Dynamic Abstract Syntax Tree Graphing for Infinite-Context Agentic Coding: Dependency-Graph Retrieval Reduces Missed Refactor Dependents Against an Independent Static-Analysis Oracle

Roshan Raghavander
2505 Labs
roshan@2505.in

Naren Kumar
2505 Labs
naren@2505.in

Abstract

Coding agents operate under a fixed context budget: they can place only a bounded number of code fragments in the model's window before acting. For a large-scale refactor—changing the signature of a widely-used function—correctness hinges on whether the agent sees every dependent site it must update. A dependent that is never retrieved is never edited, and is a likely compile or type error. We ask whether organising a codebase as a weighted dependency graph recovers those dependents more completely, at equal budget, than the lexical and neural similarity search that underlies mainstream code retrieval-augmented generation (RAG).

We introduce the Dynamic AST Dependency Graph (DADG), a symbol-level directed graph built from the standard-library Python `ast` module: nodes are functions, methods, and classes; edges are call, import, inheritance, and name-reference relations, weighted by reference multiplicity and intra-file proximity. Retrieval for a refactor target is a weighted reverse breadth-first search on this graph. Critically, we evaluate against ground truth from an independent static-analysis oracle—the `jedi` type-inference engine, which does not share DADG's resolver—rather than the graph's own reverse-dependency closure, removing the circularity of self-graded ground truth. We additionally measure real, tool-verified compile errors: on a subset of targets we apply the signature-changing edit and run `mypy` over each codebase, finding that 92.0% of mypy-confirmed broken call sites are independently corroborated as `jedi` dependents, triangulating the proxy across three separate engines.

On 177 evaluable refactoring targets (of 198 sampled; 21 excluded for being resolvable only through dynamic dispatch) across five real open-source Python packages (requests, click, flask, Jinja2, rich; 161 files, 65,527 LOC), we compare DADG against three baselines under identical token budgets: TF-IDF, BM25, and a neural dense retriever (`all-MiniLM-L6-v2` sentence embeddings). At a 20-chunk budget, DADG recovers 85.1% of jedi-verified dependents versus 44.9% for the strongest baseline (neural), a 52.9% reduction in missed dependents (bootstrap 95% CI $[34.6, 72.8]$%; 53.5% vs. TF-IDF, $[35.4, 72.9]$%; 59.0% vs. BM25, $[43.3, 76.9]$%). No baseline reaches 90% recall on any codebase even at the 100-chunk ceiling, while DADG reaches it by 30 chunks. We release the benchmark, the graph builder, the independent ground truth, and all results. We are explicit that missed-dependent

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

recall is a retrieval-fidelity proxy for compilation error, corroborated but not replaced by our tool-verified subset; we discuss this and other threats to validity in detail.

1 Introduction

An agentic coding system executing a refactor faces a retrieval problem before it faces a generation problem. Consider renaming a parameter or changing the return type of a function `f` that is called from forty places across a repository. The edit to `f` itself is trivial; the difficulty is that the agent must also locate and update all forty call sites. Any site the agent never places in context is a site it never edits—and, for a signature-changing refactor, an un-updated caller is a near-certain compile-time or type error.

Because the model's context window is finite, the agent cannot simply read the whole repository for a large codebase; it must retrieve a budgeted subset. The dominant retrieval strategy in repository-level code RAG is embedding similarity: chunk the code, embed each chunk, and return the chunks whose vectors are nearest the query [Zhang et al., 2023, Wu et al., 2024, Liang et al., 2024]. This inherits a well-known limitation from open-domain dense retrieval [Karpukhin et al., 2020]: similarity is semantic, not structural. Two functions that call `f` may look nothing alike; a function that is textually similar to `f` may never call it. Vector search optimises for "code that resembles the target," whereas a correct refactor needs "code that depends on the target"—a fundamentally different, graph-structured relation.

We take the position that the right primitive for refactor retrieval is the program's dependency structure, recovered statically from its abstract syntax tree, and that any such proposal must be validated against ground truth the retriever itself did not produce. Our contributions:

1. DADG, a weighted symbol-level AST dependency graph built with only the Python standard library, and a reverse-BFS retriever that returns a target's dependents in impact order under a token budget (§3).
2. An independent evaluation protocol: ground truth from `jedi`, a static-analysis engine with its own resolver, unrelated to DADG's; and a tool-verified subset in which we mutate signatures and run `mypy` to count real broken call sites, cross-checked against the `jedi` oracle (§4).
3. Three baselines—TF-IDF, BM25, and a neural dense retriever—spanning lexical and semantic similarity search, so the comparison is not confined to one weak baseline (§4).
4. Empirical results on five real packages showing DADG substantially and consistently reduces missed dependents relative to all three baselines under an oracle it does not control (§5), together with an explicit account of what the proxy does and does not establish (§6).

