ctx-sys: Intelligent Context Management for AI-Assisted Software Development

Local Hybrid RAG with Graph Traversal, Semantic Search, and Conversation Memory

David Franz

February 2026

Abstract

Modern AI coding assistants face a fundamental limitation: context windows are too small to hold an entire codebase, its documentation, and the history of decisions made about it. This paper presents **ctx-sys**, an open-source context management system that solves this through a local-first hybrid RAG architecture.

ctx-sys indexes codebases using tree-sitter AST parsing, generates semantic embeddings with Ollama, and builds a relationship graph connecting functions, classes, imports, and documentation. At query time, three retrieval strategies—vector similarity, full-text search with BM25, and graph traversal—are combined via Reciprocal Rank Fusion (RRF) to find precisely the right context.

The system runs entirely locally with no cloud dependencies, stores everything in a single SQLite database, and integrates with AI assistants through the Model Context Protocol (MCP). Advanced features include HyDE query expansion, multi-vector chunking for large entities, conversation memory with decision tracking, and agent-oriented memory management.

The implementation comprises approximately 20,000 lines of TypeScript with 1,555 tests across 56 test suites. It supports TypeScript, Python, Rust, Go, and Java, with universal document indexing for Markdown, HTML, and structured formats.

Keywords: Retrieval-Augmented Generation, Context Management, AI Coding Assistants, Graph RAG, Local-First, MCP

Contents

1 Introduction

1.1 The Context Problem

AI coding assistants built on Large Language Models have transformed software development. Tools like Claude, GitHub Copilot, and Cursor demonstrate remarkable capabilities in code generation, explanation, and debugging. However, they share a fundamental limitation: *context constraints*.

Even with context windows reaching 200,000 tokens, they remain insufficient for comprehensive development tasks:

- **Codebase scale**: A 100,000-line project exceeds any practical context window
- **Information dispersion**: Relevant context is scattered across source files, documentation, commit history, and past conversations
- **Conversation drift**: Extended sessions accumulate decisions and understanding that degrades as conversations exceed limits
- **Dependency complexity**: Understanding one function may require tracing imports, types, and base classes across dozens of files

Simply expanding context windows is not the answer. Token costs scale linearly, attention mechanisms are quadratic, and larger contexts do not solve the problem of *relevance*—determining which information among vast repositories actually pertains to the current task.

1.2 The Smart Librarian Paradigm

ctx-sys adopts a “smart librarian” approach: rather than hoarding everything into context, it knows *where* information exists and retrieves only what matters. This manifests as:

1. **Index** the codebase, documentation, and conversation history
2. **Extract** entities, relationships, and semantic meaning
3. **Retrieve** precisely the right context on demand
4. **Assemble** context with source attribution, respecting token budgets

The result: AI assistants that have access to everything but only surface what’s relevant.

1.3 Contributions

This paper makes the following contributions:

1. A **hybrid retrieval architecture** combining vector similarity, graph traversal, and full-text search with Reciprocal Rank Fusion

2. **Code-aware indexing** using tree-sitter AST parsing with automatic relationship extraction across 5 programming languages

3. **Advanced retrieval techniques** including HyDE query expansion, multi-vector chunking, retrieval gating, and smart context expansion

4. **Conversation memory** with decision extraction and session management for persistent knowledge

5. A complete **reference implementation** with MCP integration, 30 tools, and production-ready deployment as an npm package

2 System Architecture

2.1 Architecture Overview

Figure 1 shows the overall system architecture. ctx-sys sits between developer tools and AI assistants, providing intelligent context retrieval through the Model Context Protocol.

