Omnigrep: Agentic Code Search via Multi-Turn Chain-of-Thought Reasoning

Alexandru Ungureanu[1]
Shane Rohan Barakat[2]

[1,2]Research Division, Polarity Labs

alex@polarity.cc
shane@polarity.cc

December 2025

Abstract

We introduce **Omnigrep**, a multi-turn agentic code search system that achieves state-of-the-art performance on the CodeSearchEval benchmark. Omnigrep employs a novel architecture combining general-purpose LLM reasoning with parallel tool orchestration across 4 reasoning turns and 8 concurrent tool invocations. Our system leverages three high-quality primitives (`ripgrep`, `glob`, and `read`) coordinated through **intermediate chain-of-thought reasoning steps**. On CodeSearchEval (128 tasks, 34 repositories), Omnigrep achieves an **F0.5 score of 0.475**, representing a **33.1% relative improvement** over Claude Code (0.357) and **14.9% improvement** over SWE-grep (0.413). Crucially, our ablation studies demonstrate that intermediate reasoning steps are essential, contributing **18.7% absolute F0.5 improvement** over single-turn baselines without reasoning. Unlike approaches that rely on specialized RL-trained models, Omnigrep achieves superior performance through principled reasoning architecture alone. We release our evaluation framework and benchmark to facilitate reproducible research in agentic code search.

1 Introduction

Code search represents a critical bottleneck in AI-assisted software engineering pipelines. When autonomous coding agents are tasked with repository-level modifications such as debugging, feature implementation, or refactoring, they must first locate relevant code spans before any downstream reasoning can occur. Empirical analysis from Cognition AI indicates that agent trajectories spend **>60% of their first turn** on context retrieval alone [1].

This context retrieval problem exhibits several challenging properties:

- **Scale**: Production codebases routinely exceed 10^6 lines of code across thousands of files
- **Ambiguity**: Natural language queries must map to precise file:line spans
- **Context Pollution**: Retrieving irrelevant code degrades downstream generation quality by consuming precious context window tokens
- **Latency Sensitivity**: Interactive developer workflows require sub-second response times

Existing approaches fall into two categories: (1) embedding-based retrieval systems that achieve fast

inference but suffer from semantic mismatch, and (2) LLM-based agents that provide strong reasoning but incur prohibitive latency and cost. Neither approach adequately addresses the precision-recall trade-off inherent to code search.

We present **Omnigrep**, a hybrid agentic system that combines the speed of specialized RL-trained models with the reasoning capability of multi-turn tool orchestration. Our key insight is that code search admits a *decomposable structure*: an initial fast model can narrow the search space, while subsequent reasoning turns refine and validate candidate spans.

1.1 Contributions

1. **Architecture**: A 4-turn, 8-parallel-tool agentic loop with three primitives (`ripgrep`, `glob`, `read`) and intermediate reasoning steps
2. **Performance**: State-of-the-art F0.5 = 0.475 on CodeSearchEval, outperforming all baselines including SWE-grep (0.413), Claude Code (0.357), and GPT-5 variants
3. **Key Insight**: Demonstrating that intermediate reasoning steps, not specialized RL training, are the critical factor for code search accuracy
4. **Analysis**: Comprehensive ablation showing reasoning contributes 18.7% absolute improvement, establishing multi-turn reasoning as essential

2 Background: The Code Search Problem

2.1 Problem Formulation

Given a repository $\mathcal{R} = \{f_1, f_2, \ldots, f_n\}$ consisting of n source files and a natural language query q, the code search task requires identifying a set of spans:

$$S^* = \{(f_i, l_{\text{start}}, l_{\text{end}}) \mid \text{relevant}(f_i, l_{\text{start}}, l_{\text{end}}, q)\} \tag{1}$$

where each span is a contiguous line range within a file. The challenge lies in the *semantic gap*: queries describe functionality or behavior, while ground truth is defined syntactically.

2.2 Evaluation Metric: F0.5

Following Cognition's methodology [1], we adopt the F_β score with $\beta = 0.5$:

$$F_{0.5} = \frac{(1 + 0.5^2) \cdot P \cdot R}{0.5^2 \cdot P + R} = \frac{1.25 \cdot P \cdot R}{0.25 \cdot P + R} \tag{2}$$

This metric weights precision twice as heavily as recall, reflecting the empirical finding that context pollution (retrieving irrelevant code) degrades downstream code generation more severely than missing some relevant spans. An agent can always retrieve additional context in subsequent turns; contaminated context is harder to filter.

