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

A Hybrid Retrieval Architecture Combining Vector Embeddings, Graph Traversal, and Semantic Analysis for Scalable Code Understanding

David Franz

February 2026

Abstract

Modern AI coding assistants, built upon Large Language Models (LLMs), face fundamental limitations in context management. As conversations extend and codebases grow beyond context window capacities, these systems lose access to critical information—past decisions, code dependencies, documentation, and architectural context. This paper presents **ctx-sys**, an intelligent context management framework that addresses these limitations through a hybrid retrieval architecture.

The system implements a "smart librarian" paradigm: rather than attempting to fit all information into limited context windows, it indexes, extracts semantic meaning, and retrieves precisely relevant context on demand. The architecture integrates multiple retrieval strategies—vector similarity search using dense embeddings, graph-based traversal of code relationships, and keyword-based full-text search with BM25 ranking—unified through Reciprocal Rank Fusion (RRF).

Key contributions include: (1) a unified entity model supporting heterogeneous information types including code symbols, documentation sections, conversation history, and domain concepts; (2) multi-language AST parsing using tree-sitter for structural code understanding; (3) advanced retrieval techniques including HyDE (Hypothetical Document Embeddings) for vocabulary mismatch resolution, retrieval gating for computational efficiency, smart context expansion, and query decomposition with LLM re-ranking; (4) a draft-critique loop for hallucination detection; (5) agent-oriented memory management with explicit hot/cold tiering and checkpointing for resumable execution; and (6) overlapping chunk embeddings and incremental document updates for improved retrieval quality.

The implementation comprises approximately 20,000 lines of TypeScript with comprehensive test coverage (1,555 tests across 56 test suites spanning 10 implementation phases). The storage layer uses native SQLite via better-sqlite3 with FTS5 full-text search and BM25 ranking for high-performance local-first operation. Integration with AI assistants is achieved through the Model Context Protocol (MCP), enabling compatibility with Claude, GitHub Copilot, and similar tools. Evaluation demonstrates significant improvements in context relevance and token efficiency compared to naive full-context approaches.

Keywords: Retrieval-Augmented Generation, Context Management, AI Coding Assistants, Graph RAG, Code Intelligence, Embedding Search

1 Introduction

1.1 Motivation

The integration of Large Language Models (LLMs) into software development workflows has transformed how developers write, understand, and maintain code. Tools such as GitHub Copilot, Claude, and ChatGPT have demonstrated remarkable capabilities in code generation, explanation, and debugging. However, these systems share a fundamental limitation: *context constraints.*

Modern LLMs operate within fixed context windows—typically ranging from 8,000 to 200,000 tokens—that must accommodate both the input prompt and the generated response. While these windows have expanded significantly over recent years, they remain insufficient for comprehensive software development tasks for several reasons:

1. **Codebase Scale**: Production codebases routinely contain millions of lines of code across thousands of files. Even a moderately-sized project of 100,000 lines exceeds any practical context window.

2. **Information Dispersion**: Relevant context for a given task is typically scattered across multiple files, documentation, commit history, issue trackers, and past conversations.

3. **Conversation Drift**: Extended development sessions accumulate decisions, clarifications, and contextual understanding that degrades as conversations exceed context limits.

4. **Dependency Complexity**: Understanding a single function may require tracing imports, type definitions, base classes, and documentation across dozens of files.

The naive approach of simply expanding context windows is economically and computationally prohibitive. Token costs scale linearly or super-linearly with context size, and attention mechanisms exhibit quadratic complexity. More fundamentally, larger contexts do not solve the problem of *relevance*—determining which information among vast repositories actually pertains to the current task.

1.2 Problem Statement

This paper addresses the following research question:

> *How can we design a context management system that enables AI coding assistants to access relevant information from large, heterogeneous knowledge sources while minimizing token consumption and maximizing retrieval precision?*

This decomposes into several sub-problems:

1. How should diverse information types (code, documentation, conversations, domain knowledge) be represented in a unified retrieval system?
2. What retrieval strategies are most effective for different query types, and how should multiple strategies be combined?
3. How can structural code understanding (AST analysis) enhance retrieval beyond surface-level text matching?
4. How should conversation history be managed to preserve important decisions while respecting context limits?
5. What mechanisms can detect and prevent hallucinations when LLMs generate responses based on retrieved context?

1.3 Contributions

This paper makes the following contributions:

1. **Unified Entity Model**: A flexible data model that represents code symbols, documentation sections, conversation messages, and domain concepts as first-class entities with vector embeddings and typed relationships.
2. **Hybrid Retrieval Architecture**: A multi-strategy search system combining vector similarity, graph traversal, and full-text search with Reciprocal Rank Fusion for result combination.
3. **Code Intelligence Pipeline**: Integration of tree-sitter AST parsing with AI-powered summarization for structural code understanding across multiple programming languages.

4. **Advanced Retrieval Techniques**: Implementation of HyDE query expansion for vocabulary mismatch resolution, retrieval gating for computational efficiency, smart context expansion, and query decomposition with LLM re-ranking.

5. **Verification Mechanisms**: A draft-critique loop that validates LLM responses against retrieved context to detect hallucinations.

6. **Agent Memory Management**: Hot/cold memory tiering with explicit APIs and checkpointing for resumable long-running tasks.

7. **Reference Implementation**: A complete, tested implementation (ctx-sys) with MCP integration for practical deployment.

1.4 Paper Structure

The remainder of this paper is organized as follows:

Section 2 reviews related work in retrieval-augmented generation, code intelligence, and context management for LLMs.

Section 3 presents the system architecture, design principles, and component interactions.

Section 4 details the data model, database schema, and storage layer implementation.

Section 5 describes the code intelligence pipeline including AST parsing, symbol extraction, and relationship analysis.

Section 6 covers the embedding and retrieval system, including multi-strategy search and fusion algorithms.

Section 7 explains advanced retrieval techniques: HyDE expansion, retrieval gating, smart context expansion, query decomposition, and the draft-critique loop.

Section 8 discusses conversation management, session handling, and decision extraction.

Section 9 presents the agent patterns: checkpointing, memory tiering, and proactive context.

Section 10 describes the system integration through MCP, CLI, and configuration management.

Section 11 presents evaluation methodology and results.

Section 12 concludes with a discussion of limitations and future work.

2 Related Work

2.1 Retrieval-Augmented Generation

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

2.1.1 Dense Retrieval

Traditional information retrieval relied on sparse representations such as TF-IDF and BM25. The advent of dense retrieval, pioneered by systems like DPR (Dense Passage Retrieval), demonstrated that learned embeddings could significantly outperform lexical methods for semantic similarity tasks.

Modern embedding models, such as OpenAI's text-embedding-3-small and open-source alternatives like nomic-embed-text, map text to high-dimensional vectors where semantic similarity corresponds to geometric proximity. These embeddings form the foundation of our vector similarity search.

2.1.2 Graph-Enhanced RAG

Recent work has explored augmenting vector retrieval with knowledge graphs. Microsoft's GraphRAG demonstrated that graph structures capturing entity relationships can significantly improve retrieval quality for complex queries requiring multi-hop reasoning.

Our system implements a similar approach, though optimized for code-specific relationships (imports, inheritance, function calls) rather than general knowledge graphs.

2.1.3 Query Expansion Techniques

Query expansion addresses the vocabulary mismatch problem—users often phrase queries differently from how information is stored. HyDE (Hypothetical Document Embeddings) introduced the technique of generating hypothetical answers and using their embeddings for retrieval, which we implement as a configurable option for conceptual queries.

2.2 Code Intelligence

2.2.1 Abstract Syntax Tree Analysis

Abstract Syntax Tree (AST) parsing provides structural understanding of source code beyond surface-level text processing. Tree-sitter, a parser generator tool, has become the de facto standard for multi-language parsing in developer tools, offering incremental parsing, error recovery, and language-agnostic APIs.

Our implementation uses tree-sitter's WebAssembly variant (web-tree-sitter) to support parsing without native dependencies, enabling broader deployment scenarios.