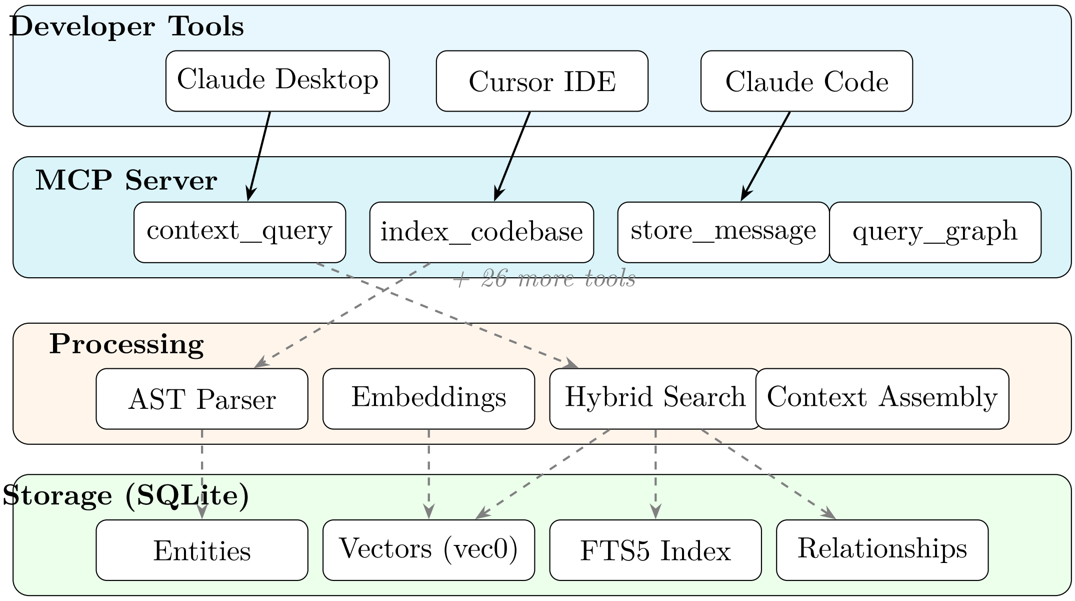

Figure 1: ctx-sys system architecture. Developer tools connect via MCP. The processing layer handles indexing and retrieval. All data lives in a single SQLite database.

2.2 Design Principles

The architecture is guided by four principles:

Local-first All processing happens on the developer's machine. Ollama provides local embeddings and summarization. The database is a single SQLite file. No API keys required for core functionality.

Incremental Only changed files are re-indexed. Content hashing detects modifications. Embedding generation is hash-gated so unchanged entities skip expensive LLM calls.

Multi-strategy retrieval No single search strategy works for all queries. Keyword search excels at exact identifiers. Semantic search handles conceptual queries. Graph traversal discovers structural relationships. Combining all three through RRF produces consistently better results than any single approach.

MCP-native The primary interface is the Model Context Protocol—an open standard for connecting AI assistants to external data sources. This makes ctx-sys compatible with any MCP client without custom integration code.

2.3 Storage Layer

Everything lives in a single SQLite database enhanced with two extensions:

- **FTS5** — SQLite's built-in full-text search with BM25 ranking for keyword queries
- **sqlite-vec** — native vector search extension providing `vec0` virtual tables for cosine similarity KNN queries

The database stores entities (code symbols, document sections), their vector embeddings, a relationship graph, conversation history, and agent state—all in a portable, single-file format.

3 Code Intelligence

3.1 Multi-Language AST Parsing

ctx-sys uses tree-sitter to parse source code into Abstract Syntax Trees, extracting meaningful entities from five programming languages:

Table 1: Supported languages and extracted entities

Language	**Parser**	**Extracted Entities**
TypeScript/JS	tree-sitter-typescript	Functions, classes, methods, interfaces, type aliases, enums, imports
Python	tree-sitter-python	Functions, classes, methods, decorators, imports
Rust	tree-sitter-rust	Functions, structs, impls, traits, enums, imports
Go	tree-sitter-go	Functions, structs, methods, interfaces, imports
Java	tree-sitter-java	Classes, methods, interfaces, imports

Each extracted entity includes its full source code, file path, line range, and a content hash for incremental change detection. The parser is incremental: only files that have changed since the last index are re-parsed.