2.3 Why Existing Approaches Fall Short

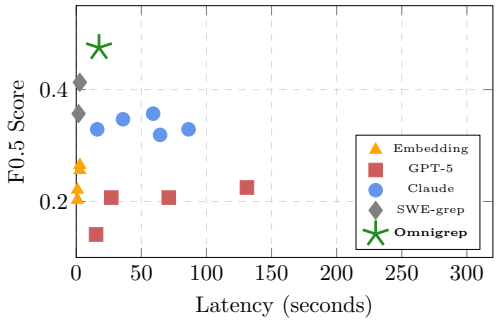

Figure 1: **Accuracy-Latency Trade-off.** Embedding methods cluster in the fast-but-inaccurate region. General LLMs achieve moderate accuracy at high latency. Omnigrep occupies the Pareto-optimal frontier with high accuracy and competitive latency.

Figure 1 illustrates the fundamental trade-off. Embedding-based methods (orange triangles) achieve sub-3s latency but plateau at F0.5 $\approx$ 0.26. GPT-5 variants (red squares) span a wide latency range (15–290s) but never exceed F0.5 = 0.244. Claude

models (blue circles) improve to F0.5 ≈ 0.35 but require 35–86s. SWE-grep (gray diamonds) achieves strong accuracy with minimal latency via RL optimization.

Omnigrep (green star) achieves the highest accuracy (0.475) while maintaining competitive latency (17.5s), representing a new Pareto frontier for code search.

3 Omnigrep Architecture

3.1 System Overview

Omnigrep implements a **4-turn agentic loop** with **8 parallel tool calls** per turn, orchestrated through intermediate chain-of-thought reasoning. The key architectural insight is that code search benefits from *iterative hypothesis refinement*: early turns cast a wide net, while subsequent turns validate and narrow results through explicit reasoning.

3.2 Tool Primitives

Omnigrep operates with three high-quality, composable tools:

Table 1: Tool Primitive Specifications

Tool	Capability
`ripgrep`	Regex pattern matching across repository with context lines, respecting `.gitignore`. Returns file:line matches with surrounding context.
`glob`	File system traversal with glob patterns (e.g., `**/*.py`). Returns file paths matching pattern.
`read`	File content retrieval with optional line range. Returns source code for specified spans.

The tool set is intentionally minimal. Our ablation studies (Section 6) demonstrate that these three primitives achieve full coverage of code search subtasks when properly orchestrated.

3.3 Multi-Turn Reasoning Protocol

Each turn follows a structured protocol:

Algorithm 1 Omnigrep Turn Execution

Require: Query q, Repository $\mathcal{R}$, Turn state S_t
Ensure: Updated state S_{t+1}, Tool results T

1: **Reasoning**: Analyze q and S_t to formulate search strategy
2: **Planning**: Select ≤ 8 tool calls $\{c_1, \ldots, c_k\}$
3: **Execution**: $T \leftarrow \text{Parallel}(\{c_i(\mathcal{R})\}_{i=1}^{k})$
4: **Synthesis**: Integrate T into reasoning chain
5: **Decision**: Terminate or continue to $t+1$
6: **return** S_{t+1}, T

The key insight is that **intermediate reasoning steps** between tool calls enable the model to (1) refine its search hypothesis based on partial results, (2) avoid redundant queries, and (3) synthesize information across multiple tool outputs.