2.2.2 Code Search and Retrieval

Academic and industrial systems have explored various approaches to code search:

- **Lexical**: Traditional search engines treating code as text
- **Semantic**: Embedding-based similarity (CodeBERT, CodeT5)
- **Structural**: Graph-based representations (code2vec)

ctx-sys combines these approaches, using lexical search for exact matches, semantic embeddings for conceptual queries, and graph traversal for dependency analysis.

2.2.3 Code Summarization

Automatic code summarization—generating natural language descriptions of code—enhances searchability and comprehension. We employ LLM-based summarization using the qwen3:0.6b model with specialized prompts for different symbol types (functions, classes, modules).

2.3 Context Management for LLMs

2.3.1 Conversation Memory

Managing conversation history in multi-turn dialogues is a recognized challenge. Approaches include:

- **Truncation**: Removing old messages (loses information)
- **Summarization**: Compressing history into summaries
- **Retrieval**: Selectively retrieving relevant past messages

ctx-sys implements summarization for session archival and retrieval for historical context access.

2.3.2 Memory-Augmented Agents

Long-running AI agents require persistent memory beyond conversation windows. Systems like MemGPT introduced explicit memory hierarchies. Our hot/cold memory tiering draws inspiration from this work while providing more explicit developer control.

2.4 Hallucination Detection

LLM hallucinations—generating plausible but incorrect information—are particularly problematic in code generation where errors have concrete consequences. Our draft-critique loop implements self-verification, where the model critiques its own outputs against retrieved context, similar to self-consistency approaches.

3 System Architecture

3.1 Design Principles

The ctx-sys architecture is guided by several core principles:

3.1.1 Smart Librarian Paradigm

Rather than attempting to include all potentially relevant information in context (the "hoarder" approach), ctx-sys acts as a "smart librarian"—knowing where information exists and retrieving precisely what is needed on demand. This principle manifests in:

- Comprehensive indexing of all information sources
- Query understanding to determine actual information needs
- Targeted retrieval within strict token budgets
- Source attribution for transparency

3.1.2 Unified Entity Model

All information types—code symbols, documentation sections, conversation messages, domain concepts—are represented as *entities* with common attributes:

- Unique identifiers and type classifications
- Textual content with optional summaries
- Vector embeddings for semantic search
- Typed relationships to other entities
- Rich metadata for filtering and ranking

This unification enables consistent retrieval across heterogeneous sources.

3.1.3 Strategy Plurality

No single retrieval strategy is optimal for all query types. The system supports multiple strategies:

- **Vector/Semantic**: Best for conceptual, explanatory queries
- **Keyword/FTS**: Best for exact symbol or identifier lookup
- **Graph**: Best for dependency and relationship queries

Results are combined through configurable fusion algorithms.

3.1.4 Local-First Architecture

The system prioritizes local execution:

- Native SQLite database (via better-sqlite3) with FTS5 full-text search for storage
- Local embedding providers (Ollama) as primary option
- Cloud providers as configurable fallbacks
- Single-file portable database

This design ensures privacy, reduces latency, and enables offline operation.

3.2 System Components

Figure 1 illustrates the high-level system architecture.

3.2.1 Integration Layer

The integration layer exposes ctx-sys functionality to external tools:

MCP Server The primary integration point, implementing the Model Context Protocol for compatibility with Claude, Copilot, and other MCP-aware assistants. Exposes tools for querying, indexing, and memory management.

CLI Interface Command-line tools for initialization, indexing, search, and server management. Used for setup, debugging, and automation scripts.

VS Code Extension (Planned) Native IDE integration with sidebar panel, hover information, and command palette access.

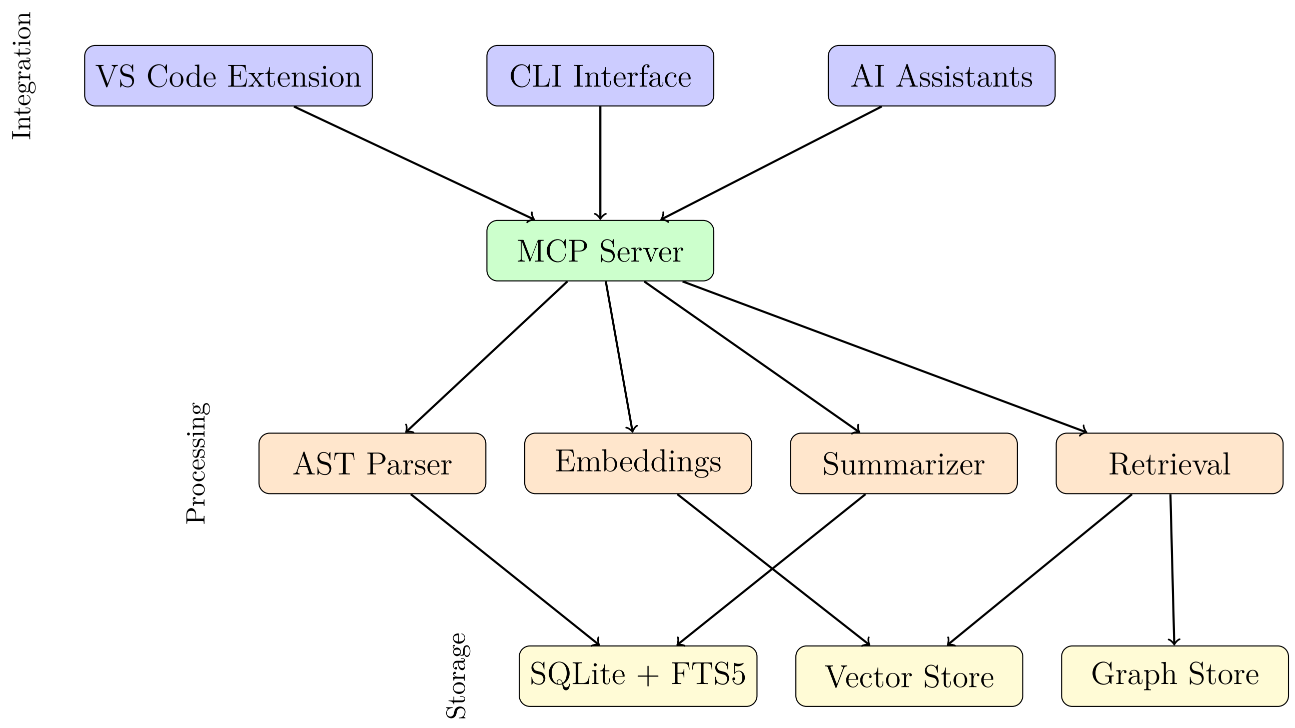

Figure 1: High-level system architecture showing the three-layer design: integration, processing, and storage layers.

3.2.2 Processing Layer

The processing layer transforms raw data into searchable representations:

AST Parser Multi-language source code parsing using tree-sitter. Extracts symbols, imports, exports, and structural relationships.

Embedding Generator Converts text content to dense vector representations. Supports multiple providers (Ollama, OpenAI) with automatic fallback.

Summarizer Generates natural language descriptions of code symbols and document sections using LLM inference (qwen3:0.6b via Ollama).

Retrieval Engine Orchestrates multi-strategy search, fusion, and context assembly.

3.2.3 Storage Layer

The storage layer persists all indexed data:

SQLite Database Primary storage using better-sqlite3 (native SQLite bindings for Node.js) with FTS5 full-text search and BM25 ranking. Stores entities, metadata, sessions, and configuration.

Vector Store Embedding vectors stored alongside entities, with cosine similarity computation for semantic search.

Graph Store Entity relationships stored as edges with types and weights, supporting efficient traversal queries.

3.3 Data Flow

3.3.1 Indexing Flow

When indexing a codebase, data flows through the system as follows:

1. **Discovery**: File system traversal identifies source files, filtered by language support and ignore patterns.
2. **Parsing**: Each file is parsed by the AST parser to extract symbols (functions, classes, methods) with their signatures, bodies, and positions.
3. **Relationship Extraction**: Import statements and inheritance declarations are analyzed to create relationship edges.
4. **Summarization**: (Optional) Each symbol is summarized by an LLM to generate natural language descriptions.
5. **Embedding**: Symbol content and summaries are embedded to dense vectors, using overlapping chunk embeddings for improved retrieval coverage.
6. **Storage**: Entities, vectors, and relationships are persisted to the database.

