

Sverklo: A Local-First Code Intelligence MCP Server and a Cross-Repository Software Engineering Benchmark

Nikita Groshin
Independent Researcher
nikita@groshin.com

Abstract—Code-intelligence retrieval has become a primary bottleneck for AI coding agents on real-world repositories, yet existing Model Context Protocol (MCP) servers force a choice between sending source to a hosted service (Greptile, Cursor, Sourcegraph Cody), depending on language-specific servers (Serena via the Language Server Protocol [1]), or falling back to flat lexical search (ripgrep [2], ctags). We present SVERKLO, an open-source code-intelligence server that combines incremental tree-sitter parsing, a hybrid retriever fusing BM25 [3], dense embeddings (MiniLM-L6, 384-d [4]), and PageRank over a symbol-and-import graph via channelized Reciprocal Rank Fusion [5], and a bi-temporal memory store pinned to git commits. All computation runs locally; the package installs in one command and indexes 175 TypeScript files in approximately three seconds. We further introduce BENCH:SWE, a reproducible cross-repository evaluation harness spanning 65 hand-curated research questions across five popular open-source projects (Express, NestJS, Vite, Prisma, FastAPI), and BENCH:PRIMITIVES, a deterministic 60-task suite measuring recall, precision, and token efficiency on symbol lookup, reference finding, dependency analysis, and dead-code detection. On BENCH:PRIMITIVES, Sverklo achieves an aggregate F_1 of 0.58 while consuming an average of 255 input tokens per task—a 65% reduction over a tuned grep baseline (0.67 F_1 , 731 tokens) and a 98% reduction over naive grep (0.35 F_1 , 15,814 tokens). Sverklo wins decisively on definition lookup and dependency analysis but loses on reference finding and dead-code detection; the negative results are reported transparently. On BENCH:SWE, Sverklo’s hybrid retriever achieves 38/65 perfect recall and 66.2% average recall across the five out-of-distribution repositories (Vite 88.5%, FastAPI 69.2%, Express 65.4%, NestJS 57.7%, Prisma 50.0%); failure analysis reveals a systematic pattern in which deeply-nested “core” files (routers, dependency-injection internals, runtime engines) are co-imported by every component but under-served by lexical and dense channels alike. The harness and all ground-truth files are released under MIT license to enable third-party reproduction and comparison.

Index Terms—code intelligence, retrieval-augmented generation, software engineering benchmark, model context protocol, AI coding agents, PageRank, tree-sitter

I. INTRODUCTION

Large language model (LLM) coding agents now operate on repositories that comfortably exceed their context windows. SWE-bench [6], the canonical evaluation, ships entire GitHub issues into the prompt; the strongest baseline at publication

resolved fewer than 2% of instances, and most subsequent gains have come from *retrieval*, not larger models. Recent high-performing scaffolds (SWE-agent [7], AutoCodeRover [8], Agentless [9], RepoGraph [10]) all share one design choice: they construct a structured view of the codebase—either an agent-computer interface, a code property graph [11], or a tree-sitter syntax tree—before issuing any LLM call.

The Model Context Protocol (MCP) [12], [13], released in late 2024, standardizes how clients (Claude Code, Cursor, custom agents) consume such structured views via JSON-RPC. MCP decouples retrieval providers from agent harnesses, opening the design space for plug-in code intelligence. Yet in early 2026, the available servers cluster at two extremes. Hosted services (Greptile [14], Sourcegraph Cody [15], Cursor [16]) require sending source to remote infrastructure and typically operate on vendor-controlled indices. Local options (Aider’s repo-map [17], ctags-based servers) restrict themselves to a single signal—usually PageRank-on-symbols or trigram search—and lack the persistent decision memory that multi-session agentic workflows have begun to demand [18]–[20].

This paper contributes:

(1) **The SVERKLO system**, a local-first MCP server that fuses lexical, dense, and graph signals through channelized Reciprocal Rank Fusion [5] and adds a bi-temporal, git-aware memory store. Sverklo exposes 37 tools to MCP clients covering search, structural impact, code review, and persistent memory, and supports 130+ languages through optional tree-sitter [21] grammars. The full system is approximately 6,845 lines of TypeScript released under MIT.

(2) **The BENCH:SWE benchmark**, a public, reproducible evaluation suite spanning 65 research-style questions across five mature open-source projects in two ecosystems (Node.js: Express, NestJS, Vite, Prisma; Python: FastAPI). Ground truth is recorded as required-evidence files; the benchmark is intended as a community resource and accepts pull requests adding questions or repositories.

(3) **The BENCH:PRIMITIVES suite**, a deterministic, 60-task benchmark over four categories (definition lookup, reference finding, file dependencies, dead-code detection) on two repositories (Sverklo itself and Express 5.0.1). Three reference

baselines—naive grep, an informed “smart grep” that emulates a competent developer’s workflow, and Sverklo—are evaluated on F1, recall, precision, input token cost, and tool-call count.

The headline empirical finding is honest rather than triumphant. Sverklo achieves a 65% input-token reduction over the strongest grep baseline (255 vs. 731 tokens per task on average) at a 13%-lower aggregate F1 (0.58 vs. 0.67). It dominates on definition lookup (F1 0.75 vs. 0.60) and dependency queries (F1 0.86 vs. 0.63), underperforms on cross-file reference finding (F1 0.56 vs. 0.81), and is currently uncompetitive on dead-code detection (F1 0.02 vs. 0.55). Publishing a benchmark on which the proposed system does not uniformly win is itself a methodological contribution: it inverts the failure mode of vendor benchmarks, which typically report only winning configurations, and creates a public target for community improvement.