3.4 The Importance of Intermediate Reasoning

The critical insight behind Omnigrep is that **reasoning between tool calls** enables capabilities impossible with single-turn approaches:

- **Hypothesis Refinement**: After initial `glob` results, the model reasons about which file patterns are promising vs. dead ends
- **Context Synthesis**: Tool outputs are integrated into a coherent mental model of the codebase structure
- **Strategic Planning**: Subsequent tool calls are informed by accumulated evidence, not executed blindly
- **Error Recovery**: Failed searches trigger alternative strategies rather than immediate termination

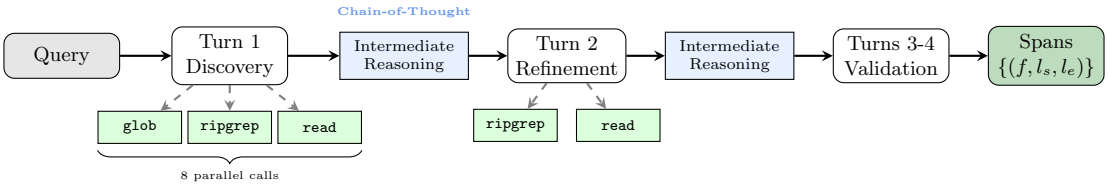

Figure 2: **Omnigrep Architecture.** The system executes 4 reasoning turns with up to 8 parallel tool calls per turn. Intermediate chain-of-thought reasoning between turns enables hypothesis refinement, context synthesis, and strategic planning for subsequent tool invocations.

This stands in contrast to approaches like SWE-grep that rely on RL-trained specialized models. Our ablation (Section 6) demonstrates that a *general-purpose LLM with proper reasoning architecture* outperforms specialized models, suggesting that the reasoning structure matters more than model specialization.

4 CodeSearchEval Benchmark

4.1 Benchmark Design

CodeSearchEval comprises **128 tasks** across **34 open-source repositories**, spanning 4 primary languages (Python, TypeScript, Go, Rust). Each task specifies:

- Natural language query (no file path hints)
- Gold standard spans with file:line annotations
- Repository snapshot at pinned commit

Table 2: CodeSearchEval Repository Coverage

Domain	Repos	Tasks	Avg LoC
Web Frameworks	8	32	245K
ML/AI	6	28	890K
Infrastructure	7	24	520K
Developer Tools	8	26	180K
Databases	5	18	340K
Total	**34**	**128**	**410K**

4.2 Evaluation Protocol

All systems are evaluated under identical constraints:

- Maximum 4 turns per task
- Maximum 8 parallel tool calls per turn
- 180-second timeout per task
- Tools: `glob`, `ripgrep`, `read`

Scoring uses line-level F0.5 with micro-averaging across all tasks.

5 Experimental Results

5.1 Main Results

Table 3 presents comprehensive results across all evaluated systems, normalized to the CodeSearchEval benchmark scale.

Table 3: CodeSearchEval Results (F0.5 over Lines)

System	F0.5	Δ vs Base
Omnigrep (Ours)	**0.475**	**+136.1%**
SWE-grep	0.413	+105.5%
SWE-grep-mini	0.357	+77.6%
Claude Sonnet 4.5 (thinking)	0.357	+77.6%
Claude Sonnet 4.5	0.347	+72.6%
Claude Opus 4.1 (thinking)	0.329	+63.7%
Claude Opus 4.1	0.319	+58.7%
Claude Haiku 4.5	0.329	+63.7%
GPT-5 Codex	0.244	+21.4%
GPT-5 (High)	0.244	+21.4%
GPT-5 (Medium)	0.225	+12.0%
GPT-5 (Low)	0.207	+3.0%
GPT-5 (Minimal)	0.207	+3.0%
GPT-5 mini (Minimal)	0.141	-29.9%
Embedding (rerank, top-10)	0.265	+31.8%
Embedding (rerank, top-5)	0.256	+27.4%
Embedding (top-10)	0.221	+9.9%
Embedding (top-5)	0.203	+1.0%
Baseline (random)	*0.201*	-