3.3.2 Query Flow

When processing a context query:

1. **Query Analysis**: The natural language query is parsed to extract intent, keywords, and entity mentions. For complex queries, query decomposition breaks the query into sub-queries for more targeted retrieval.
2. **Retrieval Gating**: (Optional) A fast check determines if retrieval is necessary or if the query can be answered from general knowledge.
3. **HyDE Expansion**: (Optional) For conceptual queries, a hypothetical answer is generated and embedded.
4. **Multi-Strategy Search**: Parallel execution of vector similarity, keyword matching, and graph traversal.
5. **Fusion and Re-ranking**: Results from all strategies are combined using Reciprocal Rank Fusion, then optionally re-ranked using LLM-based relevance scoring.

6. **Smart Context Expansion**: Top results are expanded with surrounding context—related entities, parent scopes, and sibling definitions—to provide coherent context windows.

7. **Context Assembly**: Results are formatted with source attribution, respecting token budgets.

8. **Critique**: (Optional) The assembled context is verified against claimed facts.

4 Data Model and Storage

4.1 Entity Model

The fundamental unit of storage in ctx-sys is the *entity*. An entity represents any piece of information that may be relevant for context retrieval.

4.1.1 Entity Types

Entities are classified by type, enabling type-specific processing and filtering:

Table 1: Entity type taxonomy

Category	**Type**	**Description**
Code	`file`	Source code file
	`module`	Module/namespace
	`class`	Class definition
	`function`	Function/procedure
	`method`	Class method
	`interface`	Interface/protocol
	`type`	Type alias/definition
Documentation	`document`	Document file
	`section`	Document section
	`requirement`	Extracted requirement
	`user_story`	User story
Conversation	`session`	Conversation session
	`message`	Individual message
	`decision`	Extracted decision
	`question`	Unanswered question
Domain	`concept`	Domain concept
	`person`	Person/stakeholder
	`technology`	Technology reference
	`pattern`	Design pattern

4.1.2 Entity Schema

Each entity contains the following fields:

```
interface Entity {
  id: string;                // Unique identifier (UUID)
  type: EntityType;          // Classification type
  name: string;              // Display name
  qualifiedName?: string;    // Fully qualified name (e.g., module.Class.
    method)
  content?: string;          // Full text content
  summary?: string;          // Generated summary
  metadata: Record<string, unknown>;  // Type-specific metadata
  filePath?: string;         // Source file path (for code entities)
  startLine?: number;        // Start line in source
  endLine?: number;          // End line in source
  hash?: string;             // Content hash for change detection
  createdAt: Date;
  updatedAt: Date;
}
```

Listing 1: Entity interface definition

The `metadata` field accommodates type-specific information:

- **Functions**: Parameters, return type, async flag, exported flag
- **Classes**: Base classes, implemented interfaces, member counts
- **Documents**: Frontmatter, word count, heading structure
- **Messages**: Role (user/assistant), session reference

4.2 Relationship Model

Entities are connected through typed, weighted relationships enabling graph traversal.

4.2.1 Relationship Types

4.2.2 Relationship Schema

```
interface Relationship {
  id: string;
  sourceId: string;
  targetId: string;
  relationship: RelationshipType;
  weight: number;             // Edge weight (0-1)
  metadata?: Record<string, unknown>;
  createdAt: Date;
```

Table 2: Relationship type taxonomy

Type	Source	Target	Meaning
`contains`	Parent	Child	Structural containment
`imports`	File	File/Module	Module dependency
`extends`	Class	Class	Inheritance
`implements`	Class	Interface	Interface implementation
`calls`	Function	Function	Function call
`defines`	File	Symbol	Symbol definition
`references`	Any	Any	Generic reference
`depends_on`	Any	Any	Logical dependency
`relates_to`	Any	Any	Semantic similarity

```
}
```

Listing 2: Relationship interface definition

The `weight` field encodes relationship strength:

- 1.0: Direct, definite relationship (explicit import, inheritance)
- 0.5-0.9: Inferred relationship (likely call, type usage)
- 0.1-0.5: Weak relationship (semantic similarity)

4.3 Database Schema

The SQLite database uses a hybrid of global and per-project tables.

4.3.1 Global Tables

```sql
-- Project registry
CREATE TABLE projects (
  id TEXT PRIMARY KEY,
  name TEXT UNIQUE NOT NULL,
  path TEXT NOT NULL,
  config JSON,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  last_indexed_at DATETIME,
  last_sync_commit TEXT
);

-- Embedding model registry
CREATE TABLE embedding_models (
  id TEXT PRIMARY KEY,
  name TEXT NOT NULL,
  provider TEXT NOT NULL,
  dimensions INTEGER NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Global configuration
CREATE TABLE config (
  key TEXT PRIMARY KEY,
  value JSON
);
```

Listing 3: Global schema

4.3.2 Per-Project Tables

Each project has isolated tables with a sanitized project ID prefix:

```sql
-- Entities
CREATE TABLE {prefix}_entities (
  id TEXT PRIMARY KEY,
  type TEXT NOT NULL,
  name TEXT NOT NULL,
  qualified_name TEXT,
  content TEXT,
  summary TEXT,
  metadata JSON,
  file_path TEXT,
  start_line INTEGER,
  end_line INTEGER,
  hash TEXT,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Vector embeddings (stored as JSON)
CREATE TABLE {prefix}_vectors (
  id TEXT PRIMARY KEY,
  entity_id TEXT NOT NULL,
  model_id TEXT NOT NULL,
  embedding JSON NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (entity_id) REFERENCES {prefix}_entities(id)
);

-- Graph relationships
CREATE TABLE {prefix}_relationships (
  id TEXT PRIMARY KEY,
  source_id TEXT NOT NULL,
  target_id TEXT NOT NULL,
  relationship TEXT NOT NULL,
  weight REAL DEFAULT 1.0,
  metadata JSON,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (source_id) REFERENCES {prefix}_entities(id),
  FOREIGN KEY (target_id) REFERENCES {prefix}_entities(id)
);

-- Conversation sessions
CREATE TABLE {prefix}_sessions (
  id TEXT PRIMARY KEY,
  name TEXT,
  status TEXT DEFAULT 'active',
  summary TEXT,
  message_count INTEGER DEFAULT 0,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Conversation messages
CREATE TABLE {prefix}_messages (
  id TEXT PRIMARY KEY,
  session_id TEXT NOT NULL,
  role TEXT NOT NULL,
  content TEXT NOT NULL,
  metadata JSON,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (session_id) REFERENCES {prefix}_sessions(id)
);

-- FTS5 virtual table for full-text search with BM25 ranking
CREATE VIRTUAL TABLE {prefix}_entities_fts USING fts5(
  name, qualified_name, content, summary,
  content={prefix}_entities, content_rowid=rowid
);
```

Listing 4: Per-project schema (abbreviated)

4.3.3 Indexing Strategy

Indexes are created for common query patterns:

```sql
-- Entity lookups
CREATE INDEX idx_{prefix}_entities_type ON {prefix}_entities(type);
CREATE INDEX idx_{prefix}_entities_file ON {prefix}_entities(file_path)
    ;
CREATE INDEX idx_{prefix}_entities_name ON {prefix}_entities(name);
CREATE INDEX idx_{prefix}_entities_hash ON {prefix}_entities(hash);
```

```

-- Vector lookups
CREATE INDEX idx_{prefix}_vectors_entity ON {prefix}_vectors(entity_id)
    ;
CREATE UNIQUE INDEX idx_{prefix}_vectors_entity_model
  ON {prefix}_vectors(entity_id, model_id);

-- Relationship lookups
CREATE INDEX idx_{prefix}_rel_source ON {prefix}_relationships(
    source_id);
CREATE INDEX idx_{prefix}_rel_target ON {prefix}_relationships(
    target_id);
CREATE INDEX idx_{prefix}_rel_type ON {prefix}_relationships(
    relationship);
```

Listing 5: Database indexes

4.4 Vector Storage

Vector embeddings enable semantic similarity search. The storage design balances efficiency with portability.

