outcomes is $59/(59+11) = 84.3\%$, while pooled output success is $515/1065 = 48.36\%$. The 181 transfer count comes from the complete selection ledgers. The paired control comparisons use their stated intersections; the completed policy analysis uses the full model–task set.

E.7 Output-Transfer Policy Analysis on Fixed Programs

This analysis operates on 1,065 model–task records with two already generated programs A and B. The default A is retained or replaced by B according to the tested policy. It changes output selection after generation and does not rerun or ablate the upstream context allocator. Table 15 uses operational policy names to distinguish the interventions.

The matched-random row corresponds to the "without typed witnesses" control: for each of 10,000 draws it randomly chooses transfers while preserving their number for each model. Its unrounded mean wins and losses are 20.2791 and 12.3355, yielding $20.2791/(20.2791+12.3355) \approx 62.178\%$, reported as 62.2%. The displayed means therefore must not be interpreted as counts from one deterministic run. The "without direction" intervention adds a reverse alarm, while "without abstention" always chooses B. The additional Always-A, high-precision-only, and broad-gate rows show the corresponding retention and transfer trade-offs. The combined policy achieves the highest pass rate among these recorded policies; the high-precision-only policy has greater decisive precision but makes fewer transfers.

Output policy	Pass@1 (%)	Transfers	Wins/Losses	Precision
Combined policy	48.36	181	59/11	84.3%
Always A	43.85	0	0/0	–
Matched random	44.60	181	20.28/12.34	62.2%
High-precision only	46.76	104	34/3	91.9%
Broad gate	47.42	261	63/25	71.6%
With reverse alarm	46.10	362	69/45	60.5%
Always B	45.45	1,065	96/79	54.9%

Table 15: Output-transfer policies applied to fixed generated programs. Matched-random W/L values are means over 10,000 draws preserving each model's transfer count; its precision uses the unrounded means. This table analyzes output selection, independently of the context-allocation ablations.

F Data and Implementation Records

The main evidence tables and Figure 2 correspond to the reported experiment summaries and task-level comparisons. The accompanying source bundle contains their transcribed summary tables, figure inputs, and the correspondence between descriptive method names and archived run names. Appendix A provides the implemented method and metric definitions; Appendix E records task-specific evaluation protocols. Detailed file locations are listed in the bundle's supporting-evidence documentation rather than used as method names in the paper.

The reported artifact checks cover 896,604 context records, confirming unique keys and compliance with token budgets, as well as the disjoint and complete 2,048/14,442 task partition. The reported regression results are seven passing tests for the main evaluation and one for the policy evaluation. Full-file hash checking was not part of this validation pass. The fixed context-construction implementation has a matching SHA-256 source receipt. These experimental-artifact reports are distinct from the manuscript's numeric transcription and PDF checks; the source bundle does not include the complete underlying task-level artifact.

AI Use Statement

AI assistance was used for manuscript editing and presentation. The authors are responsible for the experimental evidence, implementation descriptions, and final claims.