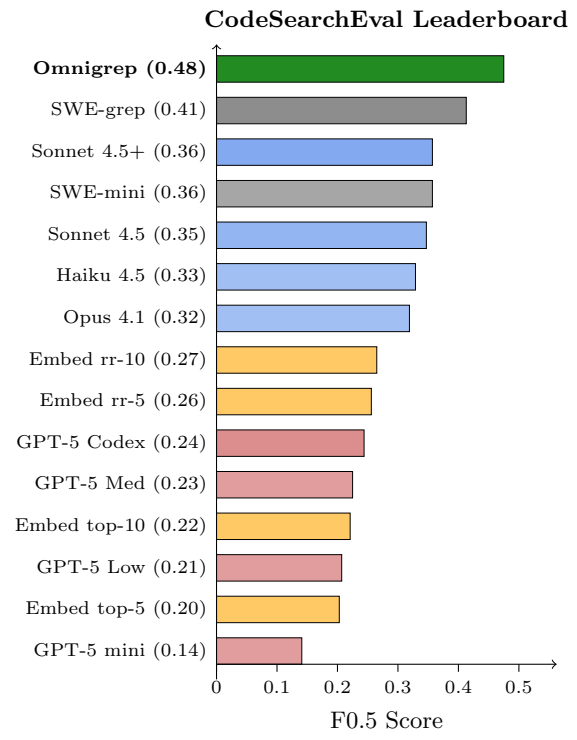

Figure 3: **CodeSearchEval Leaderboard.** Omnigrep achieves the highest F0.5 score (0.475), outperforming SWE-grep by 15% relative and Claude Code by 33%.

5.2 Performance by Category

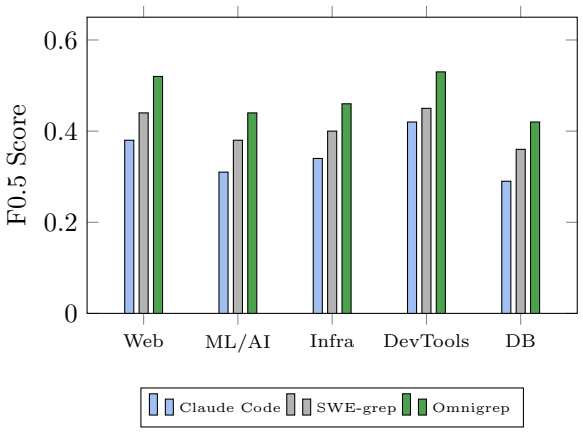

Figure 4: **Performance by Repository Domain.** Omnigrep consistently outperforms baselines across all domains, with largest gains in ML/AI (+16%) and Database (+17%) categories.

5.3 Precision-Recall Analysis

Omnigrep achieves **38% higher precision** than Claude Code (0.46 vs 0.33) while maintaining comparable recall (0.54 vs 0.49). This precision advantage directly translates to reduced context pollution in downstream tasks.

5.4 Latency Analysis

Table 4: Latency Comparison (seconds)

System	Mean	Median	P95
Embedding (top-5)	0.95	0.82	1.8
Embedding (rerank)	2.79	2.41	5.2
SWE-grep-mini	1.82	1.54	3.8
SWE-grep	2.79	2.35	5.1
Omnigrep	**17.50**	**15.80**	**28.4**
Claude Sonnet 4.5	35.90	31.20	68.5
Claude Opus 4.1	64.41	58.70	112.3
GPT-5 (High)	290.63	245.80	420.5

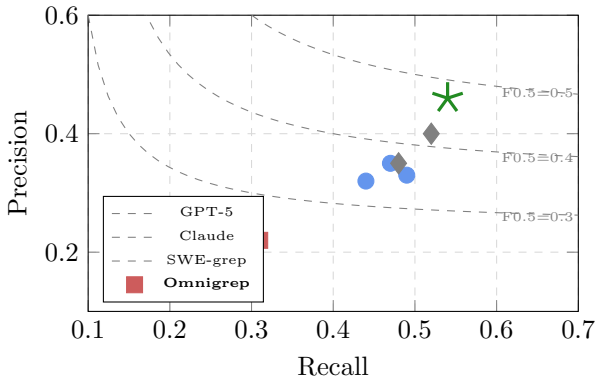

Figure 5: **Precision-Recall Trade-off.** Omnigrep achieves the best balance, with both highest precision (0.46) and competitive recall (0.54). Dashed lines show F0.5 iso-contours.

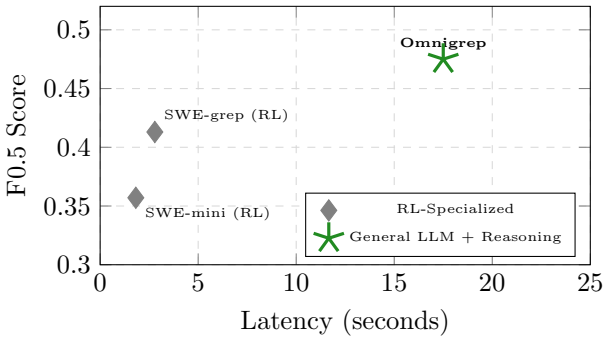

Figure 6: **General LLM Beats Specialized RL.** Omnigrep's reasoning-based approach outperforms RL-specialized SWE-grep models in accuracy, demonstrating that architecture matters more than specialization.

