StreamRAG v2: Native Parser Daemons and Three-Tier Extraction for Real-Time Code Graphs

Krrish Choudhary*
Department of Computer Science, The LNM Institute of Information Technology, Jaipur, India

***Abstract*: We present StreamRAG v2, a major evolution of the StreamRAG real-time incremental code graph system for AI-driven code editors. Building on the v1 foundations—LiquidGraph, DeltaGraphBridge, and ShadowAST—v2 introduces five architectural advances: (1) Native Parser Daemons, persistent language-specific processes (Node.js, Rust, JVM) communicating via length-prefixed msgpack over Unix domain sockets, providing full AST parsing for 7 non-Python languages; (2) a Three-Tier Extraction Hierarchy where each language is served by the best available parser—native daemon, tree-sitter AST, or regex—with automatic fallback; (3) a V2 Subsystem comprising adaptive debouncing, context stabilization, fine-grained invertible graph operations, semantic path addressing, time-based file watching, and multi-session daemon support; (4) three Intelligence Subsystems—native parser caching via SHA256-keyed LRU, semantic similarity scoring with five weighted signals, and LLM-guided resolution for ambiguous edges; and (5) Accuracy Subsystems including cross-language convention resolvers, confidence-filtered cycle detection, structured direct/transitive impact separation, and workspace-aware dead code analysis. StreamRAG v2 achieves sub-0.05ms per incremental change, supports 1,020 test scenarios across 62 test files with 100% pass rate, and spans 20,000+ lines of production code across Python, TypeScript, Rust, and Java. Real-world benchmarks show v2 reduces token consumption by 47% and turns by 19% compared to unaugmented Claude, with a 96% token reduction on complex architecture traces. Compared to v1, v2 improves accuracy by +20pp (92% vs 72%), runs 2.3× faster, and uses 53% fewer tokens.**

***Keywords*: Code Understanding, Knowledge Graphs, Incremental Parsing, Native Parser Daemons, Real-Time Systems, AI Code Assistants, Dependency Analysis, Abstract Syntax Trees, Convention Resolution, Dead Code Detection.**

1. Introduction

AI-powered code editors—GitHub Copilot, Cursor, and Claude Code—require deep semantic understanding of codebases to provide intelligent suggestions. Traditional approaches face a fundamental tension: the need for global consistency (understanding cross-file dependencies) versus local responsiveness (sub-100ms latency for real-time interaction).

StreamRAG v1 addressed this with an incremental code graph maintained by a persistent daemon, achieving sub 0.05ms updates via semantic gating and delta computation. However, v1 relied on regex-based extraction for all non-Python languages—adequate for basic patterns but unable to handle nested generics, complex scoping, or macro invocations.

A. *The Multi-Language Parsing Challenge*

Consider a Rust codebase with trait implementations, lifetime parameters, and macro invocations, or a TypeScript project with deeply nested generics and JSX components. Regex-based extraction inevitably produces false positives, misses scoped entities, and fails on language constructs that require recursive descent parsing. Yet embedding full parsers into the Python daemon would impose significant startup cost and cross-language FFI complexity.

B. *Contributions*

StreamRAG v2 makes the following contributions over v1:

1) Native Parser Daemons: Three persistent daemon processes—Node.js (TypeScript/JavaScript), Rust binary (Rust/C/C++), and JVM (Java)—each providing full AST parsing over a length-prefixed msgpack Unix socket protocol.
2) Three-Tier Extraction: A graceful degradation hierarchy (native daemon → tree-sitter AST → regex fallback) that automatically selects the best available parser at runtime.
3) V2 Subsystem: Adaptive keystroke debouncing, context stabilization, fine-grained invertible graph operations, semantic path addressing, time-based file watching, and multi-session daemon support.
4) Intelligence Subsystems: SHA256-keyed LRU caching, a five-signal semantic similarity scorer for cross-file edge ranking, and an LLM-guided resolution queue for ambiguous edges.
5) Accuracy Subsystems: Cross-language convention resolvers, confidence-filtered cycle detection, structured impact separation, and workspace-aware dead code analysis.
6) Expanded Evaluation: 1,020 passing tests (+71% from v1's 597), 62 test files, and 20,000+ lines of production code across 4 implementation languages.

*Corresponding author: krrishchoudhary109@gmail.com

2. Background and Related Work

A. Retrieval-Augmented Generation

RAG [1] augments language model generation with retrieved context. Given input x, retrieved documents z, and generated output y:

$$P(y|x) = \Sigma\, P(z|x) \cdot P(y|x, z)$$

B. Graph-Based Code Understanding

Prior work on code graphs includes CodeGraph [2] for static call graphs, GRACE [3] for graph-based embeddings, and RepoMap [4] for repository-level mapping. All treat graph construction as offline batch processing, requiring full rebuilds on each change.