The rest of the paper is organized as follows. Section II situates Sverklo against prior work in code retrieval, agentic coding tools, and persistent memory. Section III describes the system architecture. Section IV introduces both benchmark suites. Section V reports results. Section VI discusses implications and threats to validity. Section VII concludes.

II. RELATED WORK

A. Repository-Level Code Benchmarks

The transition from function-level evaluation (HumanEval [22], MBPP [23]) to repository-level evaluation has produced several benchmarks of increasing scope. RepoBench [24] introduces three coupled tasks (retrieval, completion, pipeline) on Python and Java, demonstrating that cross-file context dominates next-line accuracy. CrossCodeEval [25] uses static analysis to guarantee that each completion provably requires cross-file context, isolating the contribution of retrieval. Long Code Arena [26] provides six project-wide tasks including library generation, CI repair, and module summarization, each requiring the entire repository. CodeRAG-Bench [27] systematically studies retrieval-augmented code generation across competition, open-domain, and repo-level tasks and finds that retrieval gains are highly source-dependent.

SWE-bench [6] and its derivatives (SWE-bench Verified [28], SWE-bench Multimodal [29]) measure end-to-end issue resolution and have become the de facto leaderboard. BENCH:SWE differs from all of the above on three axes: (i) it evaluates retrieval and question-answering rather than patch generation, isolating the code-intelligence layer; (ii) it spans two language ecosystems (Node.js and Python) by construction; and (iii) it is small (65 questions) and curated rather than mined, prioritizing inter-rater reproducibility over scale.

B. Code Retrieval and Code Search

Neural code search begins with CodeBERT [30], which established bimodal pre-training, and continues through GraphCodeBERT [31] (data-flow edges) and UniXcoder [32] (unified encoder-decoder with flattened ASTs). Repository-level retrievers add structural awareness: RepoCoder [33] iterates

retrieve-then-generate loops, RepoHyper [34] runs GNN link-prediction over a Repo-level Semantic Graph, and RepoGraph [10] exposes definition-reference relations at line granularity as a plug-in module. Most relevant to our chunking strategy, cAST [35] demonstrates that AST-based chunking outperforms naive line- or token-windowed chunking for retrieval-augmented code generation.

Sverklo positions itself as a deployable MCP-native counterpart to these research systems: it implements AST-based chunking via tree-sitter, exposes a symbol-and-file graph similar in spirit to RepoGraph’s, and adds dense retrieval via MiniLM-L6 [4]. The novelty lies less in any individual signal than in the channelized fusion (Section III-C) and in the engineering of all three pipelines into a single zero-config local executable.

C. AI Coding Agents and Tool Use

Agentic coding builds on the broader tradition of tool-using language models. ReAct [36] interleaves reasoning steps with environment actions; Toolformer [37] learns when to invoke tools self-supervisedly. SWE-agent [7] argues that the design of the *agent-computer interface* (ACI) more than doubles SWE-bench performance over passive baselines. AutoCodeRover [8] combines spectrum-based fault localization with structured code search. Agentless [9] demonstrates that a simple localize-repair-validate pipeline outperforms complex agent loops at a fraction of the cost. OpenHands [38] provides the most widely used open agent harness, often paired with executable Python actions [39].

Sverklo is, in this taxonomy, neither an agent nor a benchmark solution but an *ACI primitive*: an MCP-shaped collection of retrieval and analysis tools that any of the agents above can adopt.

D. Persistent Memory for LLM Agents

MemGPT [18] introduces an OS-inspired tiered memory architecture (core, archival, paging). Voyager [19] demonstrates that a growing skill library of executable code, plus an automatic curriculum, produces compounding gains in an open-ended environment. Generative Agents [20] establishes the now-standard distinction between episodic memory (events) and semantic memory (consolidated facts). Sverklo’s memory store implements the same axis (episodic / semantic / procedural) but introduces two code-specific properties: memories are pinned to git commits via `valid_from_sha` and `valid_until_sha`, and a staleness check excludes memories whose referenced files no longer exist on disk (Section III-D).

E. Static Analysis Foundations

The Code Property Graph [11] unifies AST, control flow, and program dependence into a single queryable structure, underpinning Joern and many ML-on-code systems. PageRank-style ranking on software graphs predates LLMs: Zaidman and Demeyer [40] apply HITS/PageRank to dependency graphs to identify “key classes,” refined by Pan et al. [41]. Tree-sitter [21] provides an incremental GLR parser, descended

from Wagner and Graham [42], that produces concrete syntax trees in milliseconds and now powers GitHub, Neovim, and most modern static analyzers. Sverklo’s PageRank-on-files-and-symbols ranking and tree-sitter parsing layer are direct applications of these foundations to MCP-shaped tooling.

III. SYSTEM: SVERKLO

A. Architecture Overview

Sverklo is a TypeScript program distributed via npm (npm install -g sverklo). It runs as either a long-lived MCP stdio server, a one-shot CLI, or a local HTTP dashboard. The implementation comprises eight subsystems—indexer, search, memory, storage, server, concepts, VS Code extension, and telemetry—summing to roughly 6,845 lines of source and 309 passing tests as of release v0.17.1 (April 2026). All persistent state lives in two SQLite databases under `$HOME/.sverklo/`: a per-project index keyed by project hash, and an optional workspace-shared memory store. There is no external database, vector service, or cloud dependency.