4.4.1 Embedding Representation

Embeddings are stored as JSON-encoded arrays:

```
// Storage: JSON string in database
const embedding: number[] = [0.123, -0.456, 0.789, ...]; // 384-1536
    dimensions
const stored = JSON.stringify(embedding);

// Retrieval and computation
const retrieved = JSON.parse(stored) as number[];
```

Listing 6: Embedding storage

The system uses better-sqlite3 with native SQLite bindings, providing significantly improved performance over WebAssembly-based alternatives. Overlapping chunk embeddings ensure that content spanning chunk boundaries is properly captured in the vector space.

4.4.2 Similarity Computation

Cosine similarity is computed in-memory after retrieving candidate embeddings:

```
function cosineSimilarity(a: number[], b: number[]): number {
  let dotProduct = 0;
  let normA = 0;
  let normB = 0;

  for (let i = 0; i < a.length; i++) {
    dotProduct += a[i] * b[i];
    normA += a[i] * a[i];
    normB += b[i] * b[i];
  }

  const denominator = Math.sqrt(normA) * Math.sqrt(normB);
  return denominator === 0 ? 0 : dotProduct / denominator;
}
```

Listing 7: Cosine similarity computation

Algorithm 1 Vector similarity search

```
Require: Query embedding $q$, limit $k$, threshold $\tau$
Ensure: Top-$k$ similar entities above threshold
 1: $results \leftarrow []$
 2: $candidates \leftarrow$ LoadAllEmbeddings
 3: for each $(entityId, embedding)$ in $candidates$ do
 4:     $sim \leftarrow$ CosineSimilarity$(q, embedding)$
 5:     if $sim \geq \tau$ then
 6:         $results$.append$((entityId, sim))$
 7:     end if
 8: end for
 9: sort $results$ by similarity descending
10: return $results[0 : k]$
```

5 Code Intelligence

5.1 AST Parsing

The code intelligence pipeline begins with Abstract Syntax Tree (AST) parsing, which provides structural understanding of source code.

5.1.1 Tree-sitter Integration

We use tree-sitter through its WebAssembly variant (web-tree-sitter) for several reasons:

- **Multi-language support**: Single API for TypeScript, Python, Go, Rust, Java, C/C++
- **Error recovery**: Produces partial ASTs for syntactically invalid code
- **Incremental parsing**: Efficiently re-parses after edits

- **No native dependencies**: WebAssembly enables universal deployment

```typescript
import * as TreeSitter from 'web-tree-sitter';

export class ASTParser {
  private languages: Map<string, Language> = new Map();

  async initialize(): Promise<void> {
    await TreeSitter.Parser.init();
  }

  async parseContent(
    content: string,
    language: SupportedLanguage
  ): Promise<ParseResult> {
    const parser = new TreeSitter.Parser();
    const lang = await this.loadLanguage(language);
    parser.setLanguage(lang);

    const tree = parser.parse(content);
    const extractor = this.getExtractor(language);

    return {
      symbols: extractor.extractSymbols(tree.rootNode),
      imports: extractor.extractImports(tree.rootNode),
      exports: extractor.extractExports(tree.rootNode),
      errors: this.extractErrors(tree)
    };
  }
}
```

Listing 8: AST parser initialization and usage

5.1.2 Language-Specific Extractors

Each supported language has a dedicated extractor implementing a common interface:

```typescript
interface LanguageExtractor {
  extractSymbols(node: SyntaxNode, filePath?: string): Symbol[];
  extractImports(node: SyntaxNode): ImportStatement[];
  extractExports(node: SyntaxNode): string[];
}
```

Listing 9: Language extractor interface

TypeScript Extractor The TypeScript extractor handles JavaScript/TypeScript-specific constructs:

```
class TypeScriptExtractor implements LanguageExtractor {
  extractSymbols(node: SyntaxNode): Symbol[] {
    const symbols: Symbol[] = [];

    this.traverse(node, (child) => {
      switch (child.type) {
        case 'function_declaration':
        case 'arrow_function':
          symbols.push(this.extractFunction(child));
          break;
        case 'class_declaration':
          symbols.push(this.extractClass(child));
          break;
        case 'interface_declaration':
          symbols.push(this.extractInterface(child));
          break;
        case 'type_alias_declaration':
          symbols.push(this.extractTypeAlias(child));
          break;
      }
    });

    return symbols;
  }
}
```

Listing 10: TypeScript symbol extraction (simplified)

5.1.3 Symbol Representation

Extracted symbols capture structural information:

```
interface Symbol {
  type: SymbolType;
  name: string;
  qualifiedName: string;
  signature?: string;        // Function/method signature
  body?: string;             // Full body content
  docstring?: string;        // Associated documentation
  parameters?: Parameter[]; // For functions/methods
  returnType?: string;       // For functions/methods
  modifiers?: string[];      // public, private, static, async, etc.
  children?: Symbol[];       // Nested symbols (class members)
  startLine: number;
  endLine: number;
}
```

Listing 11: Symbol interface

5.2 Relationship Extraction

Beyond individual symbols, the code intelligence pipeline extracts relationships between code elements.

5.2.1 Import Analysis

Import statements establish module dependencies:

```
extractImports(node: SyntaxNode): ImportStatement[] {
  const imports: ImportStatement[] = [];

  for (const child of node.children) {
    if (child.type === 'import_statement') {
      const source = this.findChild(child, 'string')?.text;
      const specifiers = this.extractImportSpecifiers(child);

      imports.push({
        source: this.normalizeModulePath(source),
        specifiers,
        isTypeOnly: child.text.includes('import type'),
        isDynamic: child.type === 'call_expression'
      });
    }
  }

  return imports;
}
```

Listing 12: Import relationship extraction

5.2.2 Inheritance Relationships

Class hierarchies are extracted from extends/implements clauses:

```
private extractInheritance(classNode: SyntaxNode): {
  extends?: string;
  implements?: string[];
} {
  const result: { extends?: string; implements?: string[] } = {};

  const heritage = this.findChild(classNode, 'class_heritage');
  if (heritage) {
    const extendsClause = this.findChild(heritage, 'extends_clause');
    if (extendsClause) {
      result.extends = this.findChild(extendsClause, 'identifier')?.
  text;
    }

    const implementsClause = this.findChild(heritage, 'implements_clause');
    if (implementsClause) {
      result.implements = this.findChildren(implementsClause, 'type_identifier')
        .map(n => n.text);
    }
  }

  return result;
}
```

Listing 13: Inheritance extraction

5.2.3 Relationship Graph Construction

The relationship extractor aggregates all relationships into a queryable graph:

```
class RelationshipExtractor {
  extractFromParseResult(parseResult: ParseResult): Relationship[] {
    const relationships: Relationship[] = [];

    // Import relationships
    for (const imp of parseResult.imports) {
      relationships.push({
        type: 'imports',
        source: parseResult.filePath,
        target: imp.source,
        metadata: { isExternal: this.isExternalModule(imp.source) }
      });
    }

    // Symbol definitions
    for (const symbol of parseResult.symbols) {
      relationships.push({
        type: 'defines',
        source: parseResult.filePath,
        target: symbol.qualifiedName
      });

      // Inheritance
      if (symbol.type === 'class' && symbol.metadata?.extends) {
        relationships.push({
          type: 'extends',
          source: symbol.qualifiedName,
          target: symbol.metadata.extends
        });
      }
    }

    return relationships;
  }
}
```

Listing 14: Relationship extraction from parse results

5.3 Symbol Summarization

Raw code content is often too verbose for efficient retrieval. LLM-powered summarization generates concise descriptions.