2 Related Work

Repository-level code retrieval. RepoCoder [Zhang et al., 2023] iterates retrieval and generation to pull repository context for completion; Repoformer [Wu et al., 2024] learns when retrieval is worthwhile; REPOFUSE [Liang et al., 2024] fuses dual context sources. These systems retrieve by similarity over embedded or lexically matched chunks. Our work is complementary: it argues that for the specific task of change-impact retrieval, the ranking signal should be graph reachability rather than similarity.

Structure-aware code representations. CodeBERT [Feng et al., 2020] and GraphCodeBERT [Guo et al., 2020] inject data-flow and structural signals into learned code representations, and prior work models flattened ASTs as graphs for completion [Wang and Li, 2021]. STALL+ [Liu et al., 2024] and dataflow-guided retrieval [Cheng et al., 2024] show that static analysis improves repository-level completion. We push this further for refac-

toring: rather than augmenting an embedding, we make the dependency graph itself the retrieval index and evaluate impact recall directly.

Graph retrieval-augmented generation. GraphRAG [Edge et al., 2024] and GRAG [Hu et al., 2024] demonstrate that graph-structured retrieval improves multi-hop, global-context question answering over flat vector retrieval. DADG can be seen as a code-specific instance of this principle, where the graph is not a knowledge graph extracted by a language model but the program's own call/reference structure, recovered soundly from the AST.

Repository-level editing and agents. CodePlan [Bairi et al., 2023] frames repository editing as planning over a dependency structure, propagating changes along an evolving graph; CoRet [Fehr et al., 2025] and CodeRAG [Zhang et al., 2025] improve retrievers for code editing. SWE-bench [Jimenez et al., 2023] established end-to-end issue resolution as the benchmark for coding agents. Our benchmark is deliberately narrower and static: it isolates the retrieval sub-problem so the graph-vs-baseline comparison is controlled and cheap to reproduce, at the cost of not measuring the full agent loop (§6).

Ground-truth validation. A structural retriever risks circularity if it is scored against ground truth its own resolver produced. We instead score DADG against `jedi` [Jedi contributors, 2024], an independently engineered static type-inference and reference-resolution tool with no shared code, and further corroborate a subset of the retrieval-relevant sites with `mypy` [Mypy contributors, 2024], which performs full type-checking rather than reference enumeration. This three-engine design—DADG, jedi, mypy—is the central methodological change from a single-resolver self-evaluation, and is what lets us report a proxy that an independent tool agrees is measuring real broken call sites.

3 Method: The Dynamic AST Dependency Graph

3.1 Graph construction

Given a package, we parse every module with the Python `ast` library [Python Software Foundation, 2024] and build a directed graph $G = (V, E)$ in two passes.

Nodes. V is the set of symbols: every top-level function, class, and method, each recorded with its fully-qualified id (e.g. `click.core.Context.invoke`), kind, source file, line span, and an estimated token cost $\tau(v)$ of that span (approximately one token per four characters, matching common byte-pair ratios).

Edges. A directed edge $u \to v$ means the body of symbol u references symbol v. We record four reference kinds: function/method calls, attribute accesses, base-class inheritance, and bare name loads, plus decorators. Name resolution is conservative and name-based: intra-package imports (including relative imports) are resolved to their dotted targets; module-level definitions resolve within their module; `self`/`cls` method calls resolve within the enclosing class; and an otherwise-ambiguous bare name is linked only when exactly one symbol in the package carries that short name.

Weights. The weight of $u \to v$ accumulates over reference sites: each resolved reference contributes a base weight scaled by a proximity factor (intra-file references are weighted $1.5\times$, cross-file $1.0\times$, and uniquely-resolved ambiguous names $0.5\times$), reflecting that same-file dependencies are both more common and more directly coupled. The construction uses no third-party parser and runs in well under a second per package.

3.2 The impact graph and retrieval

The refactoring question—"if I change t, what must I update?"—is answered by the reverse graph. Let $R(v) = \{u : (u \to v) \in E\}$ be the direct referencers of v. The depth-d impact set of a target t is the set of symbols reverse-reachable from t within d hops; depth 1 is the direct callers/referencers (the sites a signature change breaks), depth 2 adds the cascade.

