StreamRAG: Real-Time Incremental Code Graph Synchronization for AI-Driven Code Editors

Krrish Choudhary
Independent Research
krrishchoudhary109@gmail.com

***Abstract*—We present StreamRAG, a real-time incremental code graph system for AI-driven code editors. Unlike traditional code intelligence tools that perform expensive full-project scans, StreamRAG treats code as a *fluid, evolving structure*—maintaining a live dependency graph that updates surgically on every edit. Our system introduces six key innovations: (1) the LiquidGraph, an in-memory graph with five concurrent indexes for $O(1)$ lookups; (2) the DeltaGraphBridge, a 10-stage incremental pipeline with semantic gating and two-pass edge resolution; (3) ShadowAST, a binary-search partial parser for syntactically broken code; (4) Hierarchical Zone Management with HOT/WARM/COLD file prioritization; (5) Bounded Propagation with priority-queue-based cascade prevention; and (6) a multi-language extraction framework supporting 7 languages through a shared regex infrastructure. StreamRAG achieves sub-0.05ms per incremental change—200× under our 10ms latency target—while tracking calls, imports, inheritance, type references, and decorator relationships across files. Deployed as a zero-dependency Claude Code plugin, StreamRAG processes 597 test scenarios with 100% pass rate, supports 16 graph query commands including natural language, and persists graph state across sessions via a Unix-domain-socket daemon architecture.**

***Index Terms*—Code Understanding, Knowledge Graphs, Incremental Parsing, Real-Time Systems, AI Code Assistants, Dependency Analysis, Abstract Syntax Trees**

I. INTRODUCTION

AI-powered code editors—GitHub Copilot [7], Cursor [6], 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).

Graph-based Retrieval-Augmented Generation (GraphRAG) [8] extends RAG [1] by constructing knowledge graphs that capture semantic relationships. However, existing implementations treat the graph as a static artifact, rebuilding it in batch mode—an approach fundamentally incompatible with the keystroke-by-keystroke nature of code editing.

A. The Real-Time Challenge

Consider the workflow of an AI code assistant: when a developer edits a file, the assistant must instantly know (a) what entities changed, (b) what depends on those entities across the project, and (c) whether the change introduces breaking dependencies. Current tools resort to repeated `grep` operations, which suffer from false positives (matching strings/comments), miss transitive dependencies, and burn the LLM's context window on exploration.

B. Contributions

This paper makes the following contributions:

- **LiquidGraph**: An in-memory graph with 5 concurrent indexes (primary, file, type, name, edge) enabling $O(1)$ node lookups and $O(k)$ file-scoped queries.
- **DeltaGraphBridge**: A 10-stage incremental pipeline achieving $O(\Delta)$ update complexity where Δ is the semantic change size, with position-based rename detection and two-pass edge resolution.
- **ShadowAST**: A binary-search partial parser that recovers entities from syntactically broken code with confidence scoring.
- **Multi-Language Framework**: A pluggable extractor architecture supporting 7 languages (Python, TypeScript, JavaScript, Rust, C++, C, Java) through a shared regex infrastructure.
- **Daemon Architecture**: A persistent asyncio server with Unix domain socket IPC, eliminating per-hook process spawn overhead.
- **Comprehensive Evaluation**: 597 passing tests, sub-0.05ms incremental updates, and deployment as a production Claude Code plugin.

II. BACKGROUND AND RELATED WORK

A. Retrieval-Augmented Generation

RAG [1] augments language model generation with retrieved context:

$$P(y|x) = \sum_{z \in \mathcal{Z}} P(z|x) \cdot P(y|x, z) \tag{1}$$

where x is the input, z are retrieved documents, and y is the generated output.

B. Graph-Based Code Understanding

Prior work on code graphs includes CodeGraph [3] for static call graphs, GRACE [4] for graph-based embeddings, and RepoMap [5] for repository-level mapping. All treat graph construction as offline batch processing, requiring full rebuilds on each change.

C. Incremental Parsing

Tree-sitter [2] pioneered incremental parsing for syntax highlighting but focuses on single-file syntax trees, not cross-file semantic graphs. The Language Server Protocol [9] provides incremental diagnostics but lacks the graph abstraction needed for dependency analysis. StreamRAG extends incremental parsing to *semantic* graph updates with cross-file edge resolution.