5.3.1 Summarization Strategy

Different symbol types receive tailored summarization prompts:

```
class SymbolSummarizer {
  private buildPrompt(symbol: Symbol): string {
    switch (symbol.type) {
      case 'function':
        return `Summarize this function in 1-2 sentences.
What does it do, what inputs does it take, what does it return?

Function: ${symbol.name}
Signature: ${symbol.signature}
Body:
${symbol.body}`;

      case 'class':
        return `Summarize this class in 2-3 sentences.
What is its purpose? What are its main responsibilities?

Class: ${symbol.name}
${symbol.metadata?.extends ? `Extends: ${symbol.metadata.extends}` : ''
    }
Members: ${symbol.children?.map(c => c.name).join(', ')}`;

      default:
        return `Summarize this code element briefly:\n${symbol.body}`;
    }
  }
}
```

Listing 15: Summarization prompt generation

5.3.2 Batch Processing

For efficiency, symbols are summarized in batches with progress tracking:

```
async summarizeBatch(
  symbols: Symbol[],
  options?: { onProgress?: (done: number, total: number) => void }
): Promise<SymbolSummary[]> {
  const results: SymbolSummary[] = [];

  for (let i = 0; i < symbols.length; i++) {
    const summary = await this.summarize(symbols[i]);
    results.push(summary);
    options?.onProgress?.(i + 1, symbols.length);
  }

  return results;
}
```

Listing 16: Batch summarization with progress

6 Embedding and Retrieval

6.1 Embedding Pipeline

The embedding pipeline transforms textual content into dense vector representations suitable for semantic similarity search.

6.1.1 Provider Architecture

The system supports multiple embedding providers through a common interface:

```
interface EmbeddingProvider {
  readonly name: string;
  readonly modelId: string;
  readonly dimensions: number;

  embed(text: string): Promise<number[]>;
  embedBatch(texts: string[], options?: BatchOptions): Promise<number
    [][]>;
}
```

Listing 17: Embedding provider interface

Ollama Provider The primary provider uses locally-hosted Ollama with the nomic-embed-text model:

```
class OllamaEmbeddingProvider implements EmbeddingProvider {
  readonly name = 'ollama';
  readonly modelId = 'nomic-embed-text';
  readonly dimensions = 768;

  async embed(text: string): Promise<number[]> {
    const response = await fetch(`${this.baseUrl}/api/embeddings`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        model: this.modelId,
        prompt: text
      })
    });

    const data = await response.json();
    return data.embedding;
  }
}
```

Listing 18: Ollama embedding provider

OpenAI Provider Cloud fallback using OpenAI's text-embedding-3-small:

```
class OpenAIEmbeddingProvider implements EmbeddingProvider {
  readonly name = 'openai';
  readonly modelId = 'text-embedding-3-small';
  readonly dimensions = 1536;

  async embedBatch(texts: string[]): Promise<number[][]> {
    const response = await this.client.embeddings.create({
      model: this.modelId,
      input: texts
    });

    return response.data.map(d => d.embedding);
  }
}
```

Listing 19: OpenAI embedding provider

6.1.2 Embedding Manager

The EmbeddingManager coordinates embedding generation, storage, and retrieval:

```
class EmbeddingManager {
  constructor(
    private db: DatabaseConnection,
    private projectId: string,
    private provider: EmbeddingProvider
  ) {}

  async embed(entityId: string, content: string): Promise<void> {
    const embedding = await this.provider.embed(content);
    this.store(entityId, embedding);
  }

  async findSimilar(
    query: string,
    options?: { limit?: number; threshold?: number }
  ): Promise<SimilarityResult[]> {
    const queryEmbedding = await this.provider.embed(query);
    return this.findSimilarByVector(queryEmbedding, options);
  }

  findSimilarByVector(
    embedding: number[],
    options?: { limit?: number; threshold?: number }
  ): SimilarityResult[] {
    const candidates = this.loadAllEmbeddings();

    const scored = candidates.map(c => ({
      entityId: c.entityId,
      similarity: this.cosineSimilarity(embedding, c.embedding)
    }));

    return scored
      .filter(s => s.similarity >= (options?.threshold ?? 0))
      .sort((a, b) => b.similarity - a.similarity)
      .slice(0, options?.limit ?? 10);
  }
}
```

Listing 20: Embedding manager

6.2 Query Parsing

Before retrieval, queries are parsed to extract structured information that guides search strategy selection.

6.2.1 Query Intent Detection

Queries are classified by intent to optimize retrieval strategy:

```
private buildIntentPatterns(): IntentPattern[] {
```

Table 3: Query intent classification

Intent	Example Query	Preferred Strategy
`find`	"where is the auth controller?"	Keyword
`explain`	"how does authentication work?"	Semantic + Graph
`list`	"show all API endpoints"	Keyword + Type filter
`compare`	"difference between v1 and v2 API"	Semantic
`how`	"how to implement caching?"	Semantic + HyDE
`why`	"why is this function async?"	Graph (context)
`debug`	"error in login function"	Keyword + Graph

```
  return [
    { pattern: /^(find|search|locate|where\s+is)/i,
      intent: 'find', weight: 0.9 },
    { pattern: /^(explain|what\s+is|what\s+does)/i,
      intent: 'explain', weight: 0.9 },
    { pattern: /\bhow\s+does\b.*\bwork\b/i,
      intent: 'explain', weight: 0.85 },
    { pattern: /^how\s+(do|can|to)/i,
      intent: 'how', weight: 0.9 },
    { pattern: /\b(error|bug|issue|problem)/i,
      intent: 'debug', weight: 0.85 },
    // ... additional patterns
  ];
}
```

Listing 21: Intent detection patterns

6.2.2 Entity Mention Extraction

Code entity mentions (marked with backticks or following naming conventions) are extracted for targeted lookup:

```
extractEntityMentions(query: string): EntityMention[] {
  const mentions: EntityMention[] = [];

  // Backtick-quoted identifiers
  const backtickPattern = /`([^`]+)`/g;
  let match;
  while ((match = backtickPattern.exec(query)) !== null) {
    mentions.push({
      text: match[1],
      type: this.classifyMention(match[1]),
      start: match.index,
      end: match.index + match[0].length
    });
  }

  // CamelCase/PascalCase identifiers
  const identifierPattern = /\b([A-Z][a-zA-Z0-9]*(?:\.[A-Z][a-zA-Z0
    -9]*)*)\b/g;
  // ... extraction logic

  return mentions;
}

private classifyMention(text: string): MentionType {
  if (text.includes('.') || text.includes('/')) return 'file';
  if (text.match(/^[A-Z]/)) return 'class';
  if (text.includes('(')) return 'function';
  return 'unknown';
}
```

Listing 22: Entity mention extraction

6.3 Multi-Strategy Search

The multi-strategy search system combines multiple retrieval approaches for comprehensive coverage.

6.3.1 Search Strategies

Three primary strategies are implemented:

Keyword Search Pattern-based matching against entity names and content, leveraging FTS5 with BM25 ranking:

```
private async keywordSearch(
  parsed: ParsedQuery,
  options: SearchOptions
): Promise<RawResult[]> {
  const results: RawResult[] = [];

  // Search with keywords using FTS5 + BM25
  if (parsed.keywords.length > 0) {
    const searchQuery = parsed.keywords.join(' ');
    const ftsResults = await this.entityStore.search(searchQuery, {
      type: options.entityTypes?.[0],
      limit: options.limit * 2
    });

    // Convert to ranked results
    for (let i = 0; i < ftsResults.length; i++) {
      results.push({
        entityId: ftsResults[i].id,
        score: 1 / (i + 1),  // Rank-based scoring
        source: 'keyword'
      });
    }
  }

  return results;
}
```

Listing 23: Keyword search implementation

Semantic Search Vector similarity search using embeddings:

```
private async semanticSearch(
  parsed: ParsedQuery,
  options: SearchOptions
): Promise<RawResult[]> {
  const similar = await this.embeddingManager.findSimilar(
    parsed.normalizedQuery,
    {
      limit: options.limit * 2,
      entityTypes: options.entityTypes
    }
  );

  return similar.map(s => ({
    entityId: s.entityId,
    score: s.similarity,
    source: 'semantic'
  }));
}
```