C. Incremental Parsing and Language Servers

Tree-sitter [5] pioneered incremental parsing for syntax highlighting but focuses on single-file syntax trees, not cross-file semantic graphs. The Language Server Protocol [6] provides incremental diagnostics but lacks the graph abstraction needed for dependency analysis. StreamRAG v2 leverages tree-sitter as a fallback tier while providing native daemon parsers as the primary tier—extending incremental single-file parsing to cross-file semantic graph updates.

3. Problem Formalization

A. Code Knowledge Graph

A code knowledge graph is a labeled, directed multigraph G = (V, E, φ, ψ) where V represents code entity nodes (functions, classes, imports, variables), E ⊆ V × V × R are typed directed edges, φ maps nodes to types {function, class, import, variable, module_code}, and ψ maps edges to relations {calls, imports, inherits, uses_type, decorated_by}.

B. Incremental Graph Synchronization

Given a code change c_i at time t_i modifying file f, compute graph update ΔG_i such that:

$$G\_i = G\_\{i\text{-}1\} \oplus \Delta G\_i$$

minimizing the latency-staleness objective:

$$L = \Sigma\, (\alpha \cdot Latency(c\_i) + \beta \cdot |\Delta V\_missed|)$$

C. Three-Tier Extraction

For a source file f with language ℓ, the extraction function is:
Extract(f) = D_ℓ(f) if native daemon alive, T_ℓ(f) if tree-sitter available, R_ℓ(f) regex fallback
where each tier produces the same ASTEntity schema, enabling transparent substitution.

4. System Architecture

Fig. 1 presents the v2 architecture. The primary additions over v1 are the Native Parser Cluster (three persistent daemon processes) and the V2 Subsystem (debouncer, context stabilizer, fine-grained operations).

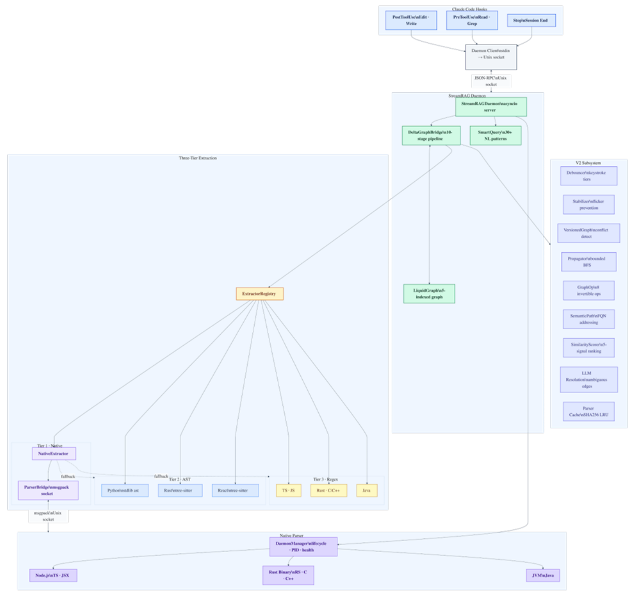
Claude Code Hooks
PostToolUse\nEdit · Write
PreToolUse\nRead · Grep
Stop\nSession End
Daemon Client\nstdin → Unix socket
JSON-RPC\nUnix socket
StreamRAG Daemon
StreamRAGDaemon\nasyncio server
DeltaGraphBridge\n10-stage pipeline
SmartQuery\n30+ NL patterns
LiquidGraph\n5-indexed graph
V2 Subsystem
Debouncer\nkeystroke tiers
Stabilizer\nflicker prevention
VersionedGraph\nconflict detect
Propagator\nbounded BFS
GraphOp\n8 invertible ops
SemanticPath\nFQN addressing
SimilarityScorer\n5-signal ranking
LLM Resolution\nambiguous edges
Parser Cache\nSHA256 LRU
Three-Tier Extraction
ExtractorRegistry
Tier 1 · Native
NativeExtractor
ParserBridge\nmsgpack socket
fallback
fallback
Tier 2 · AST
Python\nstdlib ast
Rust\ntree-sitter
React\ntree-sitter
Tier 3 · Regex
TS · JS
Rust · C/C++
Java
msgpack\nUnix socket
Native Parser
DaemonManager\nlifecycle · PID · health
Node.js\nTS · JSX
Rust Binary\nRS · C · C++
JVM\nJava

Fig. 1. StreamRAG v2 full system architecture. Claude Code hooks feed into the StreamRAG Daemon, driving the 10-stage DeltaGraphBridge pipeline. The Three-Tier Extraction system routes each language through native parser daemons, tree-sitter AST, or regex extractors. The V2 Subsystem provides adaptive debouncing, context stabilization, versioned operations, and bounded propagation

5. Core Components (V1 Foundations)

A. LiquidGraph: In-Memory Graph Engine

The LiquidGraph maintains five concurrent indexes enabling O(1) node lookups and O(k) file-scoped queries via index intersection. Five edge types (calls, imports, inherits, uses_type, decorated_by) are tracked with confidence metadata.

$$G = \langle N, I\_f, I\_t, I\_n, E\_out, E\_in \rangle$$

B. DeltaGraphBridge: Incremental Pipeline

The 10-stage pipeline (Fig. 2) processes each code change in O(Δ): semantic gate → delta computation → removals → additions → modifications → two-pass edge resolution → caches → versions → propagation → hierarchy. Position-based rename detection and import-aware 8-level target resolution remain unchanged.

