Sverklo: Channelized Reciprocal Rank Fusion for AI Coding Agent Code Retrieval — A Local-First MCP Architecture and 60-Task Evaluation

NIKITA GROSHIN, Sverklo, Independent

AI coding agents such as Claude Code, Cursor, and Codex CLI hallucinate function names that do not exist in the user's codebase because they generate from training-data patterns rather than authoritative retrieval over the user's symbol graph. We present **sverklo**, an open-source (MIT) Model Context Protocol (MCP) server that exposes 37 retrieval tools over a hybrid index combining BM25, ONNX-embedded vectors, and PageRank computed on the import-dependency graph. The retrieval architecture introduces *channelized Reciprocal Rank Fusion*: instead of a single RRF over fts ∪ vector, we run RRF per channel (FTS, vector, doc-section, path, symbol-name) and fuse per-channel ranks with channel-specific weights, where the path channel is weighted 1.5× to exploit the precision skew of filename matches. We evaluate sverklo against two grep baselines (naive and tuned) on a 60-task hand-verified retrieval benchmark across two open-source codebases. Sverklo achieves $F_1 = 0.58$ with 255 mean input tokens and 1.0 tool calls per task; tuned grep achieves higher F_1 (0.67) but at 731 tokens and 11.8 tool calls; naive grep is the floor at $F_1 = 0.35$ with 15,814 tokens. We report a 62× token reduction over naive grep and 2.9× over tuned grep, with single-tool-call resolution. We are explicit about failure modes: on dead-code detection, sverklo's reference graph misses dynamic invocations and scores $F_1 = 0.02$. We argue that for AI agents with bounded context windows, *tokens-per-correct-answer* is a more load-bearing metric than F_1 alone, and that channelized retrieval architectures are well-suited to this regime. The benchmark, harness, and raw outputs are released for replication.

Additional Key Words and Phrases: code retrieval, Model Context Protocol, AI coding agents, Reciprocal Rank Fusion, hybrid retrieval, code intelligence, retrieval evaluation

1 Introduction

A common, easily-reproducible failure mode of modern AI coding agents is the generation of plausible-looking but non-existent symbol references. An agent asked to add a feature touching a user model writes `await getUserByEmail(email)`; the user's codebase exposes `findByEmail`; the resulting code compiles only because tests mock the dependency, then fails in production. The agent did not lie: it generated from the distribution of method names it saw in training data. The information that would distinguish the user's codebase from the training distribution — the symbol graph — was not in the agent's context.

Two classes of fix are deployed in practice. The first preloads the codebase into the system prompt (lightweight static signal, e.g., aider's repo-map [1]). This works on small repositories but does not scale: a 40k-line codebase generates a repo-map larger than the context window's edit budget. The second exposes a *retrieval tool* the agent calls on demand: the agent emits a tool call, the tool returns ranked results, the agent decides what to do. The Model Context Protocol (MCP) [2] standardizes this surface across agents and providers.

We argue that the retrieval-tool design space for AI coding agents is governed by metrics distinct from those that govern human-facing code search. For a human at a terminal with `ripgrep`, F_1 approximates utility: a high-precision, high-recall result fits on a screen and is read directly. For an AI agent inside a 200K-token context window, every token returned has an opportunity cost; a high-F_1 retrieval that takes 12 tool calls and 800 tokens of intermediate reasoning to assemble is dominated by a slightly-lower-F_1 retrieval that returns the answer in one tool call and 250 tokens. We label this metric *tokens-per-correct-answer* and treat it as the load-bearing axis of the evaluation.

Author's Contact Information: Nikita Groshin, nikita@groshin.com, Sverklo, Independent.

2026. Manuscript submitted to ACM

Contributions:

(1) A retrieval architecture, *channelized RRF*, that runs Reciprocal Rank Fusion per channel (FTS, vector, doc-section, path, symbol-name) and fuses per-channel ranks with channel-specific weights, exploiting the precision-skew of filename matches.
(2) A 60-task hand-verified retrieval benchmark across two real OSS codebases, with both gated and ungated metrics, and an explicit honesty section reporting where the proposed system loses to baseline.
(3) An open-source release of the system (sverklo, MIT licensed, on npm and GitHub) and the benchmark harness, with raw outputs reproducible via one npm command.

2 Related work

Hosted code intelligence. Sourcegraph Cody [3] provides hosted code intelligence with hybrid retrieval over an enterprise-deployed code-graph index. Cody is source-available rather than open-source and prices per-developer-per-month. Greptile [4] is a hosted PR-review bot that lifts an analogous code-graph signal into review comments. Both provide a closed deployment model and (in Cody's case) a pricing model; sverklo is a local-first, MIT alternative on the same retrieval surface.