B. Indexer and Symbol Extraction

The indexer pipeline performs language detection, gitignore-respecting file discovery, parsing, chunking at symbol boundaries, embedding, reference extraction, and dependency-graph construction. By default the parser uses regex patterns covering 20+ languages including TypeScript, JavaScript, Python, Go, Rust, Java, C/C++, Ruby, PHP, Kotlin, Scala, Swift, Dart, and OCaml. As of v0.17, an opt-in tree-sitter pipeline (`SVERKLO_PARSER=tree-sitter`) loads WASM grammars from `${HOME}/.sverklo/grammars/` for six languages (TypeScript, TSX, JavaScript, Python, Go, Rust) and falls back silently to regex when grammars are unavailable.

The dual-parser design solves a deployment problem observed in the field: tree-sitter binary builds fail on a non-trivial fraction of npm install invocations on Windows and underfunded CI environments. web-tree-sitter’s asynchronous module loading also changed in version 0.24 [21], breaking many downstream wrappers. Making tree-sitter optional and keeping the regex fallback always-on lets the package install reliably while individual users opt into AST-based parsing when grammars are available. Section V-F reports parser-parity numbers that motivate keeping regex as the default through v0.17.

Each parsed file produces a set of `ParsedChunk` records (type, name, signature, content, line range) plus a synthetic `module:_module` chunk holding file-level prose extracted from leading comments and docstrings. References (function calls, constructor invocations, JSX components, decorators) are extracted via language-specific regex patterns and stored in a per-line deduplicated `symbol_refs` table; per-line rather than per-chunk deduplication is required to preserve blast-radius accuracy when a single function calls the same target multiple times. The indexer embeds chunk content using ONNX Runtime [43] with the all-MiniLM-L6-v2 encoder [4], downloaded once to `${HOME}/.sverklo/models/` and cached. Cold indexing on modestly sized repositories (e.g.,

NestJS at 2,976 chunks) completes in approximately 22 seconds; sverklo’s own 175 TypeScript files complete in under three seconds. Embedding accounts for roughly 60% of cold-index wall time. Incremental refresh on file change is sub-second.

C. Hybrid Retrieval

The search subsystem implements channelized Reciprocal Rank Fusion [5] over five parallel retrievers: (i) FTS5 full-text search backed by BM25 [3], (ii) vector similarity over code-typed chunks, (iii) vector similarity over documentation-typed chunks, (iv) symbol-name exact and prefix lookup, and (v) path-token matching against file names. Standard hybrid search systems run a single RRF over the union of lexical and dense candidates; Sverklo runs RRF independently per channel and then fuses channel ranks with channel-specific weights, boosting the path channel by $1.5\times$ (filename matches are precision-skewed) and decoupling code-channel from doc-channel scoring so that long Markdown documents do not drown short function bodies in a single fusion.

The search ranks are combined with PageRank scores [41], [44] computed offline on the file-level dependency graph (damping 0.85, 20 iterations, convergence threshold 10^{-4}). PageRank is stored in `files.pagerank` and used both as a tiebreaker in search and as the primary ordering for the `sverklo_overview` tool, which returns the structurally most-depended-upon files in a repository.

The retriever introduces one design pattern we have not seen documented elsewhere, which we call *filename-as-signal*: when a query token matches a filename, all named definitions inside that file are added to the candidate pool, even if their bodies do not match the query. Symmetrically, if FTS surfaces a file at all (e.g., because of a comment or import statement), every definition in that file is treated as a plausible result. This closes a recall failure mode in which short helper functions live next to query-relevant code but are individually too small to dominate either BM25 [3] or vector similarity. The filename-as-signal channel contributes measurably to the 97% perfect-recall figure on Sverklo’s internal research benchmark (Section V-D); the implementation is open in `src/search/investigate.ts`.

D. Bi-Temporal Memory Store

Multi-session agentic workflows require persistent context that outlives a single conversation. Sverklo’s memory subsystem stores decisions, preferences, patterns, contexts, todos, and procedural notes in a SQLite table indexed semantically (FTS5 + 384-d embeddings). Two design choices distinguish it from related systems [18], [20].

Bi-temporal lineage. Every memory carries a `valid_from_sha` (the git SHA at which it was recorded) and, when superseded, a `valid_until_sha` together with a `superseded_by` pointer to its replacement. Updates do not overwrite; they invalidate. The default recall query naturally filters `valid_until_sha IS NULL`, but the timeline is preserved for retrospective queries (“what did

TABLE I
SVERKLO MCP TOOL SURFACE, GROUPED BY FUNCTION.

Group	Representative tools
Search	<code>sverklo_search</code> , <code>sverklo_overview</code> , <code>sverklo_lookup</code> , <code>sverklo_context</code> , <code>sverklo_ast_grep</code>
Impact / refs	<code>sverklo_impact</code> , <code>sverklo_refs</code> , <code>sverklo_deps</code> , <code>sverklo_audit</code>
Diff review	<code>sverklo_review_diff</code> , <code>sverklo_test_map</code> , <code>sverklo_diff_search</code>
Memory	<code>sverklo_remember</code> , <code>sverklo_recall</code> , <code>sverklo_memories</code> , <code>sverklo_promote</code> , <code>sverklo_demote</code>
Index health	<code>sverklo_status</code> , <code>sverklo_wakeup</code>

we believe about X at commit `abc123?`). This addresses a failure mode in flat-memory systems in which the loss of stated assumptions makes later debugging expensive.