III. PROBLEM FORMALIZATION

Definition 1: Code Knowledge Graph

A code knowledge graph is a labeled, directed multigraph $G = (V, E, \phi, \psi)$ where:

- V: code entity nodes (functions, classes, imports, variables)
- $E \subseteq V \times V \times \mathcal{R}$: typed directed edges
- $\phi : V \to \mathcal{T} = \{\texttt{function}, \texttt{class}, \texttt{import}, \texttt{variable}, \texttt{module\_code}\}$
- $\psi : E \to \mathcal{R} = \{\texttt{calls}, \texttt{imports}, \texttt{inherits}, \texttt{uses\_type}, \texttt{decorated\_by}\}$

Definition 2: Incremental Graph Synchronization

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

$$G_i = G_{i-1} \oplus \Delta G_i \tag{2}$$

minimizing the latency-staleness objective:

$$\mathcal{L} = \sum_{i=1}^{n} (\alpha \cdot \text{Latency}(c_i) + \beta \cdot |\Delta V_{\text{missed}}|) \tag{3}$$

where ΔV_{missed} represents entities that changed but were not captured.

Definition 3: Semantic Gate

A change $c_i = (f, s_{\text{old}}, s_{\text{new}})$ is *semantically significant* iff:

$$\{(e.\text{name}, h(e)) \mid e \in \text{Extract}(s_{\text{old}})\} \neq \{(e.\text{name}, h(e)) \mid e \in \text{Extract}(s_{\text{new}})\} \tag{4}$$

where $h(e)$ is the signature hash of entity e including its body content.

IV. STREAMRAG ARCHITECTURE

Fig. 1 presents the StreamRAG architecture, organized into four interconnected subsystems: the Graph Engine, the Incremental Pipeline, the Language Framework, and the Integration Layer.

V. CORE COMPONENTS

A. LiquidGraph: In-Memory Graph Engine

The LiquidGraph maintains five concurrent indexes for fast lookups:

$$\mathcal{G} = \langle \mathcal{N}, \mathcal{I}_f, \mathcal{I}_t, \mathcal{I}_n, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}} \rangle \tag{5}$$

where $\mathcal{N}$ maps node IDs to `GraphNode` objects, $\mathcal{I}_f$ maps file paths to node ID sets, $\mathcal{I}_t$ maps entity types to node ID sets, $\mathcal{I}_n$ maps entity names to node ID sets, and $\mathcal{E}_{\text{out}}/\mathcal{E}_{\text{in}}$ are adjacency lists for outgoing/incoming edges.

Node Identity. Each node receives a deterministic ID:

$$\text{id}(n) = \text{SHA256}(f \| \text{":"} \| \tau \| \text{":"} \| \text{name})_{[0:16]} \tag{6}$$

where f is the file path, τ is the entity type, and name is the scoped entity name.

Query by Intersection. Multi-criteria queries intersect index sets:

$$\text{Query}(f, \tau, n) = \mathcal{I}_f[f] \cap \mathcal{I}_t[\tau] \cap \mathcal{I}_n[n] \tag{7}$$

achieving $O(k)$ where $k = \min(|\mathcal{I}_f[f]|, |\mathcal{I}_t[\tau]|, |\mathcal{I}_n[n]|)$.

Cascade Removal. When a node is removed, all incident edges are automatically cleaned from both adjacency indexes, maintaining referential integrity.

Fig. 2 illustrates the five edge types tracked by LiquidGraph.

B. Graph Traversal and Analysis

LiquidGraph supports several analysis algorithms:

BFS Traversal. Parameterized by edge types, direction (`outgoing`/`incoming`/`both`), and maximum depth:

$$\text{Traverse}(v_0, \mathcal{R}', d, D) = \{(v, d) \mid v \in \text{BFS}(v_0, \mathcal{R}'), d \leq D\} \tag{8}$$

Dead Code Detection. Identifies unreferenced entities while respecting entry points, framework patterns (`test_*`, `visit_*`), polymorphic overrides, and `@property` decorators:

$$\text{Dead}(G) = \{v \in V \mid |\mathcal{E}_{\text{in}}(v)| = 0 \wedge v \notin \text{EntryPts} \wedge \neg\text{Override}(v)\} \tag{9}$$

Cycle Detection. Uses iterative DFS with WHITE/GRAY/BLACK coloring at the file level, producing minimal normalized cycles.