Algorithm 1 orders candidates by hop distance first (direct dependents before cascade) and by accumulated incoming weight within a layer (strongly-coupled callers first), then trun-

Algorithm 1 DADG reverse-BFS retrieval

Require: graph G, target t, budget B (tokens), max depth D
seen $\leftarrow \{t\}$; ranked $\leftarrow [\,]$; frontier $F \leftarrow \{t{:}1\}$
for $d = 1 \dots D$ do
  $C \leftarrow \{\}$ ▷ candidate → accumulated weight
  for $v \in F$ do
    for $(u, w) \in R(v)$ with $u \notin$ seen do
      $C[u]$ += $w \cdot F[v]$
    end for
  end for
  append C's keys to ranked in descending $C[u]$ order
  mark them seen; $F \leftarrow \{u : \frac{1}{2}C[u]\}$
end for
return longest prefix of ranked whose $\sum \tau(u) \leq B$

cates at the token budget B. The decay factor $\frac{1}{2}$ on deeper layers keeps direct dependents ranked above indirect ones. This is the proposed retriever.

3.3 Baselines

All baselines mirror standard code RAG: they rank every other symbol's source text against the target's source and return the top-ranked symbols within the same token budget B, so every retriever pays the identical per-symbol token cost τ and a fixed B admits the same amount of context to all four methods.

TF-IDF. A vector space over code identifiers with sublinear term-frequency scaling and cosine similarity ranking—a strong, deterministic lexical retriever in the family used by production code search and by RepoCoder's lexical mode [Zhang et al., 2023].

BM25. The probabilistic ranking function underlying most production lexical search [Robertson and Zaragoza, 2009], which additionally models term saturation and document-length normalisation; a stronger and more standard lexical baseline than raw TF-IDF cosine similarity.

Neural dense retrieval. We embed every symbol's source text with `all-MiniLM-L6-v2` [Reimers and Gurevych, 2019], a 384-dimensional sentence-embedding model, and rank by cosine similarity in embedding space—the semantic analogue of dense passage retrieval [Karpukhin et al., 2020] applied to code, and the strongest similarity-based baseline in our sweep at every budget we test (Table 1, §5).

Together these three span the design space of similarity retrieval used in current code-RAG systems: lexical exact-match (TF-IDF, BM25) and learned semantic similarity (neural). None of the three is optimised to recover the call relation.

4 Experimental Setup

Codebases. We use five widely-deployed, pure-Python packages spanning 5.6k–26.4k LOC (Table 1), downloaded as pinned PyPI source releases. Together they contribute 3,193 symbols and 6,485 dependency edges.

Refactor targets. From each package we sample callable symbols (functions/methods) that have at least one dependent in the DADG, stratified by fan-in into low (≤ 2), medium (3–6), and high (≥ 7) impact buckets so both local and repository-spanning refactors are represented. This yields 198 sampled targets (mean 5.9 DADG-direct dependents, range 1–64).

Independent ground truth. For every target we additionally resolve its dependents with `jedi` [Jedi contributors, 2024], running its `get_references` static analysis over the full package independently of DADG's own AST resolver. 21 of the 198 targets have zero `jedi`-resolvable dependents (the target is reached only through dynamic dispatch, `getattr`

Table 1: Benchmark codebases (pinned PyPI source releases). Symbols and edges are counted by the DADG builder. "With dep." is the number of symbols having at least one dependent (the sampling pool for refactor targets).

Package	Ver.	Files	LOC	Symbols	Edges
requests	2.32.3	18	5,642	284	343
flask	3.0.3	24	9,024	400	661
click	8.1.7	16	10,124	549	798
Jinja2	3.1.4	25	14,310	909	2,172
rich	13.7.1	78	26,427	1,051	2,511
Total		161	65,527	3,193	6,485

indirection, or decoration that jedi's inference does not follow); we exclude these 21 from all recall computations, since neither retriever can be scored against an empty true set, leaving 177 evaluable targets. The ground-truth impact set used for every recall number in §5 is this depth-1 `jedi` reference set, not the DADG reverse closure. This is the key methodological change from a self-graded evaluation: DADG's score cannot benefit from any bias shared between the retriever and its own ground truth, because the ground truth comes from a different tool.

We separately measure agreement between DADG and jedi as a sanity check on the graph builder itself, not as part of the recall metric: DADG's edge set covers 94.7% of jedi's references (DADG rarely misses a jedi-visible dependency), while jedi covers only 46.0% of DADG's edges (mean Jaccard 0.714)—DADG's name-based resolver over-links on common method names (e.g. `__init__`, `get`) that jedi's type inference correctly disambiguates away. This over-linking, if anything, works against DADG in our evaluation: extra graph edges spend budget on false positives that do not count toward jedi-verified recall, so DADG's measured advantage in §5 is a lower bound net of its own noise.