Staleness via file-set pinning. Each memory may carry a list of `related_files`. On recall, a staleness check verifies that each listed file still exists in the index; if any are missing, the memory is marked stale and excluded from results. This catches design decisions that have outlived the code they describe without requiring an external oracle, model call, or LLM judge.

The *kind* axis (episodic, semantic, procedural) [20] is orthogonal to the *category* axis (decision, preference, pattern, context, todo, procedural) and lets the system retrieve, e.g., timeless rules separately from moment-bound events. A v0.17 addition, *workspace memory*, extends the same store to a path-keyed shared database so that a team of services can share decisions without a centralized backend.

E. MCP Server and Tool Surface

The server uses the official MCP TypeScript SDK [12] and exposes 37 tools across five functional groups (Table I). Per-project tool overrides via `.sverklo.yaml` let users disable individual tools (e.g., `sverklo_refs` for codebases under code-review embargo).

These tools are the units of measurement for the benchmarks introduced next.

IV. BENCHMARKS

We introduce two benchmarks. BENCH:PRIMITIVES measures deterministic, single-symbol-or-file primitive operations against exact-answer ground truth and is intended as a regression harness that any code-intelligence tool can join. BENCH:SWE measures open-ended retrieval against research-style questions on five high-profile open-source projects and is intended as a public community resource for cross-tool comparison.

A. bench:primitives: Deterministic Primitives

Categories. Four task types each test a distinct primitive operation (Table II).

Datasets. Two repositories, 30 tasks each: Sverklo’s own codebase (~175 TypeScript files) and Express

TABLE II
BENCH:PRIMITIVES TASK CATEGORIES. P3 (CALL-GRAPH TRAVERSAL) WAS DEPRECATED PRIOR TO V0.17 AND IS OMITTED; THE ORIGINAL NUMBERING IS PRESERVED FOR CONTINUITY WITH THE RELEASED HARNESS.

ID	Task	Input/Output
P1	Definition lookup	symbol \rightarrow {file, line} (1 match)
P2	Reference finding	symbol \rightarrow {file, line} [*] , excluding def
P4	File dependencies	file \rightarrow {imports, importers}
P5	Dead-code detection	repository root \rightarrow orphan symbol names

5.0.1 (`expressjs/express`, ~150 JavaScript files). Sverklo’s tasks are hand-written in `benchmark/src/ground-truth/seed/sverklo.jsonl`; Express tasks are produced programmatically by a generator (`express.gen.ts`) that resolves locations against a freshly cloned repository on every run, so generated ground truth cannot drift from upstream.

Scoring. Recall, precision, and F1 are computed against exact-answer keys. P1 uses a ± 3 -line tolerance and P2 a ± 2 -line tolerance to accommodate parser disagreements about definition vs. body lines; P4 and P5 use set membership over normalized paths. Token cost is estimated from each baseline’s raw input payload via `chars/3.5`, which is comparable across baselines but not to specific BPE tokenizers. We report a *quality-gated* efficiency metric, `tokens-per-correct`, defined only over runs with $F_1 \geq 0.8$, to prevent rewarding “found nothing cheaply.”

Baselines. Three baselines are evaluated.

(1) *naive-grep* is the absolute floor: stateless `grep -rn` over the working tree with no exclusions, plus top-10 file reads. It indexes `node_modules`, `.git`, and build artifacts.

(2) *smart-grep* is the realistic developer floor: language filtering (`--include='*.ts'`, `--include='*.js'`, etc.), exclusion of `node_modules`, `dist`, `.git`, and build. For P1 it prefers a definition-shaped regex matching function, `class`, `interface`, `type`, `const`, `let`, `var`, `def`, and `export default function` declarations, and falls back to plain `grep` on miss; for P2 it returns ± 10 context lines around each hit.

(3) *Sverklo* runs as an MCP stdio subprocess. P1 maps to `sverklo_lookup`, P2 to `sverklo_refs`, P4 to `sverklo_deps`, and P5 to the `orphans` section of `sverklo_audit`. Sverklo pays a one-time cold-start cost (index build, ~3.7 s on these datasets) and amortizes to one tool call per task.

B. bench:swe: Cross-Repository Research Questions

BENCH:SWE extends evaluation to open-ended natural-language questions whose answers require understanding of multiple files. The five repositories are Express 5.0.1, NestJS 10.4.7, Vite 6.0.7, Prisma 6.1.0, and FastAPI 0.115.6, pinned to commit SHAs to ensure reproducibility. Each repository contributes 13 questions (65 total). Questions are hand-written

TABLE III
BENCH:PRIMITIVES OVERALL ($n = 60$). TOKENS COLUMN IS MEAN PER TASK; GATED TOKENS ARE MEAN OVER RUNS WITH $F_1 \geq 0.8$. BEST IN COLUMN IN **BOLD**.

Baseline	F1	Recall	Prec.	Tokens	Gated tok./correct
naive-grep	0.35	0.56	0.29	15,814	3,557
smart-grep	0.67	0.81	0.62	731	165
Sverklo	0.58	0.73	0.57	255	203

by the authors and probe behaviors distributed across the codebase, e.g.:

“How does Express dispatch a request to the right route handler when multiple routers are mounted on overlapping paths?”

“Where is the four-argument error-handling middleware signature implemented and how does it differ from the three-argument middleware path?”