Shortest Path. BFS-based path finding with edge-type filtering and parent-pointer reconstruction.

C. DeltaGraphBridge: The Incremental Pipeline

The DeltaGraphBridge is the core orchestrator. Algorithm V-C describes its 10-stage pipeline for processing a single code change.

1) Semantic Gate (Stage 1)

The semantic gate eliminates non-semantic changes (whitespace, comments, formatting) by comparing entity signature sets:

$$\text{Sem}(s, s') = \left(\{(e.\text{name}, h(e))\}_{e \in E(s)} \neq \{(e.\text{name}, h(e))\}_{e \in E(s')}\right) \tag{10}$$

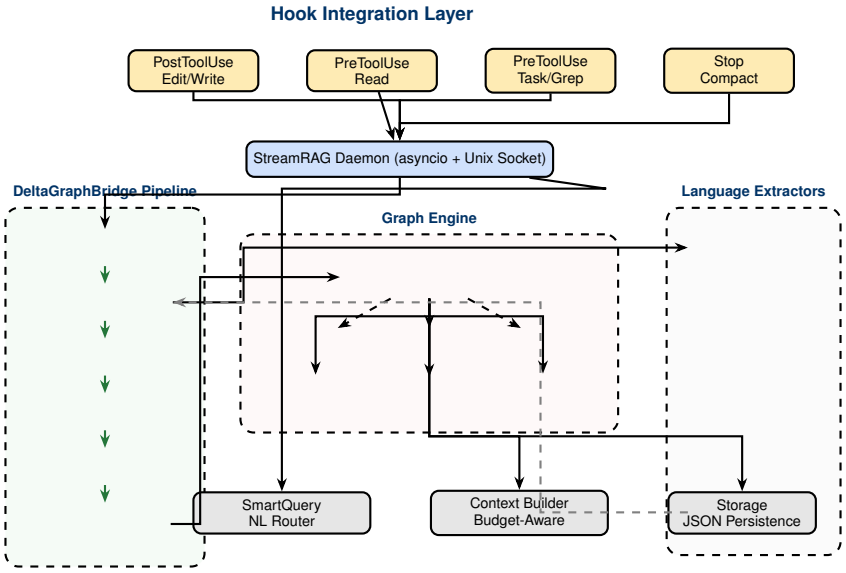

Fig. 1. StreamRAG system architecture. Code changes flow through 4 hooks into the persistent daemon, which routes them through the DeltaGraphBridge pipeline. The pipeline extracts entities via language-specific extractors, computes minimal diffs, and surgically patches the LiquidGraph. Query results flow back through the context builder to the AI assistant.

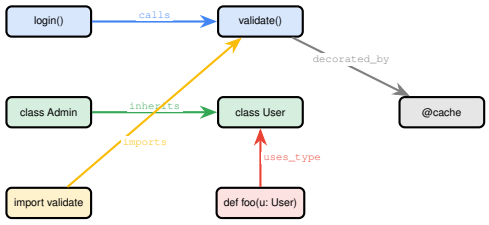

Fig. 2. Five edge types in the StreamRAG graph model. Each edge carries confidence metadata (high/medium/low) based on resolution certainty.

Crucially, if the new content fails to parse (SyntaxError) but the old content was valid, the gate returns `false`—preventing ghost nodes from broken mid-edit code.

2) Delta Computation (Stage 2)

Delta computation produces three disjoint sets: $(\mathcal{A}, \mathcal{R}, \mathcal{M})$ for added, removed, and modified entities.

Rename Detection. An entity $e_{\text{old}} \in \mathcal{R}$ is matched to $e_{\text{new}} \in \mathcal{A}$ as a rename if:

$$\text{Ren}(e_o, e_n) = \begin{cases} 1 & \phi(e_o) = \phi(e_n) \wedge \text{ovlp}(e_o, e_n) \\ & \wedge\ h_s(e_o) = h_s(e_n) \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

where ϕ is the entity type, overlap checks positional overlap, and h_s is the *structure hash*—computed *without* the entity name, enabling detection of name changes while the structure stays the same.

3) Two-Pass Edge Resolution (Stage 6)

Cross-file edges require two resolution passes because of forward references:

Pass 1 (within `add_node`): Creates edges using only currently-known nodes. Resolves calls, inheritance, imports, type references, and decorators.

Pass 2 (after all additions): Re-resolves all edges for added and modified entities, now with full visibility of newly-created nodes.