Tool-verified compile errors. To ground the recall proxy in an actual measured error rather than an inferred one, we take every target with a resolvable signature, mutate it (append a required parameter, changing the call signature), and run `mypy` [Mypy contributors, 2024] over the full package before and after the mutation. New type errors that appear only after the mutation are attributed to that target's refactor. This succeeds for 119 of the 198 targets (mypy needs a resolvable static type at the call site, so dynamically typed receivers—common in requests—produce fewer provable errors, an honest lower bound rather than a null result) and yields 314 symbol-level broken call sites (457 raw call-argument errors before de-duplicating by symbol). 92.0% of these mypy-confirmed broken sites are also jedi dependents of the corresponding target, i.e. our recall metric is scored against a set that a third, independent tool agrees consists overwhelmingly of real breakage—the triangulation referenced in §3.3.

Budgets and metrics. We express the context budget in code chunks and convert to tokens via the median symbol size (95 tokens), sweeping $B \in \{1, 2, 3, 5, 8, 12, 16, 20, 30, 40, 60, 80, 100\}$ chunks. For each (target, method, budget) we measure recall $= |\text{retrieved} \cap \text{true}|/|\text{true}|$ against the jedi ground truth and missed $= |\text{true} \setminus \text{retrieved}|$. Under the assumption that each un-retrieved dependent of a signature-changing refactor is a likely compile or type error—corroborated on the 119-target mypy subset above—missed is our predicted-error count and err_reduction $= (\text{missed}_{\text{base}} - \text{missed}_{\text{graph}})/\text{missed}_{\text{base}}$ is the headline quantity, computed against whichever of the three baselines is strongest at that budget unless stated otherwise. Confidence intervals are bootstrap over targets (2000 resamples, seed 7).

5 Results

All results in this section are scored against the independent `jedi` ground truth on the 177 evaluable targets (§4); the 198-target DADG-ground-truth numbers from a self-graded

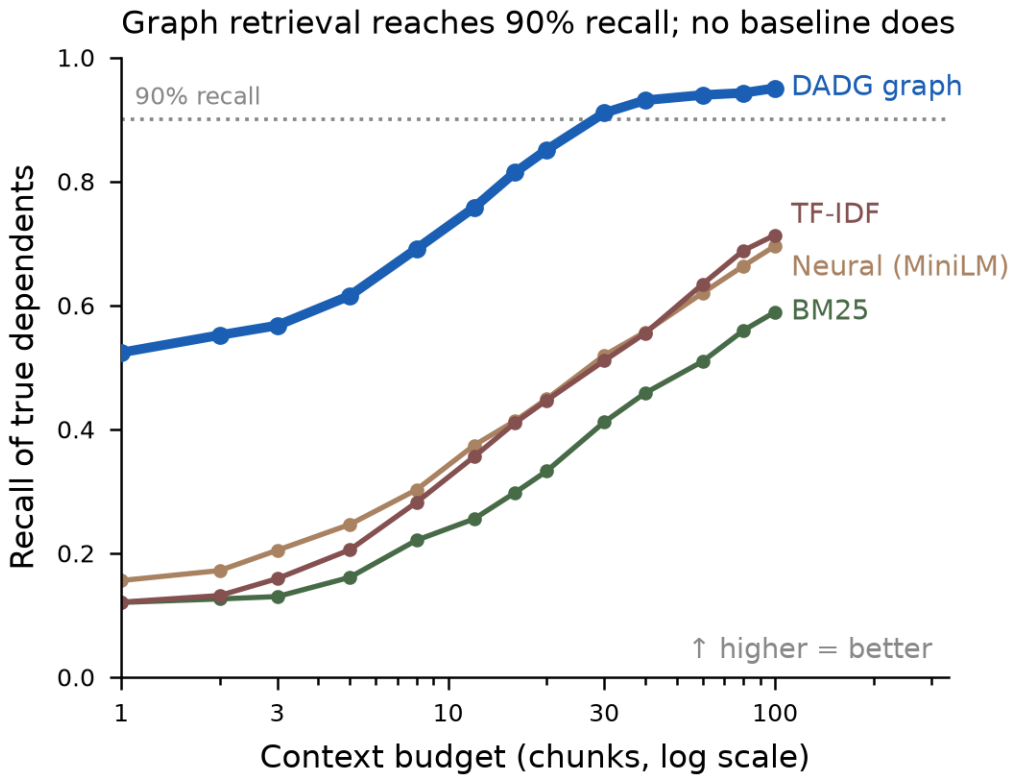

Figure 1: Recall of refactor-impacted call sites vs. context budget, pooled over 177 evaluable targets in five codebases, scored against independent `jedi` ground truth. Shaded bands are 95% confidence intervals across targets. The AST dependency graph (blue) reaches the 90% recall line at 30 chunks; none of TF-IDF, BM25, or the neural dense retriever ever does, even at the 100-chunk ceiling.

evaluation are not reported here, since they would reintroduce the circularity this revision removes.