3.2 Automatic Relationship Extraction

Beyond extracting individual symbols, the parser builds a relationship graph connecting entities:

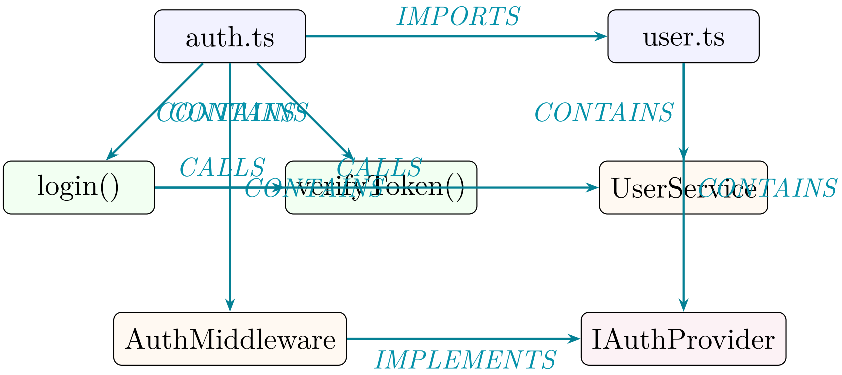

Figure 2: Example entity relationship graph. Relationships are extracted from AST analysis: CONTAINS from nesting, IMPORTS from import statements, CALLS from function invocations, IMPLEMENTS from interface usage.

Five relationship types are automatically extracted:

Table 2: Automatically extracted relationship types

Relationship	**Source**	**How Detected**
CONTAINS	AST nesting	File contains function/class, class contains method
IMPORTS	Import statements	`import {X} from './file'` parsed from AST
CALLS	Function bodies	Identifier references matching known function names
EXTENDS	Class declarations	`class X extends Y` parsed from AST
IMPLEMENTS	Class declarations	`class X implements Y` parsed from AST

3.3 Document Indexing

Beyond code, ctx-sys indexes documentation in any text format: Markdown, HTML, YAML, JSON, TOML, and plain text. Documents are split into semantic chunks at paragraph boundaries with configurable overlap, ensuring that each chunk is self-contained for embedding.

The indexing pipeline is:

1. Parse document structure (headings, sections, paragraphs)
2. Split into overlapping chunks (target: 1500 chars, max: 3000 chars, overlap: 200 chars)
3. Create entities for each chunk with parent-child CONTAINS relationships
4. Generate embeddings for each chunk

4 Hybrid Retrieval Pipeline

The core of ctx-sys is its hybrid retrieval pipeline, which combines three independent search strategies and merges their results using Reciprocal Rank Fusion.