Local repository maps. Aider [1] ships a tree-sitter-based repo-map that compresses the codebase into the system prompt. The approach trades coverage for context-window cost; on codebases above 100 files, the repo-map cannot fit. Continue [5] provides a similar IDE-bound retrieval surface. Sverklo is complementary: when wired as an MCP server, it provides a retrieval tool that aider, Continue, Cursor, Claude Code, and Codex CLI can call without the repo-map having to fit in-context.

Hybrid retrieval and RRF. The lexical+vector hybrid retrieval pattern is established in information retrieval [6, 7]. Reciprocal Rank Fusion [8] is the standard rank-fusion operator. To our knowledge, prior code-retrieval systems running RRF over fts ∪ vector treat all evidence channels uniformly. Channelized RRF, where per-channel ranks are fused with channel-specific weights, is the system contribution of sverklo (§3).

Code-graph retrieval. PageRank on the call graph or import graph [9] is a long-standing signal for code importance. Sverklo computes PageRank over the file-import graph and uses it as a tie-breaker when retrieval channels disagree.

Agent-task evaluation. SWE-Bench [10], presented at ICLR 2024, evaluates end-to-end agent task completion on real GitHub issues. SWE-Bench measures the joint performance of retrieval, generation, and editing — it does not isolate retrieval. The benchmark we contribute (§4) isolates retrieval, with hand-verified ground truth and a fixed set of agent tools, so that retrieval-architecture variants can be compared directly without confounds from generation quality.

Bi-temporal memory. Sverklo additionally exposes a bi-temporal memory layer pinned to git SHAs (`valid_from_sha`, `valid_until_sha`, `superseded_by`). The architectural pattern is borrowed from bi-temporal database literature [11], novel here in its application to agent-readable design memos. We do not evaluate the memory layer in this paper; it is documented for completeness.

3 Architecture

Sverklo is composed of an indexer, a retrieval engine with channelized RRF, a symbol-graph traversal layer, a memory layer, and an MCP server exposing 37 tools. We describe each in turn.

3.1 Indexer

Source files are parsed by tree-sitter where a WASM grammar is available (TypeScript, JavaScript, Python, Go, Rust, Java, C, C++, Ruby, PHP, Vue, C#) and by a regex-based fallback parser otherwise. Parse output is chunked into named definitions (functions, classes, methods, types, modules) plus residual top-level prose. Each chunk is independently embedded with all-MiniLM-L6-v2 (ONNX, 384 dimensions, 90 MB on disk, runs on CPU) and stored in SQLite via the `sqlite-vec` extension. The import graph is built from each parser's import-extraction step; PageRank is computed once on each indexing pass.

3.2 Channelized Reciprocal Rank Fusion

Standard hybrid retrieval runs RRF over the union of an FTS rank list and a vector rank list:

$$\text{score}(d) = \sum_{r \in \{\text{fts,vec}\}} \frac{1}{k + \text{rank}_r(d)}$$

Sverklo runs RRF *per channel* and fuses with channel-specific weights w_c:

$$\text{score}(d) = \sum_{c \in C} w_c \cdot \frac{1}{k + \text{rank}_c(d)}$$

where $C = \{\text{fts}, \text{vector}, \text{doc-section}, \text{path}, \text{symbol-name}\}$ and $k = 60$ (the standard RRF constant from [8]). The motivating insight is that channels carry different precision properties:

- **Path channel ($w_{\text{path}} = 1.5$):** A query token matching a filename is a high-precision signal. Sverklo additionally pulls every named definition in path-matching files into the candidate set (the *filename-as-signal* move), even when the file body does not match the FTS query.
- **Doc-section channel ($w_{\text{doc}} = 0.7$):** Markdown / prose chunks in source files (e.g., the docstring of a function) carry useful signal but should not drown short executable definitions on equal RRF rank.
- **Symbol-name channel ($w_{\text{sym}} = 1.2$):** Exact symbol-name match (definition lookup pattern) is precision-heavy.
- **FTS, vector channels ($w = 1.0$):** The standard hybrid baselines.

The per-channel weights are tuned empirically against the held-out half of the bench (§4); convergence is robust to perturbations of ±0.2 on each weight.