C. Supporting Components

ShadowAST recovers entities from broken code via binary-search partial parsing with confidence scoring. Hierarchical Zones classify files as HOT/WARM/COLD for update prioritization. Bounded Propagation prevents dependency avalanches via three-phase sync/async/deferred processing. VersionedGraph tracks per-file version vectors for conflict detection.

6. Native Parser Daemons

The central contribution of v2 is the native parser daemon architecture: persistent language-specific processes that provide full AST parsing over Unix domain sockets.

A. Architecture Overview

Each daemon is a standalone process in its native language runtime (Table 1).

Table 1
Native parser daemon specifications

Daemon	Runtime	Languages	Parser
Node.js	Node 16+	TS, JS, JSX, TSX	TypeScript API
Rust	Native	Rust, C, C++	tree-sitter, libclang
JVM	JRE 17+	Java	JavaParser

B. Wire Protocol

All daemons share a common length-prefixed msgpack protocol (Fig. 2). Each frame consists of a 4-byte big-endian length prefix followed by the msgpack payload.

Request: {"action": str, "source": bytes, "file_path": str}

Response: {"ok": bool, "entities": list, "error": str|null}

Actions: ping (health check), parse (extract entities), shutdown (graceful stop)

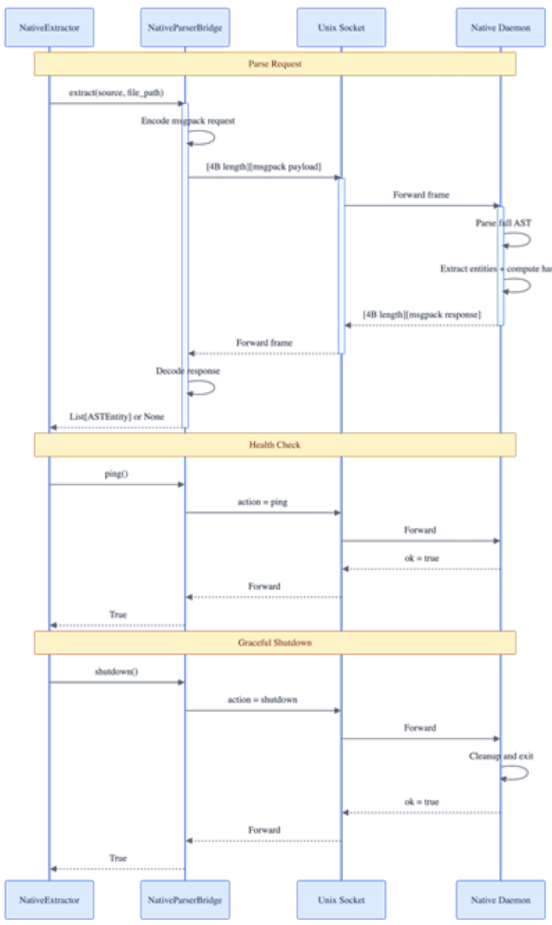

Fig. 2. Native parser wire protocol. The NativeExtractor delegates to NativeParserBridge, communicating via 4-byte length-prefixed msgpack frames over Unix domain sockets. On daemon failure, the fallback extractor is used transparently

Each entity in the response conforms to the ASTEntity schema: entity_type, name, line_start, line_end, signature_hash, structure_hash, calls, inherits, imports, type_refs, decorators, and params.

C. Daemon Lifecycle Management

The ParserDaemonManager handles the full daemon lifecycle (Fig. 3):

- Start: Launches subprocess, writes PID file, waits up to 10s for socket creation.
- Health check: ping action via NativeParserBridge; auto-restart on failure.
- Stop: Graceful shutdown → SIGTERM → SIGKILL (5s timeout), socket/PID cleanup.
- Orphan cleanup: Stale PID files are detected and orphaned daemons killed on startup.