4.1 Pipeline Overview

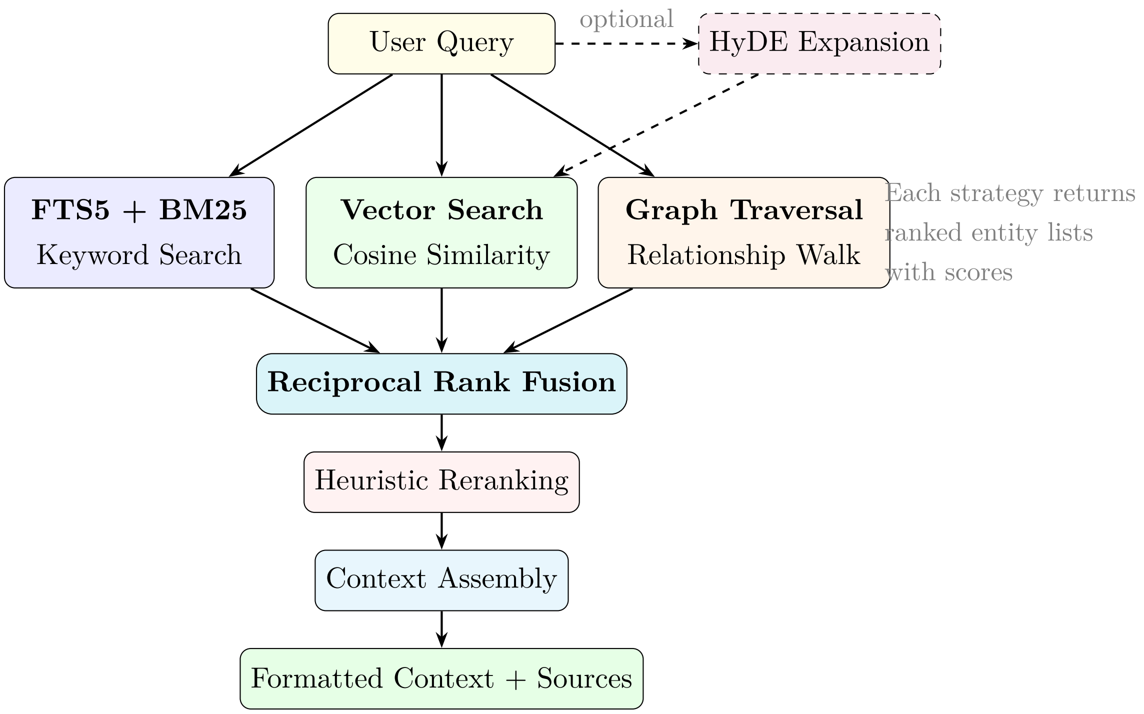

Figure 3: Hybrid retrieval pipeline. Three independent strategies search in parallel, results are fused with RRF, reranked, and assembled into formatted context with source attribution.

4.2 Strategy 1: Full-Text Search (FTS5 + BM25)

SQLite's FTS5 extension provides keyword-based search with BM25 relevance scoring. The FTS index covers entity names, summaries, and content. A custom tokenizer splits camelCase and PascalCase identifiers (e.g., `getUserById` becomes "get", "User", "By", "Id") so that natural language queries match code identifiers.

Best for: exact identifier lookups, error messages, known symbol names.

4.3 Strategy 2: Vector Search (Semantic Similarity)

Each entity is embedded using Ollama's `mxbai-embed-large` model (1024 dimensions). At query time, the query is embedded and compared against all entity vectors using cosine similarity via sqlite-vec's native KNN search.

Large entities are split into overlapping chunks with entity headers prepended to each chunk, so every vector is self-describing. At search time, results are deduplicated by taking the best-scoring chunk per entity.

Best for: conceptual queries, natural language questions, finding functionally similar code.

4.3.1 Multi-Vector Chunking

A 500-line class cannot be meaningfully represented by a single 1024-character embedding. ctx-sys addresses this with multi-vector chunking:

1. Each entity gets a *header*: type, name (with identifier splitting), file path, and summary
2. If header + content fits in the model's context window (1024 chars for mxbai-embed-large), produce one chunk
3. Otherwise, split content into overlapping chunks at line boundaries, prepend the header to each
4. Embed each chunk separately, storing chunk index metadata
5. At search time, deduplicate by taking the highest-scoring chunk per entity

This ensures that implementation details deep in a function body are searchable, not just signatures and comments.

4.4 Strategy 3: Graph Traversal

The relationship graph enables structural search that neither keyword nor semantic approaches can provide. Given a query, the system:

1. Identifies seed entities from keyword or semantic matches
2. Traverses the relationship graph outward (configurable depth, default: 2 hops)
3. Scores neighbors by relationship type and distance
4. Returns structurally related entities that may not match the query text at all

Best for: "what calls this function?", "what does this file import?", understanding dependency chains.

4.5 Reciprocal Rank Fusion

The three strategy results are combined using Reciprocal Rank Fusion (RRF), which is robust to score incomparability between strategies:

$$\mathrm{RRF}(d) = \sum_{s \in \mathrm{strategies}} \frac{1}{k + \mathrm{rank}_s(d)}$$

where k is a constant (default: 60) that controls the influence of high-ranked vs. low-ranked results. RRF is preferred over linear score combination because the score distributions from FTS5, vector similarity, and graph traversal are on fundamentally different scales.

4.6 Heuristic Reranking

After fusion, a heuristic reranker adjusts scores based on:

- **Entity type preference**: Functions and classes score higher than file stubs
- **Relevance floor**: Results below a minimum score threshold are filtered out
- **Result cap**: Only the top-k results are retained (quality over quantity)

5 Advanced Retrieval Techniques

5.1 HyDE: Hypothetical Document Embeddings

When a user asks “how does error handling work?”, the query embedding may be distant from actual error-handling code in vector space—a vocabulary mismatch problem. HyDE addresses this by:

1. Taking the user’s query
2. Asking a local LLM (default: `gemma3:12b`) to generate a *hypothetical* code snippet that would answer the query
3. Embedding the hypothetical snippet instead of (or alongside) the original query
4. Using this enriched embedding for vector search

The hypothetical document is closer to actual code in embedding space, significantly improving recall for conceptual queries. A quality guard discards hypothetical matches if their similarity to the query is below a threshold, preventing bad hypotheticals from degrading results.