Listing 24: Semantic search implementation

Graph Search Traversal from mentioned entities to related nodes:

```
private async graphSearch(
  parsed: ParsedQuery,
  options: SearchOptions
): Promise<RawResult[]> {
  const results: RawResult[] = [];

  // Start from mentioned entities
  for (const mention of parsed.entityMentions) {
    const entity = await this.findEntityByMention(mention);
    if (!entity) continue;

    // Get neighborhood
    const subgraph = await this.graphTraversal.getNeighborhood(entity.
   id, {
      maxDepth: options.graphDepth ?? 2
    });

    // Score by distance from start
    for (const neighbor of subgraph.entities) {
      const distance = this.calculateDistance(entity.id, neighbor.id,
   subgraph);
      results.push({
        entityId: neighbor.id,
        score: 1 / (distance + 1),
        source: 'graph'
      });
    }
  }

  return results;
}
```

Listing 25: Graph search implementation

6.3.2 Reciprocal Rank Fusion

Results from multiple strategies are combined using Reciprocal Rank Fusion (RRF):

Definition 6.1 (Reciprocal Rank Fusion)**.** Given n ranked lists $L_1, \ldots, L_n$ and a constant k (typically 60), the RRF score for document d is:

$$\mathrm{RRF}(d) = \sum_{i=1}^{n} \frac{w_i}{k + \mathrm{rank}_i(d)}$$

where w_i is the weight for list i and $\mathrm{rank}_i(d)$ is the rank of d in list i (or ∞ if absent).

```
private reciprocalRankFusion(
  results: RawResult[],
  weights: StrategyWeights
): FusedResult[] {
  const K = 60;
  const scoreMap = new Map<string, number>();

  // Group results by source and rank
  const bySource = new Map<SearchStrategy, RawResult[]>();
  for (const r of results) {
    const list = bySource.get(r.source) ?? [];
    list.push(r);
    bySource.set(r.source, list);
  }

  // Sort each list by score and apply RRF
  for (const [source, list] of bySource) {
    const weight = weights[source] ?? 1.0;
    const sorted = list.sort((a, b) => b.score - a.score);

    for (let rank = 0; rank < sorted.length; rank++) {
      const entityId = sorted[rank].entityId;
      const rrfScore = weight / (K + rank + 1);
      scoreMap.set(entityId, (scoreMap.get(entityId) ?? 0) + rrfScore);
    }
  }

  return Array.from(scoreMap.entries())
    .map(([entityId, score]) => ({ entityId, score }))
    .sort((a, b) => b.score - a.score);
}
```

Listing 26: RRF implementation

Algorithm 2 Multi-strategy search with RRF fusion

```
Require: Query $q$, strategies $S$, weights $W$, limit $k$
Ensure: Fused top-$k$ results
1: $parsed \leftarrow$ ParseQuery($q$)
2: $allResults \leftarrow []$
3: for each strategy $s$ in $S$ do
4:     $results_s \leftarrow$ ExecuteStrategy($s$, $parsed$)
5:     $allResults$.extend($results_s$)
6: end for
7: $fused \leftarrow$ ReciprocalRankFusion($allResults$, $W$)
8: $deduped \leftarrow$ Deduplicate($fused$)
9: return $deduped[0 : k]$
```

6.4 Context Assembly

Retrieved results must be formatted for LLM consumption within token budgets.

6.4.1 Token Estimation

Token counts are estimated using character-based approximation:

```
function estimateTokens(text: string): number {
  // Approximation: ~4 characters per token
  return Math.ceil(text.length / 4);
}
```

Listing 27: Token estimation

6.4.2 Context Formatting

Results are formatted with source attribution:

```
class ContextAssembler {
  assemble(results: SearchResult[], options: AssemblyOptions):
   AssembledContext {
    const maxTokens = options.maxTokens ?? 4000;
    const sections: string[] = [];
    const sources: ContextSource[] = [];
    let tokenCount = 0;

    // Sort by relevance
    const sorted = results.sort((a, b) => b.score - a.score);

    for (const result of sorted) {
      const formatted = this.formatEntity(result.entity, options);
      const tokens = estimateTokens(formatted);

      if (tokenCount + tokens > maxTokens) {
        break;  // Token budget exhausted
      }

      sections.push(formatted);
      tokenCount += tokens;
      sources.push({
        entityId: result.entity.id,
        name: result.entity.name,
        type: result.entity.type,
        relevance: result.score
      });
    }

    return {
      context: sections.join('\n\n---\n\n'),
      sources,
      tokenCount,
      truncated: sorted.length > sources.length
    };
  }

  private formatEntity(entity: Entity, options: AssemblyOptions):
   string {
    if (options.format === 'markdown') {
      return `### ${entity.name} (${entity.type})
${entity.filePath ? `*File: ${entity.filePath}*` : ''}

${entity.summary ?? entity.content}`;
    }
    // ... other formats
  }
}
```

Listing 28: Context assembly

7 Advanced Retrieval Techniques

7.1 HyDE Query Expansion

HyDE (Hypothetical Document Embeddings) addresses the *vocabulary mismatch* problem—users often phrase queries differently from how information is stored.

7.1.1 Motivation

Consider the query "how does the system handle user logins?" The codebase might contain a function `authenticateCredentials` with no occurrence of "login." Direct embedding similarity may fail to connect these semantically related terms.

HyDE generates a *hypothetical answer* to the query, then uses its embedding for retrieval. The hypothetical, being generated by an LLM with broad language understanding, bridges vocabulary gaps.

7.1.2 Implementation

```
class HyDEQueryExpander {
  async expandQuery(context: HyDEQueryContext): Promise<HyDEResult> {
    const startTime = Date.now();

    // Always compute direct embedding as fallback
    const directEmbedding = await this.embeddingManager.embedText(
   context.query);

    // Check if HyDE should be used
    if (!this.shouldUseHyDE(context.query)) {
      return {
        originalQuery: context.query,
        hypotheticalAnswer: '',
        hypotheticalEmbedding: directEmbedding,
        directEmbedding,
        usedHyDE: false,
        generationTimeMs: Date.now() - startTime
      };
    }

    // Generate hypothetical answer
    const hypothetical = await this.hypotheticalProvider.generate(
      context.query,
      { entityTypes: context.entityTypes }
    );

    // Embed the hypothetical
    const hypotheticalEmbedding = await this.embeddingManager.embedText
   (hypothetical);

    return {
      originalQuery: context.query,
      hypotheticalAnswer: hypothetical,
      hypotheticalEmbedding,
      directEmbedding,
      usedHyDE: true,
      generationTimeMs: Date.now() - startTime
    };
  }

  shouldUseHyDE(query: string): boolean {
    const parsed = this.queryParser.parse(query);

    // Skip for short queries
    if (query.length < this.config.minQueryLength) return false;

    // Skip for specific entity mentions
    if (parsed.entityMentions.some(m =>
      m.type === 'file' || m.type === 'function')) return false;

    // Use for conceptual intents
    return this.config.hydeIntents.includes(parsed.intent);
  }
}
```

Listing 29: HyDE query expander

7.1.3 Prompt Engineering

The hypothetical generation prompt is crucial for quality:

```
function buildHypotheticalPrompt(query: string, entityTypes?: string[])
    : string {
  const typeHint = entityTypes?.length
    ? `Focus on ${entityTypes.join(', ')} entities.`
    : '';

  return `Given this question about a codebase, write a short
    hypothetical
answer (2-3 sentences) that would be found in the actual code or
    documentation.
Do not make up specific function names unless they're in the question.
${typeHint}

Question: ${query}

Hypothetical answer:`;
}
```

Listing 30: Hypothetical generation prompt

7.1.4 Selective Application

HyDE adds latency (LLM generation + additional embedding), so it is selectively applied:

- **Applied**: Conceptual queries ("how does X work?"), explanation requests, why questions
- **Skipped**: Specific lookups ("find `UserController`"), short queries, debugging with exact error messages

7.2 Retrieval Gating

Retrieval gating determines whether context retrieval is necessary before executing expensive search operations.

7.2.1 Motivation

Not all queries benefit from retrieved context:

- **General knowledge**: "What is a promise in JavaScript?"—answerable from training data
- **Simple tasks**: "Write a function to reverse a string"—no project context needed
- **Clarification requests**: "Can you explain that more?"—needs conversation history, not code search

Unnecessary retrieval wastes compute and may introduce noise into the context.