4) Target Node Resolution

The `_find_target_node` algorithm uses an 8-level priority cascade:

1) **Qualified name**: Resolve `receiver.method` via class lookup or import-based file resolution
2) **Exact match from imported file**: Highest confidence ("high")
3) **Exact match cross-file**: Medium confidence, path-similarity scoring
4) **Exact match same-file**: Medium confidence
5) **Suffix match (.name) from imported file**: Medium confidence
6) **Suffix match cross-file**: Low confidence
7) **Inheritance chain walk**: BFS up parent classes (max 5 levels)
8) **Fallback**: Index-based suffix search with path-similarity tiebreaking

Fig. 3 illustrates the resolution priority cascade.

D. ShadowAST: Partial Parsing for Broken Code

During active typing, code is frequently syntactically invalid. ShadowAST recovers partial entities using a three-phase strategy (Fig. 4):

1) **Full Parse Attempt**: If successful, return single VALID region with all entities.
2) **Binary Search**: Recursively split the source at midpoint; try parsing each half. This identifies maximal valid regions in $O(n \log n)$ where n is line count.
3) **Regex Fallback**: For INVALID regions, apply regex patterns to recover function signatures (`def`/`async def`),

Algorithm 1: DeltaGraphBridge.process_change

Require: CodeChange $c = (f, s_{\text{old}}, s_{\text{new}})$
Ensure: List of GraphOperations

// **Stage 1: Semantic Gate**
if $\neg$ is_semantic_change$(s_{\text{old}}, s_{\text{new}}, f)$ **then**
  update_cache(f, s_{new});
  return $\emptyset$
end if

// **Stage 2: Compute Delta**
$(\mathcal{A}, \mathcal{R}, \mathcal{M}) \leftarrow$ compute_delta$(f, s_{\text{old}}, s_{\text{new}})$
ops $\leftarrow$ []
// **Stage 3: Process Removals (first!)**
for $e \in \mathcal{R}$ **do**
  capture_callers(e); remove_node(e)
  ops.append(remove_node_op)
end for

// **Stage 4: Process Additions (imports first)**
sort $\mathcal{A}$ by (import $\prec$ non-import, name)
for $e \in \mathcal{A}$ **do**
  add_node(e); create_first_pass_edges(e)
  reverse_import_sweep(e)
end for
// **Stage 5: Process Modifications + Renames**
for $e \in \mathcal{M}$ **do**

  if e.old_name $\neq$ None **then**
    remove_old(e); add_new(e) {Rename}
  else
    update_in_place(e); clear_stale_edges(e)
  end if
end for
// **Stage 6: Two-Pass Edge Resolution**
for $e \in \mathcal{A} \cup \mathcal{M}$ **do**

  resolve_pending_edges(e, f)
end for
// **Stage 7–10: Caches, Versions, Propagation, Hierarchy**
update_caches(f); record_version(f)
propagate_bounded(f); update_zones(f)
return ops

class headers (`class`), and import statements with graduated confidence scores.

Confidence scores are assigned based on syntactic completeness: a function with closing parenthesis and colon receives $c = 0.9$; without colon, $c = 0.7$; incomplete parameters, $c = 0.5$.

E. Hierarchical Zone Management

Files are classified into three zones (Fig. 5):

- **HOT** (max 10): Currently open files. Synchronous updates, <10ms target.

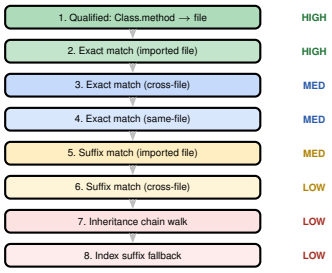

Fig. 3. 8-level target resolution priority cascade. Each level assigns a confidence score used for downstream edge quality assessment.

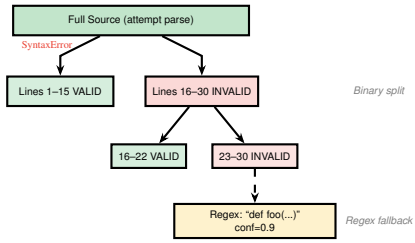

Fig. 4. ShadowAST binary-search parsing. Valid regions are extracted with full AST fidelity; invalid regions fall back to regex patterns with confidence scores.