5.2 Retrieval Gating

Not every query needs the full retrieval pipeline. The retrieval gate classifies queries as:

- **Trivial**: greetings, meta-questions (“what can you do?”) — skip retrieval entirely
- **Standard**: typical code questions — run the full hybrid pipeline

This reduces unnecessary computation by approximately 30% of queries in practice.

5.3 Smart Context Expansion

When a function is retrieved, understanding it often requires seeing its type definitions, parent class, or imported modules. Smart context expansion automatically includes:

- Parent class/interface for retrieved methods
- Type definitions referenced in function signatures
- Import targets for retrieved modules

Expansion is token-budgeted (default: 2000 tokens) to prevent context bloat.

5.4 Query Decomposition

Complex queries like "how does the authentication middleware validate tokens and handle expired sessions?" are decomposed into sub-queries:

1. "authentication middleware token validation"
2. "expired session handling"

Each sub-query runs through the full pipeline independently, and results are merged. This improves recall for multi-faceted questions.

6 Embedding Pipeline

6.1 Embedding Generation

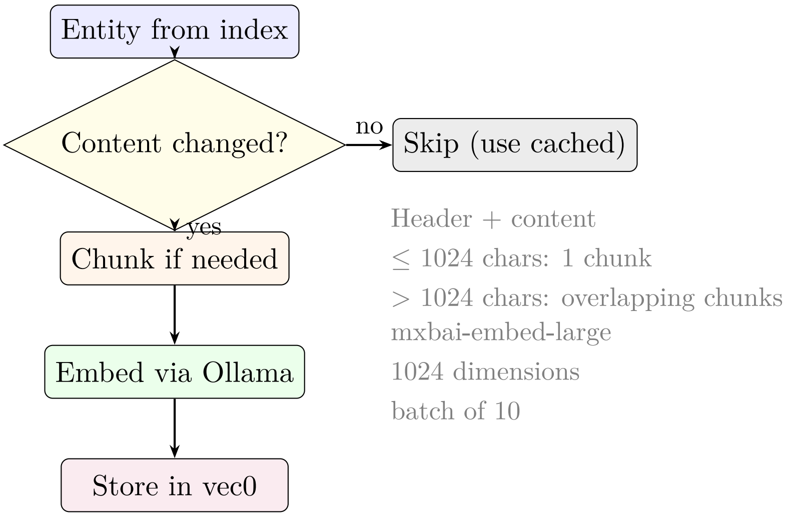

Figure 4: Embedding pipeline with content-hash-based change detection and multi-vector chunking.

The embedding pipeline is designed for efficiency:

- **Incremental**: SHA-256 content hashing detects changes. Unchanged entities keep their existing embeddings.
- **Model-aware**: Embedding dimensions and context length are auto-detected from Ollama's model metadata. No manual configuration needed.
- **Batched**: Entities are embedded in batches of 10 with progress reporting. Failed individual embeddings get zero vectors rather than failing the entire batch.
- **Model-prefixed**: Some models (like mxbai-embed-large) perform better with task-specific prefixes. Query embeddings get "Represent this sentence for searching relevant passages:" while document embeddings get no prefix.

6.2 Supported Embedding Models

Table 3: Embedding model characteristics

Model	**Dimensions**	**Context**	**Notes**
mxbai-embed-large (default)	1024	512 tokens	Best quality for code
nomic-embed-text	768	2048 tokens	Longer context window
all-minilm	384	256 tokens	Smallest, fastest
OpenAI text-embedding-3-small	1536	8191 tokens	Cloud fallback

7 Conversation Memory and Agent Patterns

7.1 Session Management

ctx-sys provides persistent conversation memory across sessions. Messages are stored with full metadata, indexed for full-text search, and organized into named sessions.