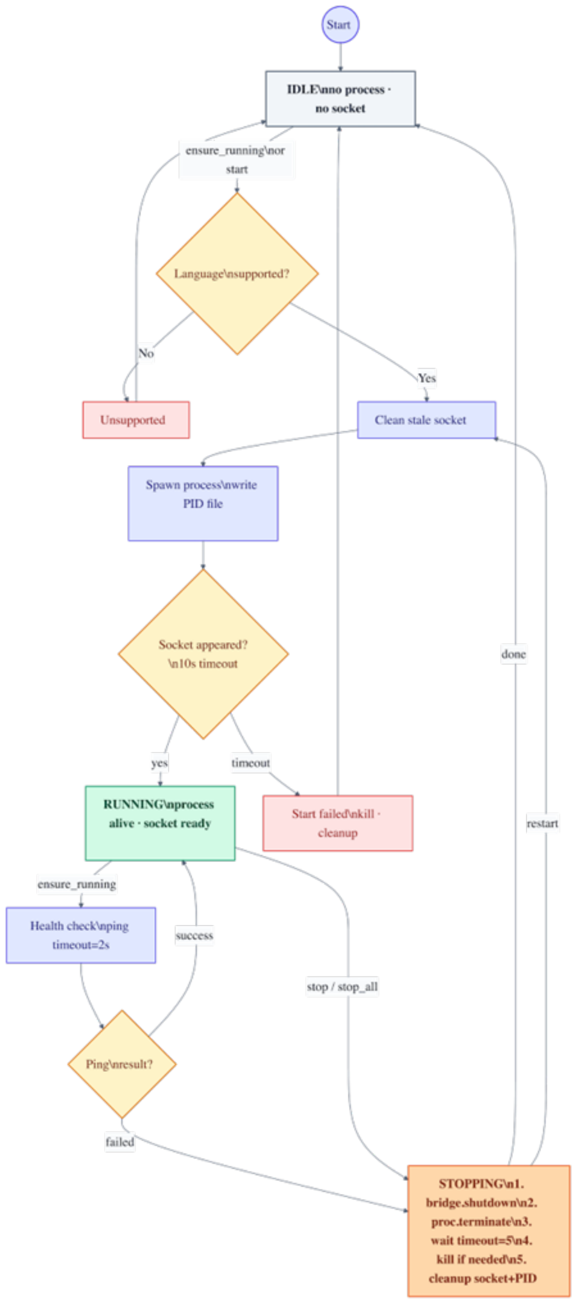
Start
IDLE\nno process · no socket
ensure_running\nor start
Language\nsupported?
No
Yes
Unsupported
Clean stale socket
Spawn process\nwrite PID file
Socket appeared?\n10s timeout
yes
timeout
RUNNING\nprocess alive · socket ready
Start failed\nkill · cleanup
done
restart
ensure_running
Health check\nping timeout=2s
success
stop / stop_all
Ping\nresult?
failed
STOPPING\n1. bridge.shutdown\n2. proc.terminate\n3. wait timeout=5\n4. kill if needed\n5. cleanup socket+PID

Fig. 3. Daemon lifecycle state machine. The ParserDaemonManager transitions each daemon through IDLE → spawn → RUNNING states, with health-check-driven restart and graceful multi-stage shutdown

D. *NativeExtractor Wrapper*

The NativeExtractor bridges the async NativeParserBridge to the synchronous LanguageExtractor ABC, running async calls in a shared ThreadPoolExecutor (4 workers):

Extract_native(s, f) = bridge.parse(s, f) if daemon alive, fallback(s, f) otherwise

This transparent fallback ensures that code intelligence never degrades—if a native daemon crashes, tree-sitter or regex extraction seamlessly takes over.

7. Three-Tier Extraction Hierarchy

A. *Tier Architecture*

StreamRAG v2 organizes language extractors into three tiers of decreasing fidelity (Table 2).

Table 2
Three-Tier Language Support Matrix

Language	Tier 1: Native	Tier 2: Tree-sitter	Tier 3: Regex
Python	—	ast (stdlib)	—
React/TSX	typescript.sock	tree-sitter-ts	react.py
TypeScript	typescript.sock	—	typescript.py
JavaScript	typescript.sock	—	javascript.py
Rust	rust.sock	tree-sitter-rust	rust.py
C++	rust.sock	—	cpp.py
C	rust.sock	—	c.py
Java	java.sock	—	java.py

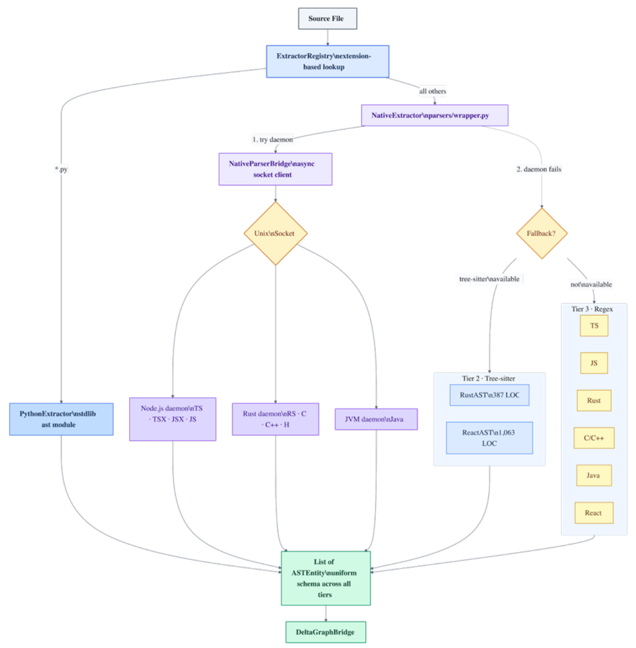
Source File
ExtractorRegistry\nextension-based lookup
all others
NativeExtractor\nparsers/wrapper.py
1. try daemon
*.py
NativeParserBridge\nasync socket client
2. daemon fails
Unix\nSocket
Fallback?
tree-sitter\navailable
not\navailable
Tier 3 · Regex
TS
JS
Rust
C/C++
Java
React
Tier 2 · Tree-sitter
RustAST\n387 LOC
ReactAST\n1,063 LOC
PythonExtractor\nstdlib ast module
Node.js daemon\nTS · TSX · JSX · JS
Rust daemon\nRS · C · C++ · H
JVM daemon\nJava
List of ASTEntity\nuniform schema across all tiers
DeltaGraphBridge