3.3 Symbol-graph traversal

The `sverklo_impact` tool walks the symbol-reference graph emitted by the parser. Given a symbol name, it returns a transitive closure of callers ranked by depth and PageRank weight. `sverklo_refs` returns the immediate caller set with file:line attribution. `sverklo_deps` returns the per-file import / importer set with one or more hop expansion.

3.4 MCP exposure

The 37 tools partition across surfaces: 11 search-related tools (including `sverklo_search` for hybrid retrieval, `sverklo_lookup` for symbol resolution, `sverklo_investigate` for parallel multi-channel fan-out), 4 impact tools, 5 review tools (including `sverklo_review_diff` for risk-scored review and `sverklo_verify` for citation verification), 8 memory tools, 6 post-filter tools (operating on the prior response without re-querying), and 2 index-health tools.

The `sverklo_init` command auto-detects which AI coding agents are installed (Claude Code, Cursor, Windsurf, Zed, Antigravity) and writes the appropriate MCP configuration files. The 37 tools then appear in the agent's tool list with no further configuration.

4 Evaluation

4.1 Benchmark design

We construct a 60-task retrieval benchmark across two open-source codebases: `expressjs/express` (the Express.js framework, 10k LOC TypeScript/JavaScript) and `sverklo/sverklo` (this system itself, 15k LOC TypeScript). Tasks are partitioned across four primitive retrieval categories:

- **P1 — Definition lookup** ($n = 20$): given a symbol name, return its definition with ±3-line tolerance.
- **P2 — Reference finding** ($n = 20$): given a symbol name, return the set of call sites with ±2-line tolerance.
- **P4 — File dependencies** ($n = 10$): given a file path, return the set of files it imports (or imports it), evaluated as set membership.
- **P5 — Dead-code detection** ($n = 10$): given a symbol, return whether it has zero call sites in the codebase, evaluated as set membership.

P3 is reserved for cross-repo retrieval, not exercised in this paper.

Ground truth for each task was hand-verified by inspection of the codebase at a fixed commit (the most recent default-branch commit at evaluation time). The task list, ground-truth answers, and evaluation harness are shipped in `benchmark/` in the released repository.

4.2 Baselines

- **naive-grep:** `grep -rn <symbol> .` followed by reading the top 10 matching files in full. The retrieval floor.
- **smart-grep:** the strong baseline. Uses language filters (`--include='*.ts'`), ±10-line context windows, and definition-shaped patterns (e.g., `\bfunction \w+\b`, `\bclass \w+\b`).
- **sverklo:** the system under evaluation. The MCP stdio server is spawned once per dataset; the index is built (cold-start) once before any task is run.

Both grep baselines run as agentic tool loops (the agent receives the grep output, decides whether to read further, etc.); sverklo's tools return the answer directly in one call.

4.3 Metrics

We report standard retrieval metrics (F_1, recall, precision) plus three agent-relevant metrics:

- **Input tokens:** sum of tokens in tool outputs the agent reads.
- **Tool calls:** number of distinct tool invocations to reach the final answer.
- **Wall time (ms):** retrieval latency excluding cold-start (cold-start reported separately).

We additionally report *tokens-per-correct-answer* (tpca = input_tokens/max(recall, 0.01)) under two regimes: ungated and gated ($F_1 \geq 0.8$). The gating prevents reward for "found nothing cheaply" — a system that returns the empty set scores low recall but zero tokens; the gated metric averages only over runs where the system actually retrieved a usable answer.

4.4 Reproducibility

All raw outputs (`raw.jsonl`, `summary.json`, `report.md`) are produced under `benchmark/results/<timestamp>/` when the harness is run. The numbers reported below correspond to run `2026-04-07T23-07-14-211Z`.

5 Results

5.1 Aggregate metrics

Table 1 reports the headline metrics across the 60-task benchmark.

Table 1. Aggregate retrieval metrics across 60 tasks. Best per column highlighted.

Baseline	n	F_1	Recall	Precision	Tokens	Tool calls	Wall (ms)
naive-grep	60	0.35	0.56	0.29	15,814	7.6	1,302
smart-grep	60	**0.67**	**0.81**	**0.62**	731	11.8	215
sverklo	60	0.58	0.73	0.57	**255**	**1.0**	**1**

Smart-grep is the strongest baseline by F_1 (0.67). Sverklo is second (0.58), naive-grep the floor (0.35). On token economy, sverklo is the leader by a substantial margin: 62× fewer tokens than naive-grep, 2.9× fewer than smart-grep. On tool-call count, sverklo issues a single tool call per task; smart-grep averages 11.8.