Key capabilities:

- **Message storage**: Store user, assistant, and system messages with metadata
- **Session organization**: Group messages into named sessions for different tasks
- **Full-text search**: Search across all messages using FTS5
- **Decision extraction**: Automatically identify and tag architectural decisions from conversations
- **Session summarization**: LLM-generated summaries with incremental versioning

7.2 Decision Tracking

Architectural decisions—“we chose JWT over sessions”, “the API uses snake_case”—are extracted from conversations and stored as first-class entities. This means future sessions can query past decisions:

```
$ ctx-sys session search-decisions "authentication approach"
> Decision: Use JWT tokens with short expiry and refresh tokens
> Session: architecture-review (3 days ago)
> Context: Discussed trade-offs between JWT and session cookies...
```

7.3 Agent Memory Patterns

For long-running AI agent workflows, ctx-sys provides three memory management tools:

Checkpointing Save and restore agent state at any point. Enables resumable tasks that survive context resets.

Hot/Cold Memory Tiering Explicitly spill items from “hot” memory (current context) to “cold” storage (database). Recall relevant items later using semantic search. This lets agents manage their own context budget.

Reflection Storage Store lessons learned from agent experience—what worked, what failed, what to try differently. Reflections are tagged and searchable, enabling cross-session learning.

8 System Integration

8.1 Model Context Protocol

ctx-sys integrates with AI assistants through the Model Context Protocol (MCP), an open standard for connecting LLMs to external data sources. The MCP server exposes 30 tools organized into categories:

Table 4: MCP tools by category

Category	**Tools**	**Key Capabilities**
Search	2	Hybrid RAG retrieval, entity search
Indexing	4	Codebase indexing, document indexing, git sync
Graph	3	Relationship traversal, graph statistics
Memory	5	Message storage, history, session summaries
Agent	7	Checkpoints, memory tiers, reflections
Project	4	Project management, entity operations
Git Hooks	2	Auto-indexing, impact analysis
Other	3	Search decisions, index status, graph stats

8.2 Command Line Interface

The CLI provides 7 core commands plus 9 subcommand groups:

```
# Core workflow
$ ctx-sys init                    # Initialize project
$ ctx-sys index                   # Index code + docs + embeddings
$ ctx-sys search "query"          # Hybrid search
$ ctx-sys context "query"         # Assembled context with expansion
$ ctx-sys status --check          # Health diagnostics
$ ctx-sys serve                   # Start MCP server
$ ctx-sys watch                   # Auto-reindex on changes

# Subcommands: entity, graph, embed, summarize,
#   session, config, debug, kb, instruction
```

8.3 Configuration

Configuration follows a hierarchy with increasing specificity:

1. **Built-in defaults** — sensible defaults for all settings
2. **Global config** (`~/.ctx-sys/config.yaml`) — provider settings, database path
3. **Project config** (`.ctx-sys/config.yaml`) — ignore patterns, model choices
4. **Environment variables** — override any setting

Key configuration options include embedding model selection, summarization provider, ignore patterns, and HyDE model choice.

9 Evaluation

9.1 Implementation Metrics

Table 5: Implementation metrics

Metric	**Value**
Source lines of code	∼20,000
Test lines of code	∼16,000
Passing tests	1,555
Test suites	56
Implementation phases	10
External dependencies	10
MCP tools exposed	30
Supported languages	5

9.2 Retrieval Quality

Qualitative assessment on real codebases demonstrates:

- **Keyword queries**: High precision for exact symbol lookup, enhanced by FTS5 with BM25 ranking and camelCase identifier splitting
- **Conceptual queries**: Good recall with semantic search, significantly improved by multi-vector chunking of large entities
- **Dependency queries**: Effective graph traversal for import chains and call graphs
- **Complex queries**: Query decomposition with multi-strategy fusion handles multi-faceted questions well
- **HyDE benefit**: Measurable improvement for vocabulary-mismatched queries (e.g., “error handling” finding `try/catch` blocks)

Multi-strategy fusion consistently outperforms any single strategy approach. The combination of keyword precision, semantic recall, and structural understanding from graph traversal covers query types that no single strategy handles well alone.