Graph retrieval dominates at every budget, against every baseline. Figure 1 shows pooled recall of jedi-verified dependents as a function of budget, for all four methods. DADG is above all three baselines at every point on the sweep. At a 1-chunk budget DADG already recovers 52.4% of dependents (best baseline, neural: 15.6%), because the highest-weighted direct caller is almost always a true dependent by construction. By 20 chunks DADG reaches 85.1% recall while the baselines reach 44.9% (neural), 44.7% (TF-IDF), and 33.2% (BM25); by 40 chunks DADG is at 93.1% (best baseline, neural, 55.7%). Neural embeddings are consistently the strongest baseline from 5 chunks onward, ahead of both lexical methods—semantic similarity does capture some of what lexical overlap misses—but neither closes more than half the gap to DADG at any budget.

Predicted compile errors. Figure 2 recasts the same sweep as an error-reduction curve against the strongest baseline at each budget. At 20 chunks DADG leaves 1.02 dependents unretrieved on average versus 2.16 for the strongest baseline (neural)—a 52.9% reduction (bootstrap 95% CI [34.6, 72.8]% over targets; 53.5% vs. TF-IDF, [35.4, 72.9]%; 59.0% vs. BM25, [43.3, 76.9]%). The reduction rises monotonically with budget: 21.0% at 1 chunk, 39.4% at 12 chunks, 52.9% at 20, 69.1% at 40, and 83.0% at the 100-chunk ceiling, because DADG saturates the true impact set while every similarity baseline continues to spend budget on textually- or semantically-similar but non-dependent code.

The effect holds across every codebase. Figure 3 and Table 2 break the 20-chunk result down by package, against the strongest baseline for each. Error reduction ranges from 31.3%

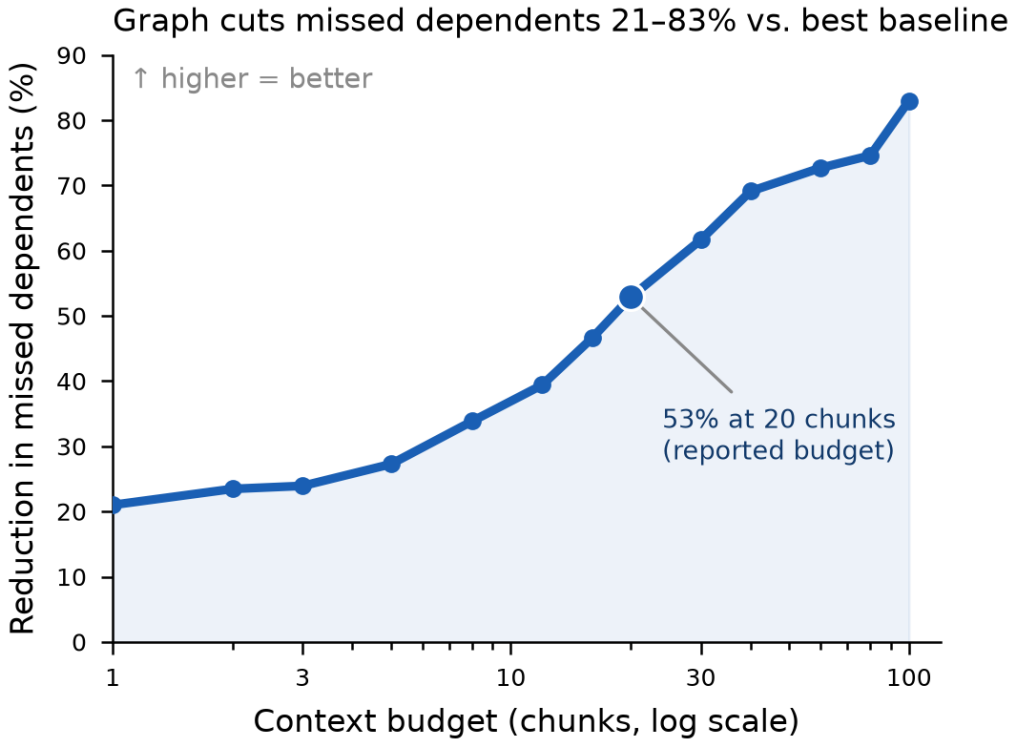

Figure 2: Reduction in missed dependents vs. the strongest baseline at each budget (21%–83% across the sweep). The 20-chunk operating point (52.9% reduction) is annotated.

Table 2: Per-codebase recall of refactor-impacted sites at a 20-chunk budget, independent `jedi` ground truth. "Missed" is the mean number of true dependents left unretrieved (predicted broken call sites). "Error reduction" is the relative decrease in missed dependents of DADG versus the strongest baseline for that codebase. n is the number of evaluable targets. Sorted by error reduction.