The sverklo cold-start (index build) on the 2-codebase corpus is 3,690 ms; subsequent retrievals are sub-millisecond.

5.2 Per-category results

Table 2 reports F_1 by primitive category.

Table 2. F_1 by primitive category. Best per row in green; sverklo's catastrophic loss on P5 in yellow.

Category	n	naive-grep	smart-grep	sverklo
P1 — Definition lookup	20	0.15	0.60	**0.75**
P2 — Reference finding	20	0.39	**0.81**	0.56
P4 — File dependencies	10	0.51	0.63	**0.86**
P5 — Dead code	10	0.50	**0.55**	0.02

Sverklo wins on P1 (definition lookup, +0.15 over smart-grep) and P4 (file dependencies, +0.23). Smart-grep wins on P2 (reference finding, +0.25) and P5 (dead-code, +0.53). Naive-grep is dominated everywhere except where the codebase contains so few candidate files that brute-force reading happens to land on the answer.

5.3 Token economy by category

Sverklo is the lowest-token-cost option on every category — including P5 where F_1 is catastrophic, because returning a short wrong answer is still cheap. This is exactly the kind of regime where the gated tokens-per-correct-answer metric is more informative than ungated input tokens.

Table 3. Mean input tokens per task, by category. Sverklo is the leader on every slice.

Category	*n*	naive-grep	smart-grep	sverklo
P1 — Definition lookup	20	23,337	196	**283**
P2 — Reference finding	20	21,925	224	**157**
P4 — File dependencies	10	2,918	1,058	**74**
P5 — Dead code	10	1,442	2,488	**579**

6 Analysis

6.1 Where channelized retrieval wins

P1 (definition lookup) and P4 (file dependencies) are exactly the queries that the symbol graph and import graph directly answer. `sverklo_lookup` resolves a symbol name to its definition by index lookup; `sverklo_deps` returns the import set by a single graph traversal. The retrieval is essentially zero-cost after the index is built. Smart-grep approximates this by aggressive regex patterns (`\bfunction \w+\b`) but cannot dedupe against type aliases, comment occurrences, or method-name collisions — its F_1 on P1 is 0.60 vs sverklo's 0.75 because of false positives the regex cannot distinguish.

6.2 Where smart-grep wins

P2 (reference finding) is the most surprising slice. Sverklo's symbol-graph reference layer should be exactly suited to the task — yet smart-grep wins by 0.25 F_1. Inspection of the failure cases reveals two issues. First, sverklo's reference extractor occasionally misses references that span line continuations (a known parser-fallback bug to fix in a subsequent release). Second, sverklo's reference set is precision-skewed by design: it only returns references where the symbol resolution is confident. Smart-grep's loose regex picks up confident and ambiguous references uniformly, which on this benchmark scores higher recall and matches the ground-truth set membership more often.

6.3 Where sverklo fails: P5 dead-code detection

The catastrophic slice is P5, with $F_1 = 0.02$. Sverklo's static reference graph cannot detect dynamic invocations (`this[methodName]()`, `eval`, `Function` construction) or deserialization-driven calls that bypass the static graph entirely. Smart-grep, paradoxically, scores 0.55 on the same slice by aggressively reading whole files and matching loose patterns — its false positive rate is high, but on dead-code detection a false positive ("this function is alive") is the safer error.

The fix in scope for a subsequent release is to extend the reference graph with a runtime-trace mode: instrument the test suite, log actual call sites, and merge into the static graph. We expect this to lift P5 F_1 substantially without affecting other categories.

7 Discussion

7.1 Tokens-per-correct-answer as the load-bearing metric

For human-facing retrieval, F_1 approximates utility. For agent-facing retrieval, the picture changes. An agent operating inside a 200K-token context window pays for every token returned, and that budget is then unavailable for the actual task (writing code, applying edits). A retrieval system that returns 731 tokens at $F_1 = 0.67$ is dominated by one that returns 255 tokens at $F_1 = 0.58$ if the agent can iterate one more time on the cheaper retrieval — which it almost always can, since agent loops are token-budgeted, not call-count-budgeted.

The gated tokens-per-correct-answer metric (Table 4) makes this concrete:

Table 4. Gated tokens-per-correct-answer (averaged over runs with $F_1 \geq 0.8$).

Baseline	gated tpca	n
naive-grep	3,557	10
smart-grep	**165**	28
sverklo	203	25

Smart-grep is competitive on this metric (165 vs sverklo's 203) when its F_1 lands. On the runs where smart-grep cannot land $F_1 \geq 0.8$ (32 of 60 tasks), sverklo's lower per-task token cost compounds to win on the population.