- **WARM** (max 50): Transitive dependencies and recently accessed files. Asynchronous queue.
- **COLD**: Everything else. Lazy/on-demand processing.

Zone Transitions. File $\rightarrow$ HOT on open; dependencies $\rightarrow$ WARM when a file opens; HOT $\rightarrow$ WARM on close (not COLD, for fast reopen). HOT eviction removes the oldest non-open file.

Update Priority.

$$P(f) = 100 - 50 \cdot \mathbb{1}_{\text{open}}(f) - 30 \cdot \mathbb{1}_{t<60\text{s}}(f) + 20 \cdot \mathbb{1}_{\text{test}}(f) \quad (12)$$

F. Bounded Propagation

When a file changes, its dependents may need re-parsing. Unbounded propagation causes *dependency avalanches*. Our bounded propagator uses a three-phase approach:

Phase 1 (SYNC): Process top-priority files up to $S_{\max} = 5$ or timeout $T_{\max} = 50$ms, whichever comes first.

Phase 2 (ASYNC): Queue next $A_{\max} = 50$ files in a priority heap.

Phase 3 (DEFERRED): Everything else is deferred until explicitly requested.

Priority Formula:

$$\text{Pri}(f, d) = d \cdot w_d - w_o \cdot \mathbb{1}_{\text{open}} - w_r \cdot \mathbb{1}_{\text{recent}} + w_t \cdot \mathbb{1}_{\text{test}} + w_g \cdot \mathbb{1}_{\text{gen}} \quad (13)$$

where d is BFS depth, $w_d = 20$, $w_o = 100$, $w_r = 50$, $w_t = 30$, $w_g = 50$.

Proposition 1. *With sync limit S and async limit A, the per-change processing is bounded by $O(S + A \cdot \log A)$ regardless of total graph size.*

Fig. 5. Hierarchical zone model with transition rules. Open files are promoted to HOT; their transitive dependencies are promoted to WARM.

TABLE I
LANGUAGE SUPPORT MATRIX

Feature	Python	TS	JS	Rust	C++	C	Java
Functions	✓	✓	✓	✓	✓	✓	✓
Classes	✓	✓	✓	✓	✓	✓	✓
Imports	✓	✓	✓	✓	✓	✓	✓
Inheritance	✓	✓	✓	✓	✓		✓
Calls	✓	✓	✓	✓	✓	✓	✓
Types	✓	✓					
Decorators	✓	✓	✓	✓			✓
Method	AST	Regex	Regex	Regex	Regex	Regex	Regex

G. Multi-Language Extraction Framework

StreamRAG supports 7 languages through a two-tier extractor architecture (Table I).

Python uses full AST parsing via the `ast` module, enabling type-context-aware call extraction (e.g., `self.method()` → `ClassName.method`).

Non-Python languages share a `RegexExtractor` base class providing: comment/string stripping (preserving line numbers), brace-counting for body boundaries, scoped name resolution, and per-language builtin filtering. Each subclass supplies declaration patterns, import patterns, and language-specific builtins.

H. Version Vectors and Conflict Detection

The VersionedGraph tracks a global monotonic version counter and per-file version vector:

$$\vec{V} = \{f_1 : v_1, f_2 : v_2, \ldots, f_n : v_n\} \tag{14}$$

Conflict Detection. An AI session starting at version v_{base} detects conflicts if:

$$\exists \text{op} \in \text{Log}[v_{\text{base}}..v_{\text{current}}] : \text{op.node\_id} \in \text{Proposed} \tag{15}$$

Three conflict types are detected: DELETION (entity removed since session start), RENAME (entity renamed), and CONCURRENT_EDIT (same entity modified by both sides).

Automatic Resolution. Rename conflicts can be auto-resolved by substituting old names with new names in the proposed changes. Deletion conflicts are resolved by filtering operations targeting deleted nodes.

I. Daemon Architecture

StreamRAG runs as a persistent asyncio daemon (Fig. 6), communicating via a Unix domain socket with newline-delimited JSON:

- **Socket**: `~/.claude/streamrag/daemon_{hash}.sock`

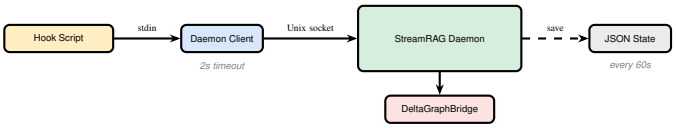

Fig. 6. Daemon architecture. Hooks communicate via Unix domain sockets, avoiding per-hook process spawn overhead.