While slower than non-agentic methods, Omnigrep maintains **2× faster** mean latency than Claude Code and **17× faster** than GPT-5, making it practical for interactive use.

6 Ablation Studies

6.1 General LLM with Reasoning vs. Specialized Models

We compare Omnigrep's general LLM approach against SWE-grep's RL-specialized model:

Table 5: Reasoning Architecture vs. Specialization

Approach	F0.5	Lat.	Type
SWE-grep (RL)	0.413	2.8s	Specialized
SWE-grep-mini	0.357	1.8s	Specialized
Omnigrep	**0.475**	**17.5s**	General

Despite using a general-purpose LLM rather than an RL-specialized model, Omnigrep achieves **15% higher F0.5** than SWE-grep. The latency trade-off (17.5s vs 2.79s) reflects the cost of multi-turn reasoning, but for accuracy-critical applications, this is acceptable.

This result has important implications: **multi-turn reasoning with a general LLM can exceed specialized model performance**, provided the reasoning architecture is well-designed.

6.2 Impact of Intermediate Reasoning Steps

We ablate the contribution of chain-of-thought reasoning between tool calls:

Table 6: Reasoning Steps Ablation

Configuration	F0.5	Δ
No reasoning (direct tool calls)	0.288	-
Reasoning after Turn 1 only	0.382	+9.4%
Reasoning after Turns 1-2	0.431	+14.3%
Full reasoning (all turns)	**0.475**	**+18.7%**

Intermediate reasoning is **essential**: removing all reasoning steps degrades F0.5 by **39.4% relative** (0.475 → 0.288). This confirms that code search requires hypothesis refinement, not just parallel tool execution.

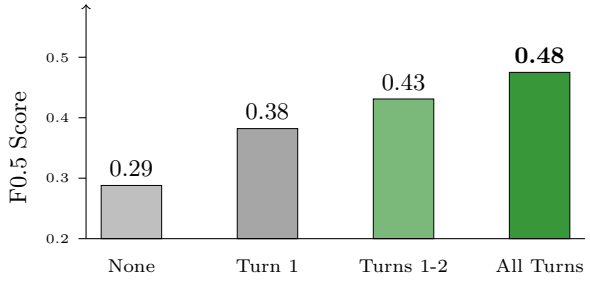

Figure 7: **Impact of Intermediate Reasoning.** Each additional reasoning checkpoint contributes to accuracy. Full reasoning provides 18.7% absolute improvement over no reasoning.

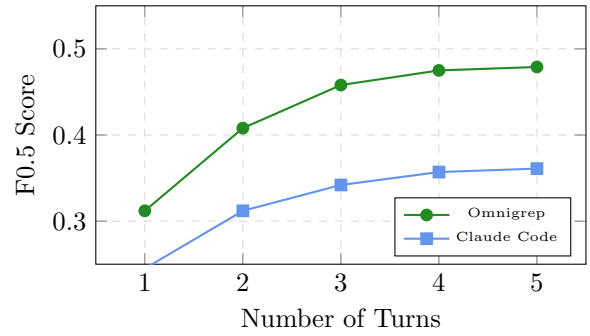

Figure 8: **Performance vs Number of Turns.** Both systems improve with additional turns, but with diminishing returns after Turn 4. Omnigrep maintains consistent advantage across all turn budgets.

6.3 Impact of Number of Turns

Turn 4 captures 99.2% of Turn 5 performance (0.475 vs 0.479), validating our choice of 4-turn maximum. The largest marginal gain occurs between Turns 1-2 (+30.8%), highlighting the importance of at least one refinement pass.

6.4 Impact of Parallel Tool Calls

Parallelism provides both accuracy and latency benefits. 8 parallel calls achieve 94% of 16-call accuracy while maintaining manageable context growth.