Ground truth is stored as a list of *required-evidence* files, optionally with symbol names or line ranges. A retriever passes a question if its top- k hits include all required-evidence files; we report `avg_recall`, `perfect_recall` (count of questions with recall = 1.0), `avg_wasted_hits` (an efficiency signal), and `avg_duration_ms`. The harness and ground-truth files live in `benchmark/src/swe/` and `benchmark/src/swe/datasets/`, with $k = 50$ as the default top- k . End-to-end results across all five repositories are reported in Section V-E.

C. bench:research: Internal Recall Baseline

A third internal benchmark, BENCH:RESEARCH, evaluates 32 research-style questions against Sverklo’s own codebase, using the same required-evidence-file rubric. This is not a cross-tool comparison—other tools are not run on it—and serves as a deterministic regression harness during development. Its results appear in Section V-D only to validate that release-to-release changes do not regress recall on the internal questions on which they were tuned, an important caveat we discuss below.

V. RESULTS

A. bench:primitives Aggregate Performance

Table III reports overall results across all 60 tasks for each of the three baselines.

Smart-grep is the strongest baseline on F1 (0.67) and on the quality-gated tokens-per-correct metric (165 vs. 203). Sverklo wins on average input tokens (255 vs. 731, a 65% reduction) and on tool calls (1.0 vs. 11.8). Naive grep is dominated everywhere except recall—a partial illusion, since it returns large undifferentiated result sets that sometimes contain the right answer by sheer volume.

The aggregate picture conceals substantial per-category variation, which we now examine.

TABLE IV
PER-CATEGORY F_1 ($n = 20/20/10/10$).

Baseline	P1 (def)	P2 (ref)	P4 (deps)	P5 (dead)
naive-grep	0.31	0.54	0.34	0.21
smart-grep	0.60	0.81	0.63	0.55
Sverklo	0.75	0.56	0.86	0.02

B. Per-Category Performance

Table IV disaggregates by the four task categories.

Sverklo wins decisively on P1 (definition lookup, F1 0.75 vs. 0.60) and P4 (file dependencies, F1 0.86 vs. 0.63). The win on P4 is particularly large: smart-grep’s `cat <file>` plus import-line regex breaks down on dynamic imports, re-exports, and TypeScript path aliases that Sverklo’s AST-based dependency extraction handles correctly. Of 10 P4 tasks, *zero* pass smart-grep’s $F_1 \geq 0.8$ quality gate; eight pass Sverklo’s.

Sverklo loses on P2 (reference finding) and is currently uncompetitive on P5 (dead-code detection). The P2 gap is rooted in a specific recall failure: Sverklo’s symbol-graph stores references at the chunk-source granularity but does not always resolve aliased imports when the importing file uses a renamed binding. Smart-grep, by simply matching the symbol name across the working tree, returns all syntactic references including renamed ones; it pays for this in precision but not catastrophically. Closing this gap is a v0.18 priority.

The P5 result— F_1 0.02 with recall 0.60 and precision 0.01—is the largest negative finding in the paper. Sverklo’s `sverklo_audit` tool currently identifies orphan candidates via the symbol-reference graph but does not handle several cases common in Express-shaped codebases: re-exports through index files, dynamic dispatch tables, and decorator-based registration patterns. The tool finds *some* real orphans (recall is non-zero) but flags many false positives. The result reflects a deficiency in the current implementation rather than a property of the benchmark, and is therefore a clear community improvement target.

C. Token Efficiency at Equal Quality

The case for Sverklo at equal quality is concentrated on the structural categories (P1, P4). On P1 tasks that pass the $F_1 \geq 0.8$ gate, Sverklo consumes an average of 257 input tokens per task vs. smart-grep’s 187. On P4 tasks, Sverklo averages 90 tokens vs. smart-grep’s order-of-magnitude-higher 1,000+ for the runs that did pass (with the caveat that almost no smart-grep P4 runs pass the gate). The cold-start penalty (index build, 3,690 ms) is amortized after ~ 5 queries on these datasets.

D. bench:research: Internal Recall and Caveats

On 32 research-style questions over Sverklo’s own codebase, the investigate retriever (Section III-C) achieves 31/32 ($\sim 97\%$) perfect recall, with the single missing case (`sverklo-evidence-verify`) involving a 2-of-3

TABLE V

BENCH:SWE RESULTS BY REPOSITORY ($n = 13$ TASKS EACH). AVG RECALL IS THE MEAN PER-TASK RECALL OVER THE 13 TASKS; PERFECT RECALL COUNTS TASKS FOR WHICH THE TOP-50 RESULT SET COVERS ALL REQUIRED-EVIDENCE FILES.

Repository	Tasks	Avg recall	Perfect recall
Vite 6.0.7	13	88.5%	11/13
FastAPI 0.115.6	13	69.2%	8/13
Express 5.0.1	13	65.4%	7/13
NestJS 10.4.7	13	57.7%	6/13
Prisma 6.1.0	13	50.0%	6/13
Aggregate	65	66.2%	38/65 (58.5%)

evidence-file boundary case tracked since release v0.16. Result is deterministic across runs.

This number is presented with deliberate caution. The benchmark is internal: questions were written by the same team that wrote the code, and the retriever’s synonyms list (`src/search/synonyms.ts`) has been hand-tuned during development to address specific failures. The number is therefore a regression baseline—it confirms that the retriever does not regress on its training distribution—rather than evidence of generalizable retrieval quality. Generalizable claims require third-party reproduction on out-of-distribution questions, which is precisely the design intent of BENCH:SWE (Section IV-B).