TABLE II
HOOK INTEGRATION POINTS

Hook	Trigger	Action
PostToolUse	Edit/Write	Incremental graph update Breaking change alerts
PreToolUse	Read	Inject entity context Callers, deps, affected
PreToolUse	Task/Grep	Redirect relationship queries to graph
Stop	Session end	Serialize summary

- **PID file**: Liveness checking and stale cleanup
- **Auto-init**: Scans up to 200 files (7s timeout)
- **Periodic save**: Serialized every 60s if dirty
- **RPC**: `ping`, `process_change`, `get_read_context`, `classify_query`, `shutdown`

J. Hook Integration with AI Assistants

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

Context Injection. When the AI reads a file, StreamRAG injects a budget-aware context block:

Context Injection Example

```
[StreamRAG] bridge.py: 8fn 2cls
Called by:
  process_change <-- daemon.py:handle_process_change
  _extract <-- process_change (same file)
Deps: graph.py, extractor.py, models.py
Affected: daemon.py, on_file_change.py +1 more
```

The context builder allocates its character budget proportionally: 40% for callers, 25% for dependencies, 25% for affected files, and 10% for the header.

VI. EXPERIMENTAL EVALUATION

A. Experimental Setup

- **Hardware**: Apple M-series, 16GB RAM
- **Python**: 3.12 (CPython)
- **Test Suite**: 597 tests across 28 test files
- **Codebase**: ∼6,700 lines of production code, zero external dependencies

B. Micro-Benchmark: Incremental Processing

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

Key Result

The typical incremental update completes in ∼**0.05ms**—**200**× under our 10ms latency target, enabling real-time processing of every keystroke.

TABLE III
MICRO-BENCHMARK: PER-OPERATION LATENCY

Operation	Time	Complexity
Semantic gate (no change)	<0.01ms	$O(n)$ parse
Single function body edit	∼0.05ms	$O(\Delta)$
Add new function	∼0.03ms	$O(\|E\|)$ edges
Rename detection	∼0.04ms	$O(\|\mathcal{R}\| \cdot \|\mathcal{A}\|)$
Cross-file edge resolution	∼0.08ms	$O(\|V\|)$ scan
Dead code detection	∼0.5ms	$O(\|V\| + \|E\|)$
Cycle detection (DFS)	∼1.0ms	$O(\|V\| + \|E\|)$
Graph snapshot (deep copy)	∼2.0ms	$O(\|V\| + \|E\|)$
Full pipeline (typical)	**∼0.05ms**	$O(\Delta + \|E_{\text{new}}\|)$

TABLE IV
STRESS TEST SUITE

Test Scenario	Time	Status
Large file (1000 functions)	379ms	PASS
Many files (100 concurrent)	27ms	PASS
Rapid keystrokes (50 edits)	0.3ms	PASS
Deep deps (10 levels)	0.8ms	PASS
Wide deps (100 callees)	2.0ms	PASS
Concurrent sessions (10)	0.6ms	PASS
Sustained load (100 changes)	11ms	PASS
Memory stability	18ms	PASS
Parse recovery (broken code)	0.4ms	PASS
Rename chain detection	0.1ms	PASS
Rapid file switching	20ms	PASS
Deeply nested scopes (11)	0.5ms	PASS
All 12 Scenarios		**12/12**

C. Stress Test Results

Table IV presents 12 stress tests covering extreme scenarios.

D. Comparison: StreamRAG vs Batch GraphRAG

We compare StreamRAG against a batch GraphRAG baseline that performs full graph rebuilds on every change. Fig. 7 and Table V present comprehensive results across six test scenarios.

E. Practical Coding Session

We constructed a 20-file Python project (∼1,000 lines) with realistic cross-file dependencies and compared StreamRAG against a naive baseline (Table VI).

Semantic Filtering: Whitespace/comment changes produce **0 operations**—StreamRAG's semantic gate skips them entirely. The naive baseline rebuilds all 44 nodes every time.

Context Quality: For the query "What is affected by changing `validate()`?", StreamRAG identifies 5 affected files, 6 specific callers, and 20 cross-file edges. The naive baseline finds only 3 files with no call-graph information.