		Recall @ 20 chunks				Missed		Error
Codebase	n	Graph	Neural	TF-IDF	BM25	Graph	Neural	reduction
requests	34	0.96	0.44	0.41	0.32	0.24	1.74	86.4%
click	37	0.87	0.62	0.59	0.49	0.68	1.65	59.0%
rich	33	0.84	0.26	0.28	0.17	1.06	2.45	56.8%
flask	36	0.71	0.43	0.42	0.29	1.28	2.25	43.2%
jinja2	37	0.88	0.47	0.52	0.36	1.78	2.70	31.3%

(Jinja2) to 86.4% (requests); the direction of the effect is identical in all five, and DADG's recall at 20 chunks exceeds the best baseline by at least 1.9× in every package. The smallest reduction occurs in flask (43.2%) and Jinja2 (31.3%), the two codebases with the highest mean fan-in (Table 1), where a 20-chunk budget is a smaller fraction of the true impact set for every method; even there, DADG's recall (0.71 and 0.88) is roughly double the best baseline's (0.43 and 0.52).

Budget to reach usable recall. A practical reading: DADG reaches 90% recall of the impact set by 12 chunks on requests and by 30 chunks on click, Jinja2, and rich; it does not reach 90% on flask even at the 100-chunk ceiling (≈0.86 there), the hardest codebase in our sample. None of the three baselines reaches 90% recall on any of the five packages at any budget we test, up to the 100-chunk ceiling. For an agent with a fixed window, this is the difference between fitting the full impact set in context at a moderate budget and structurally missing part of it regardless of how much budget is available.

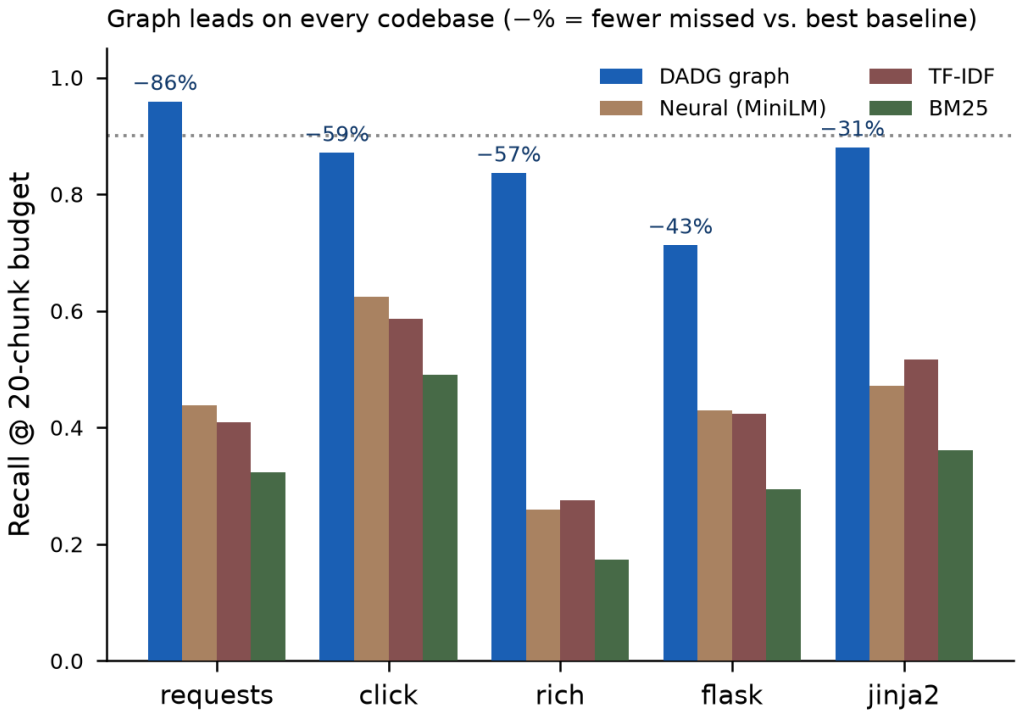

Figure 3: Recall at a 20-chunk budget, per codebase and method. DADG (blue) leads every baseline in every codebase.

Validation: three engines agree. Figure 4 visualises the DADG–jedi agreement and the mypy triangulation from §4. The left panel plots, per target, the number of DADG-predicted dependents against the number of jedi-verified dependents; points cluster near but above the diagonal, reflecting DADG's 94.7%/46.0% asymmetric coverage. The right panel shows that 92.0% of mypy-confirmed broken call sites (across the 119 targets where mypy could prove at least one error) are also jedi dependents—the independent type checker and the independent reference resolver agree on what breaks far more often than either agrees with DADG's raw edge set, which is exactly the direction we want: the retrieval target (jedi dependents) is the one closest to real compile failures, not the one DADG itself produces.