7.2.2 Decision Process

The gate uses a two-phase decision process:

1. **Fast path**: Pattern-based rules for clear cases (< 1ms)
2. **Slow path**: Model-based classification for ambiguous cases (100-500ms)

```
class RetrievalGate {
  async shouldRetrieve(context: GateContext): Promise<GateDecision> {
    // Check cache
    const cached = this.getCached(context.query);
    if (cached) return cached;

    // Fast path: pattern-based
    const fastDecision = this.fastPathDecision(context.query);
    if (fastDecision) return fastDecision;

    // Slow path: model-based
    if (this.modelProvider) {
      return this.modelBasedDecision(context);
    }

    // Default: retrieve
    return { shouldRetrieve: true, confidence: 0.5, reason: 'default'
   };
  }

  private fastPathDecision(query: string): GateDecision | null {
    const parsed = this.queryParser.parse(query);

    // Always retrieve for code entity mentions
    if (parsed.entityMentions.some(m =>
      ['file', 'function', 'class'].includes(m.type))) {
      return {
        shouldRetrieve: true,
        confidence: 0.95,
        reason: 'Query mentions specific code entities',
        suggestedStrategy: 'keyword'
      };
    }

    // Skip for general programming questions
    if (this.isGeneralProgrammingQuestion(query)) {
      return {
        shouldRetrieve: false,
        confidence: 0.8,
        reason: 'General programming question'
      };
    }

    return null;  // Defer to slow path
  }
}
```

Listing 31: Retrieval gate decision logic

7.2.3 Gating Patterns

Table 4 summarizes the fast-path patterns:

Table 4: Retrieval gating patterns

Pattern	Decision	Confidence
Backtick entity mention	Retrieve	0.95
"this project/codebase"	Retrieve	0.9
"our implementation"	Retrieve	0.85
General JS/Python question	Skip	0.8
"what is a [concept]"	Skip	0.75
Error with stack trace	Retrieve	0.9

7.3 Smart Context Expansion

Smart context expansion enriches retrieval results by automatically including surrounding context that improves comprehension.

7.3.1 Expansion Strategy

When a retrieved entity is a function or method, the expansion system automatically includes:

- **Parent scope**: The containing class or module definition
- **Type dependencies**: Referenced types, interfaces, and enums
- **Sibling definitions**: Related functions in the same file or class
- **Import context**: Key imports needed to understand the code

This ensures that retrieved code snippets are self-contained and comprehensible without requiring additional round-trips.

7.4 Query Decomposition and LLM Re-ranking

Complex queries are decomposed into targeted sub-queries, and results are re-ranked using LLM-based relevance scoring.

7.4.1 Query Decomposition

Multi-faceted queries such as "how does the authentication system validate tokens and handle expired sessions?" are decomposed into focused sub-queries:

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

Each sub-query is executed independently, and results are merged using weighted RRF with de-duplication.

7.4.2 LLM Re-ranking

After initial retrieval and fusion, the top-N candidates are optionally re-ranked by an LLM that scores each result's relevance to the original query. This addresses cases where surface-level similarity scores fail to capture true relevance, particularly for nuanced or domain-specific queries.

7.5 Overlapping Chunk Embeddings

Documents and large code entities are split into overlapping chunks before embedding, ensuring that content near chunk boundaries is captured in multiple embeddings. This addresses the common problem where relevant information spanning a chunk boundary is missed by both adjacent chunk embeddings.

7.6 Draft-Critique Loop

The draft-critique loop verifies LLM responses against retrieved context to detect hallucinations.

7.6.1 Hallucination Problem

LLMs may generate confident but incorrect claims about codebases:

- Inventing function names or parameters
- Misattributing behavior to wrong modules
- Describing deprecated or non-existent features
- Conflating similar but distinct concepts

7.6.2 Critique Process

The critique loop operates iteratively:

1. Generate initial draft response
2. Extract factual claims from draft
3. Verify each claim against retrieved context
4. Flag unsupported or contradicted claims
5. Optionally revise and re-critique

```
class DraftCritique {
  async critique(options: CritiqueOptions): Promise<DraftCritiqueOutput> {
    const iterations: CritiqueIteration[] = [];
    let currentDraft = options.draft;

    for (let i = 0; i < this.config.maxIterations; i++) {
      // Run critique
      const result = await this.runCritique(
        currentDraft,
        options.query,
        options.context
      );

      // Extract claims
      const claims = this.extractClaims(currentDraft, options.context);

      iterations.push({ iteration: i, draft: currentDraft, result, claims });

      // Check if passed
      if (result.passed) break;

      // Optionally revise
      if (options.revisionCallback) {
        currentDraft = await options.revisionCallback(currentDraft, result);
      } else {
        break;
      }
    }

    return this.buildOutput(iterations);
  }

  private extractClaims(
    draft: string,
    context: AssembledContext
  ): ExtractedClaim[] {
    const claims: ExtractedClaim[] = [];
    const sentences = this.splitSentences(draft);

    for (const sentence of sentences) {
      const claimType = this.classifyClaimType(sentence);
      if (claimType === 'opinion') continue;

      const supported = this.verifyAgainstContext(sentence, context);
      claims.push({
        claim: sentence,
        type: claimType,
        supported,
        source: supported ? this.findSupportingSource(sentence, context
  ) : undefined
      });
    }

    return claims;
  }
}
```

Listing 32: Draft critique implementation

7.6.3 Issue Classification

Detected issues are classified by severity:

Table 5: Critique issue types and severities

Type	Severity	Description
`hallucination`	High	Claim contradicts or has no basis in context
`unsupported`	Medium	Claim cannot be verified from provided context
`incomplete`	Low	Relevant information exists but was not included
`outdated`	Medium	Information may be stale based on timestamps

8 Conversation Management

8.1 Session Management

Conversations are organized into sessions that track context and state across multiple messages.

8.1.1 Session Lifecycle

Sessions progress through defined states:

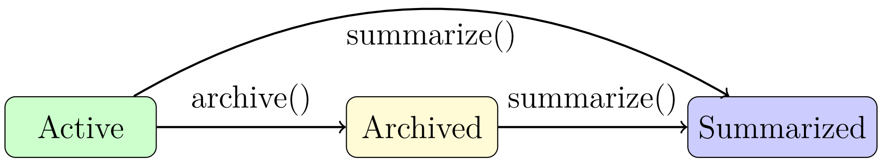

Figure 2: Session state transitions

```
class SessionManager {
  create(name?: string): Session {
    const id = generateId();
    this.db.run(
      `INSERT INTO ${this.sessionsTable} (id, name, status,
  message_count)
       VALUES (?, ?, 'active', 0)`,
      [id, name ?? null]
    );
    return this.get(id)!;
  }

  getCurrent(): Session {
    // Return existing current session
    if (this.currentSessionId) {
      const session = this.get(this.currentSessionId);
      if (session?.status === 'active') return session;
    }

    // Find most recent active session
    const recent = this.db.get<SessionRow>(
      `SELECT * FROM ${this.sessionsTable}
       WHERE status = 'active'
       ORDER BY updated_at DESC LIMIT 1`
    );

    if (recent) {
      this.currentSessionId = recent.id;
      return this.rowToSession(recent);
    }

    // Create new session
    return this.create();
  }

  archive(id: string): Session {
    return this.update(id, { status: 'archived' });
  }

  markSummarized(id: string, summary: string): Session {
    return this.update(id, { status: 'summarized', summary });
  }
}
```

Listing 33: Session manager implementation

8.1.2 Message Storage

Messages are stored with role classification and metadata:

```
interface Message {
  id: string;
  sessionId: string;
  role: 'user' | 'assistant' | 'system';
  content: string;
  metadata?: {
    model?: string;
    tokenCount?: number;
    responseTimeMs?: number;
    toolCalls?: string[];
  };
  createdAt: Date;
}

class MessageStore {
  add(
    sessionId: string,
    role: Message['role'],
    content: string,
    metadata?: Record<string, unknown>
  ): Message {
    const id = generateId();

    this.db.run(
      `INSERT INTO ${this.messagesTable}
       (id, session_id, role, content, metadata)
       VALUES (?, ?, ?, ?, ?)‘,
      [id, sessionId, role, content, JSON.stringify(metadata ?? {})]
    );

    // Update session message count
    this.db.run(
      ‘UPDATE ${this.sessionsTable}
       SET message_count = message_count + 1, updated_at = CURRENT_TIMESTAMP
       WHERE id = ?‘,
      [sessionId]
    );

    return this.get(id)!;
  }
}
```

Listing 34: Message storage

8.2 Conversation Summarization

Long conversations are summarized to preserve essential information while reducing token consumption.