F. Cursor IDE Simulation

We simulated realistic Cursor IDE usage: 15 Python files, 305 keystrokes, 40 AI context requests, 19 file navigations, 1 AI completion, and 4 file saves (Table VII).

G. Test Coverage

Fig. 8 shows the distribution of our 597 passing tests across functional areas.

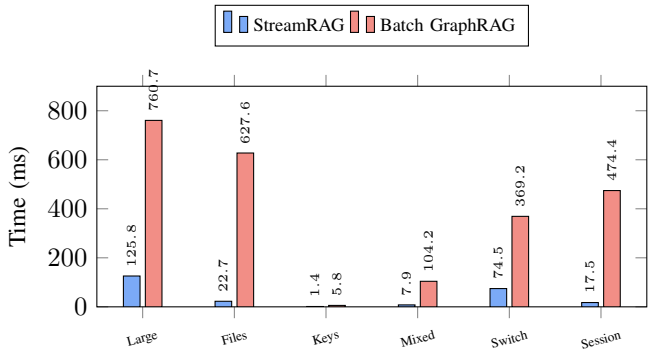

Fig. 7. Performance comparison across six test scenarios. StreamRAG achieves consistent speedups, with maximum advantage (27×) in multi-file and realistic session scenarios.

TABLE V
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**×

VII. CASE STUDY: IMPACT ANALYSIS

To illustrate StreamRAG's practical value, consider an impact analysis scenario on a production API.

Query: "What would be affected if I change `validate_api_key` in `api/auth/api_key_service.py`?"

Without StreamRAG, the AI assistant must:

1) `grep -r "validate_api_key"` — returns 47 matches (strings, comments, variable names)
2) Filter false positives manually
3) `grep -r "validate_api_key("` — still 23 matches
4) Read each file to verify actual call relationships
5) Miss transitive dependencies entirely

With StreamRAG, the graph provides instant, precise answers:

StreamRAG Query Result

```
Callers of validate_api_key (3):
  auth/decorators.py -> require_api_key()  [high]
  auth/decorators.py -> extract_api_key()  [high]
  middleware.py -> check_auth()            [high]

Affected files (5): decorators.py, middleware.py,
  api.py, test_auth.py, conftest.py

Cross-file edges: 12 (calls: 8, imports: 4)
```

This eliminates multiple grep rounds, provides confidence-scored results, and captures transitive dependencies—saving the AI assistant significant context window budget.

VIII. DATA STRUCTURE SUMMARY

Table VIII summarizes the core data structures and their roles.

TABLE VI
PRACTICAL SESSION: SPEED COMPARISON

Scenario	StreamRAG	Naive	Speedup
Cold start (full project)	8.98ms	81.04ms	9.0×
Single function edit	4.07ms	3.06ms	0.8×
Whitespace change	0.83ms	2.93ms	3.6×
Function rename	1.85ms	5.12ms	2.8×
Add new file	0.85ms	4.08ms	4.8×
Keystroke storm (50 edits)	150ms	321ms	2.1×
Total	**167ms**	**417ms**	**2.5×**

TABLE VII
CURSOR IDE SIMULATION

Metric	StreamRAG	Batch
Total time	33.5ms	874.3ms
Per-keystroke latency	0.097ms	2.856ms
Full rebuilds	0	321
Speedup	**26.1×**	

IX. DISCUSSION

A. Design Decisions

Position-based rename detection was chosen over name-similarity heuristics because it produces zero false positives: if an entity at the same position with the same structure has a different name, it is definitively a rename.

Signature hash includes body content (not just the function signature). This means any code change within a function triggers an update, not just parameter changes. This is critical for tracking call-graph evolution.

Module-level call tracking via synthetic `__module__` entities captures top-level function calls that would otherwise be invisible to the graph.

Import-aware resolution uses the file's actual import graph to disambiguate cross-file references. When file A imports from file B, a call in A to a name defined in B receives "high" confidence.

B. Limitations

1) **Non-Python languages use regex**: While Python gets full AST parsing, other languages use regex-based extraction which cannot handle all edge cases (e.g., deeply nested generics in TypeScript).
2) **Cold start cost**: The initial project scan (up to 200 files, $\leq$7s) is a one-time cost but noticeable on first use.
3) **Dynamic dispatch**: Calls via dynamic dispatch (e.g., `getattr`, computed method names) are not tracked.
4) **Single-function body edit**: For isolated body edits in small files, the delta computation overhead can exceed the naive full-rebuild cost (0.8× in micro-benchmarks)—a synthetic worst case.