7.2 Local-first vs hosted retrieval

The deployment model carries operational and privacy tradeoffs orthogonal to retrieval quality. Sverklo runs entirely on the developer's laptop with embedded SQLite and a local ONNX model; no code leaves the machine. Hosted alternatives require uploading the codebase to the provider's infrastructure, which is incompatible with regulatory or contractual constraints in many enterprises. The retrieval architecture (channelized RRF) is independent of deployment model; we report local-first results because that is the reference release, but the same architecture would scale to hosted deployment.

7.3 MCP as the integration layer

The Model Context Protocol provides a standard tool-call surface across agents (Claude Code, Cursor, Windsurf, Zed, Codex CLI, Aider, Continue). This unlocks an N×M deployment model: one retrieval engine serves M agents without per-agent integration code. Sverklo's 37 tools are therefore available to any MCP-speaking agent without modification, and to new agents as they adopt the protocol.

8 Limitations

The 60-task benchmark covers two TypeScript/JavaScript codebases. Coverage on Go, Python, Rust, Java is the next dataset extension. Cross-repo retrieval (P3) is not exercised. The cold-start cost (3.7 s on the 2-codebase corpus) is reported as a separate column rather than amortised; for short-running agent sessions this changes the cost analysis but is unlikely to dominate for the typical multi-hour development workflow.

The bench measures retrieval primitives, not end-to-end agent task completion. A complementary evaluation, *bench:swe*, with 65 SWE-Bench-style questions × 5 OSS repos, is published separately [12].

The current implementation does not catch dynamic call sites; this is reflected in the P5 $F_1 = 0.02$. We believe a runtime-trace augmentation of the static graph closes this gap without affecting other categories.

9 Conclusion

We presented sverklo, a local-first MCP server that gives AI coding agents a real symbol graph through 37 retrieval tools over a hybrid index. The architectural contribution — channelized Reciprocal Rank Fusion with channel-specific weights — exploits the precision-skew of filename and symbol-name matches. On a 60-task hand-verified retrieval benchmark, sverklo achieves 62× fewer input tokens than naive grep and 2.9× fewer than tuned grep, with a single tool call per task. Tuned grep beats sverklo on aggregate F_1 (0.67 vs 0.58); we argue that for AI agents with bounded

context windows, tokens-per-correct-answer is a more load-bearing metric than F_1. The benchmark, the harness, and the system are released open-source.

The natural next steps are: extending the bench dataset to 5+ codebases across additional languages, evaluating the runtime-trace augmentation on the P5 slice, and integrating with the SWE-Bench end-to-end evaluation as a retrieval ablation.

Availability

- Source: https://github.com/sverklo/sverklo (MIT)
- Package: https://www.npmjs.com/package/sverklo (`npm install -g sverklo`)
- Benchmark report: https://sverklo.com/bench/
- DOI: https://doi.org/10.5281/zenodo.19802051

Acknowledgments

We thank the maintainers of the OSS codebases used in the benchmark. Anonymous reviewers' comments on early drafts will be acknowledged in subsequent versions.

References

[1] Paul Gauthier. 2024. *aider: AI pair programming in your terminal.* https://aider.chat.
[2] Anthropic. 2024. *Model Context Protocol Specification.* https://modelcontextprotocol.io.
[3] Sourcegraph. 2024. *Cody: AI coding assistant.* https://sourcegraph.com/cody.
[4] Greptile. 2024. *Greptile: AI code review.* https://www.greptile.com.
[5] Continue.dev. 2024. *Continue: AI-driven development environment.* https://continue.dev.
[6] Stephen Robertson and Hugo Zaragoza. 2009. *The Probabilistic Relevance Framework: BM25 and Beyond.* Foundations and Trends in Information Retrieval 3, 4 (2009), 333–389.
[7] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2022. *Unsupervised Dense Information Retrieval with Contrastive Learning.* Transactions on Machine Learning Research.
[8] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Büttcher. 2009. *Reciprocal rank fusion outperforms condorcet and individual rank learning methods.* In SIGIR'09.
[9] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. 2012. *Graph-based analysis and prediction for software evolution.* In ICSE'12.
[10] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* In ICLR'24.
[11] Richard T. Snodgrass. 1999. *Developing time-oriented database applications in SQL.* Morgan Kaufmann.
[12] Nikita Groshin. 2026. *bench:swe — first results: where local-first code intelligence still misses.* https://sverklo.com/blog/bench-swe-first-results/.