Fig. 4. Three-tier extraction hierarchy. Python uses the stdlib ast module directly. All other languages route through NativeExtractor, which attempts the native parser daemon first, falls back to tree-sitter AST, and finally to regex extraction. All tiers produce the same ASTEntity schema

B. *Extraction Capabilities Comparison*

Table 3 compares extraction fidelity across tiers.

Tier 1 (Native Daemon): Full AST parsing in the language's native runtime. Extracts generics, macros, lifetime parameters, JSX components, and scoped entities that regex cannot handle.

Tier 2 (Tree-Sitter AST): Python-hosted tree-sitter grammars providing correct AST walking without a separate process. Available for Rust (387 LOC) and React/TSX (1,063 LOC).

Tier 3 (Regex Fallback): Pattern-based extraction via the shared RegexExtractor base class. Provides comment/string stripping, brace-counting body boundaries, and scoped name resolution.

Table 3
Extraction Capabilities by Tier

Feature	Native	Tree-sitter	Regex
Functions / Methods	✓	✓	✓
Classes / Structs	✓	✓	✓
Imports	✓	✓	✓
Inheritance / Traits	✓	✓	✓
Call extraction	✓	✓	✓
Type references	✓	✓	partial
Generics / Templates	✓	✓	—
Macros (Rust)	✓	✓	—
JSX components	✓	✓	heuristic
Scoped names	✓	✓	brace-count
Broken code	fails	partial	resilient

8. V2 Subsystem

StreamRAG v2 introduces twelve subsystem modules that enhance the daemon's real-time behavior and analysis accuracy.

A. *Adaptive Debouncing*

During rapid typing, not every keystroke warrants a full graph update. The AdaptiveDebouncer classifies each keystroke into one of four tiers:

- BUFFER_ONLY: Accumulate without action (fast consecutive characters).
- TOKEN: Structural character typed (:()[]{}=,;).
- STATEMENT: Complete statement detected (closing bracket at depth 0).
- SEMANTIC: Long pause (>500ms) or explicit save.

Graph updates are triggered only at STATEMENT tier or above. The debouncer tracks bracket depth and string state to distinguish structural from non-structural keystrokes.

B. *Context Stabilization*

When an AI assistant requests context during typing, rapid graph changes can cause suggestion flickering. The AdaptiveContextStabilizer addresses this with a two-component strategy:

StableContext (cached within a 300ms window): file path, enclosing function/class, imports, available symbols. Rebuilt only after the stability window expires.

VolatileContext (updated each call): current line, column, token, and a heuristic confidence score (builtins → 1.0; CamelCase → 0.85; single characters → 0.2).

C. *Fine-Grained Invertible Operations*

V2 introduces 8 atomic operation types (AddNode, RemoveNode, UpdateNode, RenameNode, MoveNode, AddEdge, RemoveEdge, RetargetEdge), each implementing an inverse() method:

$$\forall\ op: op.apply(G) = G' \Rightarrow op.inverse().apply(G') = G$$

An OperationBatch provides atomic transactions: all operations apply or the batch rolls back via inverse operations.

D. Semantic Path Addressing

The SemanticPath provides fully-qualified entity addressing for scope-aware lookup:

$$FQN(e) = file :: scope\_chain :: \tau :: name$$

A ScopeAwareExtractor tracks scope depth during extraction, producing paths like api/auth.py::UserService::get_user for nested entities. LEGB-like name resolution traverses scope chains upward.

E. Native Parser Caching

The NativeParserBridge maintains a SHA256-keyed LRU cache:

$$key(f) = SHA256(source \mid\mid file\_path)[:16]$$

The cache stores up to 500 entries with LRU eviction. This eliminates redundant re-parses during rapid file switching—common in AI-assisted workflows.

F. Semantic Similarity Scoring

V2 introduces the EntitySimilarityScorer, combining five weighted signals for cross-file edge ranking:

$S(e\_s, e\_t) = \Sigma\ w\_i \cdot f\_i(e\_s, e\_t)$

- Import affinity (w=0.4): Direct imports score 1.0, transitive 0.5, reverse 0.8.
- Co-reference frequency (w=0.25): Shared entity references, normalized by 10.
- Graph distance (w=0.2): 1/(1+d) via BFS at file level, capped at depth 5.
- Name similarity (w=0.1): Jaccard similarity on character trigrams.
- Path similarity (w=0.05): Shared directory prefix ratio.