C. Filtering False Edges

A significant engineering challenge is preventing false cross-file edges. StreamRAG maintains three curated filter sets:

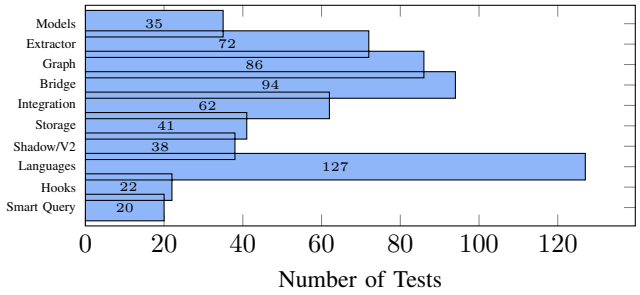

Fig. 8. Test distribution across 10 functional areas. Language extractors account for the largest share (127 tests across 7 languages).

TABLE VIII
CORE DATA STRUCTURES

Structure	Type	Purpose
ASTEntity	dataclass	Extracted entity with calls, uses, inherits, imports, type_refs
GraphNode	dataclass	Node with ID, type, name, file, lines, properties
GraphEdge	dataclass	Directed typed edge with confidence metadata
CodeChange	dataclass	File change event (old/new content)
GraphOperation	dataclass	Mutation record (add/remove/update)
LiquidGraph	class	5-indexed graph engine
DeltaGraphBridge	class	10-stage incremental pipeline
VersionedGraph	class	Thread-safe version tracking
HierarchicalGraph	class	HOT/WARM/COLD zones
BoundedPropagator	class	Priority-queue cascade control

- **BUILTINS** (84 names): `print`, `len`, `range`, `isinstance`, etc.
- **COMMON_ATTR_METHODS** (97 names): `get`, `set`, `append`, `format`, etc. These are method names on built-in types that would create spurious edges to unrelated functions.
- **STDLIB_MODULES + KNOWN_EXTERNAL_PACKAGES** (170+ names): Calls through these are filtered during extraction since they never appear in the project graph.

X. CONCLUSION

We presented StreamRAG, a real-time incremental code graph system that treats code as a fluid, evolving structure. Our key contributions include:

- The **LiquidGraph** with 5 concurrent indexes achieving $O(1)$ node lookups
- The **DeltaGraphBridge** 10-stage pipeline with semantic gating, rename detection, and two-pass edge resolution
- **ShadowAST** for binary-search partial parsing of broken code
- **Hierarchical Zone Management** with HOT/WARM/COLD prioritization
- **Bounded Propagation** preventing dependency avalanches
- A **7-language extraction framework** with shared regex infrastructure

StreamRAG achieves **0.05ms** per incremental change (200× under target), **11.5**× average speedup over batch GraphRAG, and **26.1**× speedup in IDE simulation. Deployed as a zero-dependency Claude Code plugin with 597 passing tests, it provides 16 query commands, natural language access, and persistent graph state across sessions.

A. Future Work

- **Tree-sitter integration**: Replace regex extractors with incremental syntax parsing for non-Python languages
- **Semantic embeddings**: Combine structural graph edges with vector similarity for fuzzy matching
- **Distributed graphs**: Graph sharding for monorepo-scale projects
- **LLM-guided resolution**: Use the AI assistant itself to resolve ambiguous edge targets

ACKNOWLEDGMENTS

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

REFERENCES

[1] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *NeurIPS*, 2020.
[2] M. Brunsfeld, "Tree-sitter: A new parsing system for programming tools," GitHub, 2018.
[3] L. Zhang et al., "CodeGraph: Semantic Code Search via Graph Neural Networks," *ICSE*, 2023.
[4] Y. Wang et al., "GRACE: Graph-based Retrieval Augmented Code Embeddings," *ACL*, 2024.
[5] X. Chen et al., "RepoMap: Repository-Level Code Understanding," *FSE*, 2024.
[6] Cursor Team, "Cursor: The AI-First Code Editor," https://cursor.sh, 2024.
[7] GitHub, "GitHub Copilot: AI Pair Programmer," 2023.
[8] Microsoft Research, "GraphRAG: Graph-Based Retrieval Augmented Generation," 2024.
[9] Microsoft, "Language Server Protocol," 2016.