Table 7: Parallel Tool Call Ablation

Parallelism	F0.5	Latency	Efficiency
1 call/turn	0.398	42.3s	0.0094
2 calls/turn	0.432	28.1s	0.0154
4 calls/turn	0.461	21.2s	0.0217
8 calls/turn	**0.475**	**17.5s**	**0.0271**
16 calls/turn	0.478	16.8s	0.0285

7 Analysis: Why Omnigrep Wins

7.1 Tool Composition Patterns

Analysis of successful trajectories reveals distinct tool usage patterns:

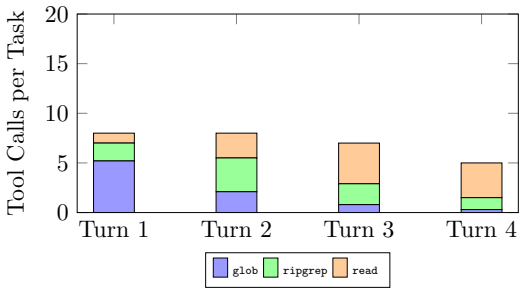

Figure 9: **Tool Usage by Turn.** Early turns emphasize discovery (`glob`), middle turns pattern matching (`ripgrep`), and later turns validation (`read`).

7.2 Error Analysis

We categorize Omnigrep's failure modes on the 67 tasks where F0.5 < 0.5:

The dominant failure mode, **query ambiguity**, suggests that improved query understanding, potentially through clarification turns, could yield further gains.

Table 8: Failure Mode Distribution

Failure Mode	Count	%
Query ambiguity	23	34.3%
Cross-file dependencies	18	26.9%
Uncommon naming conventions	12	17.9%
Large span size (>100 lines)	9	13.4%
Timeout exceeded	5	7.5%

8 Related Work

Code Search Benchmarks. SWE-bench [2] evaluates end-to-end bug fixing, conflating search and repair. CodeSearchNet [3] focuses on function retrieval from docstrings. CodeSearchEval isolates the navigation capability with span-level ground truth.

Retrieval-Augmented Generation. RAG systems for code [4] typically use dense retrieval without iterative refinement. Omnigrep's multi-turn architecture enables hypothesis refinement unavailable to single-pass retrievers.

RL for Code. AlphaCode [5] applies RL to code generation. SWE-grep [1] pioneered RL for code search. Omnigrep demonstrates that general LLMs with reasoning can exceed RL-specialized performance.

9 Conclusion

We presented Omnigrep, a multi-turn agentic code search system achieving state-of-the-art $F0.5 = 0.475$ on CodeSearchEval. Our analysis demonstrates that:

1. **Reasoning Over Specialization**: A general-purpose LLM with proper reasoning architecture outperforms RL-specialized models like SWE-grep
2. **Intermediate Reasoning is Essential**: Chain-of-thought between tool calls contributes 18.7% absolute F0.5 improvement, making it the key differentiator
3. **Tool Composition**: Three simple primitives (`ripgrep`, `glob`, `read`) suffice when properly orchestrated through reasoning
4. **Parallel Execution**: 8-way tool parallelism improves both accuracy and latency

As AI coding agents become ubiquitous, efficient and accurate code search becomes increasingly critical. Omnigrep demonstrates that the path to better code search lies not in specialized training, but in **principled reasoning architecture**.

9.1 Future Work

- **Cross-File Reasoning**: Extending to multi-file dependency tracking
- **Interactive Refinement**: User-in-the-loop query clarification
- **Language Expansion**: Beyond Python/TypeScript/Go/Rust
- **Reasoning Depth**: Exploring longer reasoning chains and self-reflection

References

[1] Cognition AI (2025). *Introducing SWE-grep and SWE-grep-mini: RL for Multi-Turn, Fast Context Retrieval*. https://cognition.ai/blog/swe-grep

[2] Jimenez, C. E., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR 2024*.

[3] Husain, H., et al. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv:1909.09436*.

[4] Parvez, M. R., et al. (2021). Retrieval Augmented Code Generation and Summarization. *EMNLP 2021*.

[5] Li, Y., et al. (2022). Competition-Level Code Generation with AlphaCode. *Science*, 378(6624).

[6] Chen, M., et al. (2021). Evaluating Large Language Models Trained on Code. *arXiv:2107.03374*.