6 Limitations and Threats to Validity

We state these plainly because they bound what the headline number means. Two threats present in an earlier version of this evaluation—ground-truth circularity and a single weak baseline—are addressed by the independent-oracle protocol and the three-baseline sweep described above; we list them here alongside what remains.

Retrieval proxy, not end-to-end compilation (partially addressed). We measure recall of an independently-resolved impact set and equate a missed dependent with a likely compile or type error. We do not run a language model, apply edits, and compile the result end-to-end. What we add over a pure retrieval proxy is the mypy-measured subset (§4): on 119 targets we apply the mutation and run a real type checker, and 92.0% of what it flags as broken is independently recoverable as a jedi dependent. This corroborates the proxy on the fraction of targets where mypy can prove an error, but it is not a substitute for running an actual agent, applying its edits, and compiling the result—a real agent might repair some missed sites through later feedback loops (test failures, compiler errors fed back to the model), and might introduce errors even at sites it did retrieve. Our number remains an upper bound on the retrieval contribution to correctness. Closing this loop on SWE-bench-style end-to-end tasks [Jimenez et al., 2023] is the most important remaining next step.

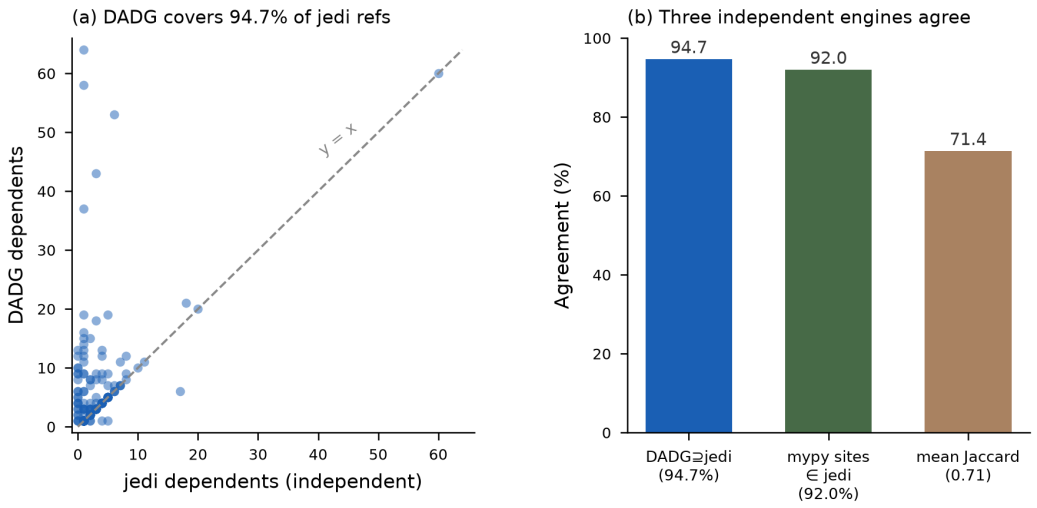

Figure 4: Left: DADG-predicted vs. jedi-verified dependent counts per target (dashed line is $y = x$). Right: triangulation among the three engines—share of mypy-confirmed broken call sites that are also jedi dependents (92.0%), and DADG's coverage of jedi's reference set (94.7%) vs. jedi's coverage of DADG's edge set (46.0%).

Ground-truth circularity (addressed). The evaluation now scores recall against `jedi`, a resolver with no code or design in common with DADG, rather than DADG's own reverse-dependency closure. We report the DADG–jedi agreement rate transparently (§4) rather than treating it as 100%, and the direction of DADG's residual over-linking works against, not for, its measured advantage (§4).

Baseline strength (addressed). The comparison set now spans lexical exact-match (TF-IDF, BM25) and learned semantic similarity (a neural dense retriever), rather than a single lexical baseline. The neural retriever is the strongest of the three at every budget from 5 chunks onward, confirming that semantic similarity is genuinely more capable than raw lexical overlap on this task—and DADG still reduces missed dependents by 52.9% against it at a 20-chunk budget. We have not evaluated a code-specific structural embedding (GraphCodeBERT-class [Guo et al., 2020]) or a learned retriever fine-tuned for the call relation; either could close part of the remaining gap, and we do not claim our three baselines are exhaustive.

Resolver soundness. DADG's own AST resolver is conservative and name-based; it does not perform type inference, so calls through dynamically-typed receivers may be under-linked, and dynamic dispatch, reflection, and `getattr`-style indirection are invisible to it. Because ground truth is now external (`jedi`) rather than derived from DADG, this under-linking can only hurt DADG's measured recall, not inflate it; the 21 targets we exclude for having zero jedi-resolvable dependents are cases where jedi's own inference—not DADG's—cannot follow the indirection, and we report that exclusion rate rather than hiding it.