E. *bench:swe*: Cross-Repository Recall

We executed BENCH:SWE across all five pinned repositories using release v0.17.1 and the default top- $k = 50$ ranking. Aggregate recall is reported in Table V; the harness clones each repository fresh on every run, so the numbers are reproducible by any third party with `npm run bench:swe`.

The aggregate—38/65 perfect recall, 66.2% mean recall—sits between the internal BENCH:RESEARCH ceiling (97%) and the weakest individual repository, demonstrating the gap that the out-of-distribution design was intended to expose. Vite, the most modern of the five repositories and the one with the cleanest src-tree organization, is the easiest target. Prisma, the most deeply nested, is the hardest.

Failure pattern: deeply-nested core files. Inspection of the 27 missed-evidence files reveals a consistent pattern. The questions that fail are not those whose answers live in obscure or rarely-edited files; they are questions whose canonical answer lives in a *central, heavily-imported* file that is co-referenced by nearly every component in the repository. Concrete examples:

- `lib/router/index.js` (Express): missed in 5 of 13 Express tasks (router-mount, error-middleware, app-mount, stream-disconnect, router-method-chain).
- `packages/core/injector/injector.ts` and `instance-loader.ts` (NestJS): missed in 4 tasks covering DI resolution, circular deps, request-scope, and lifecycle.
- `packages/client/src/runtime/core/engines/library/LibraryEngine.ts` (Prisma): missed in queries about query-engine IPC and connection pooling.

- `fastapi/applications.py` and `fastapi/routing.py` (FastAPI): missed in CORS and startup-shutdown queries.

These files share a structural property that defeats hybrid retrieval as currently implemented: PageRank pushes them up the overall importance ranking (every component imports them), but the per-query lexical and dense channels assign each individual question to the more specific feature file (e.g., a CORS-middleware query matches a CORS-middleware file by name) and miss the fact that the *logic* lives further upstream in the central god-file. Closing this gap likely requires either a co-import-aware re-ranking step or a separate “upstream traversal” channel that follows imports backward from feature files to their centralizing counterparts. We defer the implementation to v0.18 and report this finding as the most actionable single result of the cross-repository evaluation.

F. *Tree-sitter* vs. *Regex Parser Parity*

We compared the regex parser (default through v0.17) against the opt-in tree-sitter parser on Sverκλο’s 175 TypeScript source files. The regex parser identifies 748 named chunks; tree-sitter identifies 851. Their intersection is 664, yielding a Jaccard agreement of $664 / (748 + 851 - 664) \approx 71\%$. Tree-sitter additionally identifies 187 symbols missed by regex (predominantly class methods and nested function expressions); the regex parser produces 84 chunks not present in tree-sitter output, almost all of which are synthetic `module:_module` chunks holding file-level prose—a feature we deliberately retain because it materially improves retrieval of file-intent documentation.

Because flipping the default parser would change the chunk set underlying every retrieval ranking, and because BENCH:SWE results have not yet been collected against tree-sitter, v0.17 keeps regex as the default. We expect to flip in v0.18 once we confirm that BENCH:SWE and BENCH:RESEARCH recall does not regress after the switch. The parity script (`scripts/parity-check.mjs`) is open-sourced and produces the reproducible 71% baseline cited above.

VI. DISCUSSION

We now interpret the aggregate numbers and the methodological choices that produced them.

A. *Structural* vs. *Textual Queries*

The BENCH:PRIMITIVES results admit a simple summary. For queries whose answer is structural—“where is this defined,” “what does this file import, and what imports this file?”—a graph-aware indexer beats grep substantially on both quality and tokens. For queries whose answer is textual—“where else does the string X appear,” “what symbols are unused?”—an informed grep still wins, because the recall ceiling on a textual query is set by substring matching rather than by structural understanding, and grep implements substring matching exactly.

This decomposition suggests that the right deployment of Sverκλο in an agentic loop is *alongside*, not *instead of*, grep. The agent picks the tool whose native primitive matches the

question shape: `sverklo_lookup` and `sverklo_deps` for structural queries, `ast_grep` or shell `grep` for syntactic ones. The MCP standard makes this composition trivial; Sverklo’s recommended client configuration ships `ripgrep` [2] alongside its own tools.

B. Open Benchmarks as Shared Substrate

Vendor benchmarks in the LLM space have a well-documented selection bias: benchmarks are often introduced together with results showing the introducing system on top, and the methodology is sometimes only inspectable in the same publication that announces those results. `BENCH:SWE` and `BENCH:PRIMITIVES` are released separately from any specific Sverklo improvement claim. The harness, ground truth, and baseline implementations live in the open-source repository (`benchmark/`); third parties can reproduce the numbers in this paper by running `npm run bench:quick` after a clone, or extend them with new tools by implementing a 60-line baseline adapter against a documented `Task` interface. The honest aggregate result— F_1 0.58 vs. `smart-grep`’s 0.67—is the asset: it transforms the benchmark from a marketing artifact into a shared substrate. Improvements to Sverklo’s P2 and P5 performance will be measured against the same harness; competing systems will be measured against it as well.