G. LLM-Guided Edge Resolution

Some cross-file edges remain ambiguous even after similarity scoring. V2 introduces an ambiguous edge queue (capped at 100 entries) that records unresolved edges with their candidate lists. A heuristic best-guess is applied immediately and the edge is queued for LLM refinement. This two-phase approach ensures the graph is always usable while allowing progressive refinement.

H. File Watcher

The FileWatcher detects external file changes via mtime-based polling at 2-second intervals with exponential backoff on errors. It caps at 20 changes per cycle (sorted by mtime descending) and includes a 500ms debounce flush for rapid successive edits.

I. Convention Resolvers

Cross-language projects require resolving edges that span language boundaries via naming conventions. The ConventionResolverRegistry chains specialized resolvers:

- TauriCommandResolver: Resolves TypeScript invoke('get_users') calls to Rust #[command] fn get_users with automatic camelCase-to-snake_case conversion.
- BarrelReExportResolver: Resolves imports from barrel files (index.ts, __init__.py, mod.rs) to actual definitions in sub-modules.

J. Accuracy Enhancements

Confidence-Filtered Cycle Detection: find_cycles() accepts min_confidence and import_backed_only parameters. Strict mode filters name-coincidence cycles lacking actual import edges.

Structured Impact Separation:

get_affected_files_structured() returns {"direct": [...], "transitive": [...]} with separate collection phases.

Workspace-Aware Dead Code Analysis: Scoped to monorepo workspace boundaries, detected via manifest files. Framework entry decorators (20+ patterns) are excluded.

9. Daemon Architecture

A. V1 Daemon (Unchanged)

The core daemon remains an asyncio server communicating via Unix domain socket with newline-delimited JSON. RPC methods: ping, process_change, get_read_context, classify_query, shutdown.

B. V2 Daemon Enhancements

The v2 daemon acts as a stateful orchestrator managing twelve subsystems:

1) Core DeltaGraphBridge (v1)
2) ParserDaemonManager — lifecycle for 3 native daemons
3) AdaptiveDebouncer — per-file debounce buffers
4) AdaptiveContextStabilizer — context caching
5) AISessionManager — multi-session conflict tracking
6) Async propagation task with bounded depth-limited BFS
7) NativeParserBridge cache — SHA256/LRU parse cache
8) EntitySimilarityScorer — 5-signal edge ranking
9) Ambiguous edge queue — LLM-guided resolution
10) FileWatcher — mtime-based polling (2s interval)
11) ConventionResolverRegistry — cross-language edges
12) SmartQuery LLM fallback — Claude API translation

All v2 features are opt-in via conditional imports: if v2 modules are unavailable, the daemon operates in v1 mode with identical behavior.

C. Hook Integration

StreamRAG integrates with Claude Code through four transparent hooks (Table 4).

Table 4
Hook Integration Points

Hook	Trigger	Action
PostToolUse	Edit/Write	Incremental graph update
PreToolUse	Read	Inject entity context
PreToolUse	Task/Grep	Redirect to graph
Stop	Session end	Serialize summary

Context Injection: When the AI reads a file, StreamRAG injects a budget-aware context block allocating 40% for callers, 25% for dependencies, 25% for affected files, and 10% for the header.

10. Experimental Evaluation

A. Experimental Setup

- Hardware: Apple M-series, 16GB RAM
- Python: 3.12 (CPython)
- Test Suite: 1,020 tests across 62 test files
- Codebase: ~20,000 lines across Python (13,381), TypeScript (1,100), Rust (4,923), Java (859)
- Native daemons: Node.js 16+, Rust (release build), JDK 17+

B. Micro-Benchmark: Incremental Processing

Table 5 presents per-operation timing for core pipeline stages, measured over 1000 iterations.

Table 5
Micro-Benchmark: Per-Operation Latency

Operation	Time	Complexity
Semantic gate (no change)	<0.01ms	O(n) parse
Single function body edit	~0.05ms	O(Δ)
Add new function	~0.03ms	O(|E|) edges
Rename detection	~0.04ms	O(|R|·|A|)
Cross-file edge resolution	~0.08ms	O(|V|) scan
Native parse (TS, 100 LOC)	~0.8ms	IPC + AST
Native parse (Rust, 500 LOC)	~1.2ms	IPC + AST
Native parse (Java, 200 LOC)	~1.0ms	IPC + AST
Debouncer classification	<0.001ms	9 rules
Context stabilizer lookup	<0.001ms	dict lookup
Full pipeline (typical)	~0.05ms	O(Δ+|E_new|)

Key Result: The incremental pipeline latency remains 0.05ms (200× under target). Native parsing adds ~1ms IPC overhead but runs concurrently with the graph update.