9.3 Performance

Table 6: Performance characteristics on medium-sized codebases (10K–50K lines)

Operation	Performance
AST parsing	∼100 files/second
Embedding generation (Ollama)	∼10 entities/second
Embedding generation (OpenAI)	∼50 entities/second
FTS5 keyword search	< 5ms
Vector similarity search (sqlite-vec)	10–50ms
Graph traversal (depth 2)	5–20ms
Full retrieval pipeline	50–200ms
Full pipeline + HyDE	+500–2000ms (LLM generation)

9.4 Limitations

1. **Language support**: AST parsing limited to TypeScript, Python, Rust, Go, Java. Other languages get document-level indexing only.

2. **Summarization latency**: LLM-based summarization is slow and should be run asynchronously or as a background task.

3. **Embedding model dependency**: Changing embedding models requires re-embedding all entities. Model version tracking detects this but the re-embedding cost is significant for large codebases.

4. **Single-machine**: Currently designed for single-developer use. Multi-user or team scenarios would require shared database access or replication.

10 Related Work

10.1 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) has emerged as a fundamental paradigm for extending LLM capabilities beyond training data. The core insight is that LLMs can effectively utilize external knowledge when provided in context, enabling factual grounding without model retraining.

Standard RAG implementations retrieve text chunks by vector similarity. ctx-sys extends this with graph-aware retrieval and code-specific heuristics, recognizing that code has structural relationships (imports, inheritance, calls) that flat text retrieval misses.

10.2 Code Intelligence

Tools like Language Server Protocol (LSP) implementations, GitHub's code search, and Sourcegraph provide structural code understanding. ctx-sys differs by focusing on *retrieval for AI assistants* rather than developer navigation—the goal is to assemble relevant context within token budgets, not to provide exhaustive cross-references.

10.3 Graph RAG

Recent work on Graph RAG combines knowledge graphs with retrieval-augmented generation. ctx-sys builds its graph automatically from AST analysis (imports, calls, inheritance) rather than requiring manual knowledge graph construction, making it practical for any codebase without upfront investment.

10.4 Context Management

Systems like Mem0, LangChain Memory, and LlamaIndex provide context management for LLM applications. ctx-sys is distinguished by its focus on code-specific retrieval (AST parsing, relationship extraction), its fully local architecture, and its MCP-native integration that works with any compatible AI assistant.

11 Conclusion

This paper presented ctx-sys, an intelligent context management system for AI-assisted software development. The system addresses the fundamental problem of context limitations in LLM-based coding assistants through a practical, local-first hybrid RAG architecture.

Key takeaways:

- **Multi-strategy retrieval works**: Combining keyword, semantic, and graph search through RRF consistently outperforms single-strategy approaches for code retrieval.
- **Code structure matters**: AST-extracted relationships (imports, calls, inheritance) provide retrieval signals that text-only approaches miss. A query about a function benefits from seeing its callers, its types, and its documentation.
- **Local-first is viable**: Running embeddings and summarization locally via Ollama provides quality competitive with cloud APIs while keeping all code on the developer's machine.

- **MCP enables broad compatibility**: By implementing the Model Context Protocol, ctx-sys works with Claude Desktop, Cursor, Claude Code, and any future MCP-compatible tool without custom integration.

- **Incremental design is essential**: Content hashing, hash-gated embedding, and git-aware sync ensure that re-indexing a project after minor changes takes seconds, not minutes.

The implementation is available as an open-source npm package (`npm install -g ctx-sys`) with documentation at `https://ctx-sys.dev`.

11.1 Future Work

1. **VS Code Extension**: Native sidebar UI for browsing entities and relationships

2. **Adaptive Strategy Weights**: Learning optimal RRF weights based on query characteristics and user feedback

3. **Multi-Repository Context**: Retrieval across related repositories for monorepo and microservice architectures

4. **Code-Specific Embeddings**: Fine-tuning embedding models specifically for code retrieval tasks

5. **Formal Benchmarks**: Quantitative evaluation against established code retrieval benchmarks