C. Threats to Validity

Internal validity. The token estimator uses `chars/3.5`, which is comparable across baselines within this benchmark but does not match any specific BPE tokenizer (e.g., Anthropic’s). The quality gate ($F_1 \geq 0.8$) is arbitrary; we chose it to prevent “found nothing cheaply” but acknowledge that the threshold is not externally calibrated. Scoring tolerances (± 3 on P1, ± 2 on P2) protect against parser pedantry but may mask genuine off-by-one errors in callers. Ground truth was authored by a single team without independent inter-rater verification; this is a known limitation that future releases will address by recruiting external annotators for at least the `BENCH:SWE` questions.

External validity. Both benchmarks are small. Sixty tasks on two repositories is sufficient to surface failure modes but insufficient to ground claims about behavior on truly large codebases (e.g., Linux, Chromium). Repository selection skews toward popular, well-maintained open-source projects in two language ecosystems (JavaScript/TypeScript and Python); embedded systems, legacy codebases, and polyglot monorepos are not represented. Questions are in English. None of the baselines are agentic in the strong sense (none plan, none use chains of tools); a fully agentic baseline running, e.g., OpenHands [38] would be a more demanding comparator and is the most important piece of unfinished work.

Construct validity. The `BENCH:PRIMITIVES` categories (P1, P2, P4, P5) cover four operations but not, e.g., type-flow, control-flow, or concurrency reasoning. `BENCH:SWE` is intended to backstop this gap with open-ended questions but, as noted, its full headline numbers are pending.

D. Lessons for MCP-shaped Code Intelligence

Two design patterns crystallized during Sverklo’s development that appear to generalize beyond this system.

Single-binary, zero-config installation is a feature of the retrieval system. Tree-sitter binary builds fail often enough on fresh installs that we made the dependency optional with a regex fallback; this single decision, and the silent fall-back logic, is responsible for a non-trivial fraction of installation success in our telemetry. Code-intelligence tools that require multiple system dependencies, language-specific servers, or paid backends will be adopted accordingly less often, regardless of their on-paper quality.

Per-channel RRF outperforms global RRF in our setting. Standard hybrid retrieval runs one RRF pass over a single ranked list combining lexical and dense candidates. Anecdotally, this lets a long Markdown documentation chunk drown a four-line function at the same query relevance, and path-token matches (which are precision-skewed) get diluted in the combined ranking. The per-channel design is straightforward to implement (a few hundred lines) and to tune (each channel’s weight is a single scalar), and is a reasonable default for code-RAG deployments.

VII. CONCLUSION

We presented Sverklo, a local-first MCP code-intelligence server, and two reproducible benchmarks for evaluating it. `BENCH:PRIMITIVES` measures structural primitives on two repositories; on it Sverklo loses to a competently configured `grep` baseline on aggregate F_1 (0.58 vs. 0.67) but uses 65% fewer input tokens, and dominates on the structural categories (P1 definition lookup, P4 file dependencies). `BENCH:SWE` extends evaluation to 65 hand-curated research questions across five popular open-source projects in two language ecosystems; the harness and ground-truth files are publicly available and accept community contributions. We argue that publishing a benchmark on which the proposed system does not uniformly dominate is itself a methodological contribution, and release the system, harness, and 75 ground-truth files under MIT license to encourage third-party comparison.

Future work has four priorities. First, address the central god-file failure pattern surfaced by `BENCH:SWE` (Section V-E) via co-import-aware re-ranking or an upstream-traversal retrieval channel. Second, resolve aliased imports through the symbol graph to close the P2 reference-finding gap on `BENCH:PRIMITIVES`. Third, redesign `sverklo_audit`’s orphan detection to handle re-exports, dynamic dispatch, and decorator-based registration; the current F_1 0.02 on P5 is the largest single deficit in the system. Fourth, run `BENCH:SWE` against the `smart-grep` and `naive-grep` baselines and recruit independent annotators for at least one repository’s ground truth. Beyond v0.18, the broader open question is whether channelized fusion and bi-temporal memory generalize to non-MCP retrieval pipelines.

REPRODUCIBILITY

The Sverklo source, both benchmark suites, and all ground-truth files are released under the MIT license at <https://github.com/sverklo/sverklo>. Numbers in this paper come from release v0.17.1 (April 2026) and the result file `benchmark/results/2026-04-07T23-07-14-211Z`. Reported measurements were collected on an Apple M-series laptop with 16 GB RAM running macOS 14, Node.js 20.11, and npm 10.2; the embedding model and PageRank parameters are fixed in v0.17.1, and all reported runs are deterministic across repeated executions. To reproduce BENCH:PRIMITIVES, install Node.js ≥ 20 and run `npm install && npm run bench:quick` from the repository root; output is written to `benchmark/results/`. To reproduce BENCH:SWE, run `npm run bench:swe`; the harness clones each pinned repository into `benchmark/.cache/`, indexes it, and writes per-question results to a timestamped output directory.