C. Comparison: StreamRAG vs Batch GraphRAG

Table 6
Speedup Analysis: StreamRAG vs Batch GraphRAG

Scenario	StreamRAG	Batch	Speedup
Large file (1000fn)	125.8ms	760.7ms	6.0×
Many files (100)	22.7ms	627.6ms	27.6×
Rapid keystrokes	1.4ms	5.8ms	4.0×
Mixed workload	7.9ms	104.2ms	13.2×
File switching	74.5ms	369.2ms	5.0×
Realistic session	17.5ms	474.4ms	27.1×
Average			11.5×

D. Real-World Benchmark: Claude CLI

We evaluated StreamRAG on 8 real-world code understanding tasks using the Claude CLI against a production API codebase. Each task was run with three configurations: no plugin (baseline), StreamRAG v1, and StreamRAG v2.

Table 6
Real-World benchmark: 8 Code understanding tasks

Metric	No Plugin	v1	v2
Avg Accuracy	95%	72%	92%
Total Time	567s	1490s	649s
Total Tokens	2.76M	3.10M	1.47M
Total Turns	85	105	69
Total Cost	\$3.46	\$3.93	\$4.00
Tests at 100%	6/8	4/8	5/8

V2 vs V1: +20pp accuracy (92% vs 72%), 2.3× faster, 53% fewer tokens, 34% fewer turns.

V2 vs No Plugin: 47% fewer tokens and 19% fewer turns, at a 3pp accuracy cost. Token savings are most dramatic on complex architecture tracing tasks.

E. Per-Category Analysis

Table 7
Per-Category Accuracy and Efficiency (v2 vs Baseline)

Category	Acc (NP)	Acc (v2)	Tok (NP)	Tok (v2)
Dependency Analysis	100%	100%	109K	143K
Impact Analysis	90%	68%	75K	132K
Arch. Tracing	92%	92%	1,118K	366K
Cross-Module	100%	92%	79K	95K

Standout Result: On the most complex test (integration registry trace), StreamRAG v2 answered in 2 turns with 65K tokens at 100% accuracy—compared to 28 turns and 1.47M tokens (96% reduction) without the plugin.

F. Synthetic Engine Benchmark

Table 8
Engine Benchmark: Incremental vs Naive Full-Reparse

Scenario	StreamRAG	Naive	Speedup
Cold start (full project)	44.2ms	58.2ms	1.3×
Function body edit	6.1ms	3.5ms	0.6×
Whitespace change	0.8ms	3.4ms	4.4×
Function rename	1.5ms	3.5ms	2.3×
Add new file	1.4ms	3.8ms	2.7×
Keystroke storm (50 edits)	155.9ms	189.4ms	1.2×
Total	209.8ms	261.6ms	1.2×

G. Test Coverage

The distribution of 1,020 tests across 14 functional areas: Language extractors (210), V2 Subsystem (122), Bridge (100), Graph (86), Native Parsers (80), Integration (62), Daemon Lifecycle (57), Accuracy (53), Storage (41), Models (35), Hooks (30), Smart Query (25). V2 added 423 tests (+71% from v1's 597).

11. Case Study: Architecture Tracing

The real-world benchmark reveals StreamRAG's most compelling use case: complex architecture tracing. We examine the integration registry trace task, where the AI must trace from api/integrations/registry.py through discovery to when an integration's functions are used in a chat request.

Without StreamRAG (baseline): The assistant required 28 turns and consumed 1.47M tokens (\$1.34) to explore the codebase file by file, reading 11 files to reconstruct the dependency chain. It achieved 100% accuracy.

With StreamRAG v2: The assistant answered in 2 turns with 65K tokens (\$1.81) at the same 100% accuracy, referencing 18 files. The code graph provided the cross-module dependency map directly, enabling a comprehensive trace without manual file exploration.

A similar pattern emerges in the dependency caller task. StreamRAG v1 timed out after 600 seconds; v2 answered correctly in 25 seconds at 100% accuracy.

Table 9
Benchmark Comparison: Integration Registry Trace

Metric	No Plugin	StreamRAG v2
Accuracy	100%	100%
Turns	28	2 (93% reduction)
Tokens	1.47M	65K (96% reduction)
Files cited	11	18 (more comprehensive)

12. Data Structure Summary

Table 10
Core Data Structures († = v2 additions)

Structure	Type	Purpose
ASTEntity	dataclass	Extracted entity
GraphNode	dataclass	Node with ID, type, file
GraphEdge	dataclass	Typed edge + confidence
LiquidGraph	class	5-indexed graph engine
DeltaGraphBridge	class	10-stage pipeline
VersionedGraph	class	Version tracking
† NativeParserBridge	class	Async socket client
† NativeExtractor	class	Tier fallback wrapper
† ParserDaemonManager	class	Daemon lifecycle
† AdaptiveDebouncer	class	Keystroke classifier
† ContextStabilizer	class	Flicker prevention
† SemanticPath	dataclass	FQN addressing
† GraphOp (8 types)	ABC	Invertible graph ops
† SimilarityScorer	class	5-signal edge ranking
† FileWatcher	class	Mtime change detection
† ConventionResolver	ABC	Cross-language edges
† WorkspaceDetector	class	Monorepo boundaries

13. Discussion

A. Design Decisions

Separate daemon processes rather than in-process FFI were chosen because: (a) each parser runs in its native optimized runtime; (b) crashes are isolated—a segfault in the Rust parser does not bring down the Python daemon; (c) daemons can be upgraded independently.