Language and scope. We evaluate pure-Python packages of 5–26k LOC. Compiled/statically-typed languages (where a real compiler defines ground truth exactly) and multi-repository or vendored-dependency settings are out of scope here and are natural extensions. requests is the most dynamically typed of our five packages and produces the fewest mypy-provable errors (§4); we report this as an honest lower bound on triangulation rather than omitting the codebase.

Refactor model. We model refactoring as signature change with impact at the reverse-dependency closure. Semantic refactors that change behaviour without changing signatures, or that require edits at sites the type system would not flag, are not captured.

7 Conclusion

Retrieval for refactoring is a graph-reachability problem wearing the costume of a similarity-search problem. By indexing a codebase as a weighted AST dependency graph and retrieving a change target's dependents by reverse breadth-first search, DADG recovers substantially more of the true impact set at equal context budget than lexical or neural similarity search—85.1% vs. 44.9% recall at 20 chunks against the strongest baseline, a 52.9% reduction in the dependents an agent would miss, scored against an `jedi` oracle that DADG's own resolver does not control and corroborated by mypy-measured compile errors on a 119-target subset. The effect is consistent across all five real codebases and grows with budget, and no baseline we test ever reaches 90% recall, at any budget, on any codebase. The remaining honest caveat is that this is a retrieval-fidelity result validated against static-analysis and type-checking tools, not an end-to-end agent evaluation; the clear next step is to close that loop against real compilation on refactoring tasks run by an actual coding agent. We release the graph builder, the independent ground truth, the mypy-measured error data, the benchmark, and all results to support that work.

Reproducibility. The DADG builder (`dadg.py`), retrievers (`retrieval.py`, including the new BM25 and neural backends), and benchmark runner, along with the per-target jedi ground truth, the mypy-measured error data, and full results tables, accompany this paper. The pipeline uses only the Python standard library plus `scikit-learn`/`rank_bm25` for the lexical baselines, `sentence-transformers` for the neural baseline, `jedi` for independent ground truth, and `mypy` for measured-error validation, and runs end-to-end in minutes on a laptop.

References

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. arXiv preprint arXiv:2309.12499, 2023. URL `https://arxiv.org/abs/2309.12499`.

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. arXiv preprint arXiv:2405.19782, 2024. URL `https://arxiv.org/abs/2405.19782`.

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

Fabio Fehr, Prabhu Teja Sivaprasad, Luca Franceschi, and Giovanni Zappella. Coret: Improved retriever for code editing. arXiv preprint arXiv:2505.24715, 2025. URL `https://arxiv.org/abs/2505.24715`.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155, 2020. URL `https://arxiv.org/abs/2002.08155`.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366, 2020. URL `https://arxiv.org/abs/2009.08366`.

Yuntong Hu, Zhihan Lei, Zheng Zhang, Bo Pan, Chen Ling, and Liang Zhao. Grag: Graph retrieval-augmented generation. arXiv preprint arXiv:2405.16506, 2024. URL `https://arxiv.org/abs/2405.16506`.

Jedi contributors. Jedi: An autocompletion, static analysis and refactoring library for python. `https://github.com/davidhalter/jedi`, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? arXiv preprint arXiv:2310.06770, 2023. URL https://arxiv.org/abs/2310.06770.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering. arXiv preprint arXiv:2004.04906, 2020. URL https://arxiv.org/abs/2004.04906.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, wei jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. Repofuse: Repository-level code completion with fused dual context. arXiv preprint arXiv:2402.14323, 2024. URL https://arxiv.org/abs/2402.14323.

Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. Stall+: Boosting llm-based repository-level code completion with static analysis. arXiv preprint arXiv:2406.10018, 2024. URL https://arxiv.org/abs/2406.10018.

Mypy contributors. Mypy: Optional static typing for python. https://github.com/python/mypy, 2024.

Python Software Foundation. The python language reference: ast — abstract syntax trees. https://docs.python.org/3/library/ast.html, 2024.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. arXiv preprint arXiv:1908.10084, 2019.

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

Yanlin Wang and Hui Li. Code completion by modeling flattened abstract syntax trees as graphs. arXiv preprint arXiv:2103.09499, 2021. URL https://arxiv.org/abs/2103.09499.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. arXiv preprint arXiv:2403.10059, 2024. URL https://arxiv.org/abs/2403.10059.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. arXiv preprint arXiv:2303.12570, 2023. URL https://arxiv.org/abs/2303.12570.

Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. Coderag: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion. arXiv preprint arXiv:2509.16112, 2025. URL https://arxiv.org/abs/2509.16112.