REFERENCES

- [1] Microsoft, “Language server protocol specification,” <https://microsoft.github.io/language-server-protocol/>, 2016.
- [2] A. Gallant, “ripgrep: Recursively search directories with a regex pattern,” <https://github.com/BurntSushi/ripgrep>, 2016.
- [3] S. Robertson and H. Zaragoza, “The probabilistic relevance framework: BM25 and beyond,” in *Foundations and Trends in Information Retrieval*, 2009.
- [4] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, “MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [5] G. V. Cormack, C. L. A. Clarke, and S. Büttcher, “Reciprocal rank fusion outperforms Condorcet and individual rank learning methods,” in *Proc. ACM SIGIR*, 2009.
- [6] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “SWE-bench: Can language models resolve real-world GitHub issues?” in *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- [7] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “SWE-agent: Agent-computer interfaces enable automated software engineering,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
- [8] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, “AutoCodeRover: Autonomous program improvement,” in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2024.
- [9] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Agentless: Demystifying LLM-based software engineering agents,” in *Proc. International Conference on Machine Learning (ICML)*, 2025.
- [10] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu, “RepoGraph: Enhancing AI software engineering with repository-level code graph,” in *Proc. International Conference on Learning Representations (ICLR)*, 2025.
- [11] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *Proc. IEEE Symposium on Security and Privacy*, 2014.
- [12] Anthropic, “Model context protocol (specification),” <https://modelcontextprotocol.io/specification>, 2024.
- [13] X. Hou, Y. Zhao, S. Wang, and H. Wang, “Model context protocol (MCP): Landscape, security threats, and future research directions,” 2025.
- [14] Greptile, “Greptile: AI code search and review,” <https://www.greptile.com/>, 2024.
- [15] Sourcegraph, “Cody: AI coding assistant,” <https://sourcegraph.com/cody>, 2024.
- [16] Anysphere, “Cursor: The AI code editor,” <https://www.cursor.com/>, 2024.
- [17] P. Gauthier, “Aider repository map,” <https://aider.chat/docs/repomap.html>, 2024.
- [18] C. Packer, S. Wooders, K. Lin, V. Fang, S. Patil, I. Stoica, and J. E. Gonzalez, “MemGPT: Towards LLMs as operating systems,” in *Proc. Conference on Language Modeling (COLM)*, 2024.
- [19] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, “Voyager: An open-ended embodied agent with large language models,” *Transactions on Machine Learning Research (TMLR)*, 2023.
- [20] J. S. Park, J. C. O’Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, “Generative agents: Interactive simulacra of human behavior,” in *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, 2023.
- [21] M. Brunsfeld, A. Turnbull, P. Thomson, A. Saratchandran, A. Hlynyskiy et al., “Tree-sitter: An incremental parsing system for programming tools,” <https://tree-sitter.github.io/>, 2018.
- [22] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” 2021.
- [23] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, “Program synthesis with large language models,” 2021.
- [24] T. Liu, C. Xu, and J. McAuley, “RepoBench: Benchmarking repository-level code auto-completion systems,” in *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- [25] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang, “Cross-CodeEval: A diverse and multilingual benchmark for cross-file code completion,” in *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.
- [26] E. Bogomolov, A. Eliseeva, T. Galimzyanov, E. Glukhov, A. Shapkin, M. Tigina, Y. Golubev, A. Kovrigin, A. van Deursen, M. Izadi, and T. Bryksin, “Long Code Arena: A set of benchmarks for long-context code models,” 2024.
- [27] Z. Z. Wang, A. Asai, X. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried, “CodeRAG-Bench: Can retrieval augment code generation?” in *Findings of the Association for Computational Linguistics: NAACL*, 2025.
- [28] OpenAI, “Introducing SWE-bench Verified,” <https://openai.com/index/introducing-swe-bench-verified/>, 2024.
- [29] J. Yang, C. E. Jimenez, A. L. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. R. Narasimhan, D. Yang, S. I. Wang, and O. Press, “SWE-bench Multimodal: Do AI systems generalize to visual software domains?” in *Proc. International Conference on Learning Representations (ICLR)*, 2025.
- [30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” in *Findings of EMNLP*, 2020.
- [31] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “GraphCodeBERT: Pre-training code representations with data flow,” in *Proc. International Conference on Learning Representations (ICLR)*, 2021.
- [32] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “UniXcoder: Unified cross-modal pre-training for code representation,” in *Proc. Annual Meeting of the Association for Computational Linguistics (ACL)*, 2022.
- [33] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, “RepoCoder: Repository-level code completion through iterative retrieval and generation,” in *Proc. Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.
- [34] H. N. Phan, H. N. Phan, T. N. Nguyen, and N. D. Q. Bui, “RepoHyper: Search-expand-refine on semantic graphs for repository-level code completion,” 2024.
- [35] Y. Zhang, X. Zhao, Z. Z. Wang, C. Yang, J. Wei, and T. Wu, “cAST: Enhancing code retrieval-augmented generation with structural chunking via abstract syntax tree,” 2025.
- [36] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao, “ReAct: Synergizing reasoning and acting in language models,” in *Proc. International Conference on Learning Representations (ICLR)*, 2023.

- [37] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, “Toolformer: Language models can teach themselves to use tools,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [38] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, “OpenHands: An open platform for AI software developers as generalist agents,” in *Proc. International Conference on Learning Representations (ICLR)*, 2025.
- [39] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji, “Executable code actions elicit better LLM agents,” in *Proc. International Conference on Machine Learning (ICML)*, 2024.
- [40] A. Zaidman and S. Demeyer, “Automatic identification of key classes in a software system using webmining techniques,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, 2008.
- [41] W. Pan, D. Lo, X. Xia, and A. Zaidman, “A PageRank-based recommender system for identifying key classes in software systems,” in *ICSE Companion*, 2013.
- [42] T. A. Wagner and S. L. Graham, “Efficient and flexible incremental parsing,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 5, 1998.
- [43] ONNX Runtime developers, “ONNX Runtime: Cross-platform, high-performance ML inferencing and training accelerator,” <https://onnxruntime.ai/>, 2018.
- [44] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” Stanford InfoLab Technical Report, 1999.