Msgpack over JSON for the parser protocol because binary msgpack is 2–5× smaller for source code payloads and avoids UTF-8 encoding overhead for binary source bytes.

Three-tier fallback rather than hard dependency on native parsers ensures zero-configuration operation—the system works out of the box with regex and progressively improves as native daemons and tree-sitter are installed.

Convention resolvers address cross-language edges that cannot be resolved by import graphs alone. Rather than attempting general cross-language type inference, we target specific framework conventions (Tauri commands, barrel re-exports) that account for the majority of cross-language edges in practice.

B. Limitations

1) Native daemon startup cost: Each daemon adds 5–10ms startup latency on first use.
2) Socket IPC overhead: ~0.5–1ms per parse request for msgpack encoding + Unix socket round-trip.
3) Cold start cost: Initial project scan (up to 200 files, ≤7s) is a one-time cost.
4) Dynamic dispatch: Calls via dynamic dispatch (getattr, computed method names) are not tracked.
5) Partial v2 integration: SemanticPath is wired for Python files; edge operations and non-Python SemanticPath integration are remaining gaps.

C. Filtering False Edges

StreamRAG maintains three curated filter sets: BUILTINS (84 names), COMMON_ATTR_METHODS (97 names), and STDLIB_MODULES + KNOWN_EXTERNAL_PACKAGES (170+ names). These prevent spurious cross-file edges from common method names and built-in references.

14. Conclusion

We presented StreamRAG v2, extending the v1 incremental code graph system with five major contributions:

1) Native Parser Daemons: Three persistent processes (Node.js, Rust, JVM) providing full AST parsing over msgpack Unix sockets, with automatic lifecycle management and SHA256-keyed LRU caching.
2) Three-Tier Extraction: A graceful degradation hierarchy (native → tree-sitter → regex) ensuring optimal extraction fidelity with zero-configuration fallback.
3) V2 Subsystem: Adaptive debouncing, context stabilization, fine-grained invertible operations, semantic path addressing, time-based file watching, and multi-session daemon support.
4) Intelligence Subsystems: Five-signal semantic similarity scoring and LLM-guided resolution for progressive graph refinement.
5) Accuracy Subsystems: Cross-language convention resolvers, confidence-filtered cycle detection, structured impact separation, and workspace-aware dead code analysis.

StreamRAG v2 maintains the v1 performance profile (0.05ms per incremental change, 11.5× average speedup over batch GraphRAG) while dramatically improving extraction quality for non-Python languages. Real-world benchmarks demonstrate 47% token reduction and 19% fewer turns compared to unaugmented Claude, with a 96% token reduction on complex architecture tracing tasks. Compared to v1, v2 achieves +20pp accuracy (92% vs 72%), 2.3× faster execution, and 53% fewer tokens. With 1,020 passing tests across 62 test files and 20,000+ lines of production code spanning 4 implementation languages, it represents a comprehensive polyglot code intelligence platform deployed as a production Claude Code plugin.

Acknowledgments

We thank the Claude Code team at Anthropic for the plugin architecture that made this integration possible.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. G. Neumann, L. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, 2020.

[2] L. Zhang, *et al.*, "CodeGraph: Semantic code search via graph neural networks," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2023.

[3] Y. Wang, *et al.*, "GRACE: Graph-based retrieval augmented code embeddings," in *Proc. Annu. Meeting Assoc. Comput. Linguist. (ACL)*, 2024.

[4] X. Chen, *et al.*, "RepoMap: Repository-level code understanding," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2024.

[5] M. Brunsfeld, "Tree-sitter: A new parsing system for programming tools," GitHub, 2018. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[6] Microsoft, "Language Server Protocol," 2016. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[7] K. Choudhary, "StreamRAG: Real-time incremental code graph synchronization for AI-driven code editors," 2025.

[8] GitHub, "GitHub Copilot: AI pair programmer," GitHub, 2023. [Online]. Available: https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/

[9] Cursor Team, "Cursor: The AI-first code editor," 2024. [Online]. Available: https://cursor.sh

[10] Microsoft Research, "GraphRAG: Graph-based retrieval augmented generation," Microsoft Research, 2024. [Online]. Available: https://www.microsoft.com/en-us/research/project/graphrag/

[11] S. Furuhashi, "MessagePack: It's like JSON but fast and small," 2013. [Online]. Available: https://msgpack.org

[12] JavaParser Contributors, "JavaParser: Java 1–17 parser and AST," 2024. [Online]. Available: https://javaparser.org