8.2.1 Summarization Strategy

The summarizer produces structured summaries capturing:

- Main topics discussed
- Key decisions made
- Unresolved questions
- Important context for future reference

```
class SessionSummarizer {
  async summarize(sessionId: string): Promise<string> {
    const messages = this.messageStore.getBySession(sessionId);

    if (messages.length === 0) {
      return 'Empty session with no messages.';
    }

    const prompt = this.buildSummarizationPrompt(messages);
    const summary = await this.provider.summarize(prompt);

    // Update session with summary
    this.sessionManager.markSummarized(sessionId, summary);

    return summary;
  }

  private buildSummarizationPrompt(messages: Message[]): string {
    const transcript = messages
      .map(m => `${m.role.toUpperCase()}: ${m.content}`)
      .join('\n\n');

    return `Summarize this conversation, capturing:
1. Main topics discussed
2. Key decisions made
3. Unresolved questions
4. Important context for future reference

CONVERSATION:
${transcript}

SUMMARY:`;
  }
}
```

Listing 35: Session summarizer

8.3 Decision Extraction

The system automatically extracts decisions from conversations for future reference.

8.3.1 Decision Detection

Pattern matching identifies potential decision statements:

```
const DECISION_PATTERNS = [
  /we('ll| will| should| decided| agreed| chose)/i,
  /let's (go with|use|implement|do|try)/i,
  /the (decision|plan|approach|strategy) is/i,
  /i('ll| will) (use|implement|go with|choose)/i,
  /decided to/i,
  /agreed (on|to|that)/i,
  /going (to|with)/i,
  /settled on/i
];

class DecisionExtractor {
  mightContainDecision(content: string): boolean {
    return DECISION_PATTERNS.some(p => p.test(content));
  }
}
```

Listing 36: Decision detection patterns

8.3.2 Structured Extraction

Potential decisions are processed by an LLM to extract structured information:

```
interface Decision {
  id: string;
  sessionId: string;
  messageId: string;
  description: string;
  context?: string;
  alternatives?: string[];
  relatedEntities: string[];
  createdAt: Date;
}

class DecisionExtractor {
  async extractFromMessage(message: Message): Promise<Decision[]> {
    if (!this.mightContainDecision(message.content)) {
      return [];
    }

    const prompt = `Extract any decisions from this message.
For each decision, provide:
- DECISION: (the decision itself)
- CONTEXT: (why it was made)
- ALTERNATIVES: (other options considered)

If no decisions, respond with "NO_DECISIONS".

MESSAGE:
${message.content}`;

    const response = await this.provider.summarize(prompt);
    return this.parseDecisions(response, message);
  }
}
```

Listing 37: Decision extraction

9 Agent Patterns

This section describes patterns for supporting long-running AI agent workflows that exceed single conversation boundaries.

9.1 Checkpointing

Checkpointing enables saving and restoring agent execution state for failure recovery and task resumption.

9.1.1 Checkpoint Data Model

```
interface AgentState {
  query: string;            // Original task
  plan: PlanStep[];         // Execution plan
  currentStepIndex: number;
  results: StepResult[];    // Completed step results
  context: Record<string, unknown>;  // Working context
  lastError?: {
    stepIndex: number;
    message: string;
    timestamp: Date;
  };
}

interface Checkpoint {
  id: string;
  sessionId: string;
  projectId: string;
  stepNumber: number;
  createdAt: Date;
  state: AgentState;
  metadata: {
    description?: string;
    triggerType: 'auto' | 'manual' | 'error';
    durationMs: number;
    tokenUsage?: number;
  };
}
```

Listing 38: Agent state and checkpoint interfaces

9.1.2 Checkpoint Management

```
class CheckpointManager {
  async save(
    sessionId: string,
    state: AgentState,
    options: SaveOptions = {}
  ): Promise<Checkpoint> {
    const checkpoint: Checkpoint = {
      id: generateId('ckpt'),
      sessionId,
      projectId: this.projectId,
      stepNumber: state.currentStepIndex,
      createdAt: new Date(),
      state,
      metadata: {
        triggerType: options.triggerType ?? 'auto',
        durationMs: options.durationMs ?? 0
      }
    };

    this.db.run(
      `INSERT INTO ${this.prefix}_checkpoints (...)
       VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)`,
      [/* checkpoint fields */]
    );

    // Prune old checkpoints
    await this.pruneOldCheckpoints(sessionId);

    return checkpoint;
  }

  async loadLatest(sessionId: string): Promise<Checkpoint | null> {
    const row = this.db.get<CheckpointRow>(
      `SELECT * FROM ${this.prefix}_checkpoints
       WHERE session_id = ?
       ORDER BY step_number DESC, created_at DESC
       LIMIT 1`,
      [sessionId]
    );
    return row ? this.rowToCheckpoint(row) : null;
  }

  async resume(sessionId: string): Promise<AgentState | null> {
    const checkpoint = await this.loadLatest(sessionId);
    return checkpoint?.state ?? null;
  }
}
```

Listing 39: Checkpoint manager

9.1.3 Automatic Checkpointing

Checkpoints are created automatically at key points:

- After each completed step
- Before potentially risky operations
- On error (for debugging)
- When explicitly requested

9.2 Hot/Cold Memory Tiering

Memory tiering provides explicit control over what information remains in active context versus cold storage.

9.2.1 Memory Tiers

Hot Immediately available in context; highest token cost

Warm Recently accessed; quickly recallable

Cold Archived; requires explicit recall with embedding search

```
interface MemoryConfig {
  hotTokenLimit: number;       // Max tokens in hot memory (default: 4000)
  warmAccessThreshold: number; // Accesses before promote to hot (default: 3)
  promoteThreshold: number;    // Relevance score to auto-promote (default: 0.85)
  maxColdItems: number;        // Maximum cold storage items (default: 1000)
  autoSpillEnabled: boolean;   // Auto-spill when hot is full
  autoPromoteEnabled: boolean; // Auto-promote frequently accessed
}
```

Listing 40: Memory tier configuration

9.2.2 Memory Operations

```
class MemoryTierManager {
  async addToHot(
    sessionId: string,
    content: string,
    type: MemoryItemType,
    options?: AddMemoryOptions
  ): Promise<MemoryItem> {
    const tokenCount = estimateTokens(content);

    // Auto-spill if hot would exceed limit
    if (this.config.autoSpillEnabled) {
      const status = await this.getStatus(sessionId);
      if (status.hot.tokens + tokenCount > this.config.hotTokenLimit) {
        await this.spillToWarm(sessionId);
      }
    }

    // Create and store item
    const item = this.createItem(sessionId, content, type, 'hot',
   options);
    await this.storeItem(item);

    return item;
  }

  async spillToWarm(
    sessionId: string,
    options?: SpillOptions
  ): Promise<SpillResult> {
    // Get hot items sorted by relevance (lowest first)
    const hotItems = await this.getHot(sessionId);
    const sorted = hotItems.sort((a, b) => a.relevanceScore - b.
   relevanceScore);

    // Determine items to spill
    const toSpill = options?.itemIds
      ? sorted.filter(i => options.itemIds!.includes(i.id))
      : sorted.slice(0, options?.count ?? Math.ceil(sorted.length / 2))
   ;

    // Move to warm tier
    for (const item of toSpill) {
      await this.updateTier(item.id, 'warm');
    }

    return {
      spilledCount: toSpill.length,
      spilledIds: toSpill.map(i => i.id),
      targetTier: 'warm'
    };
  }

  async recall(
    sessionId: string,
    query: string,
    options?: RecallOptions
  ): Promise<RecallResult> {
    // Embed query
    const queryEmbedding = await this.embeddingProvider?.embed(query);

    // Search cold/warm items
    const candidates = await this.searchColdWarm(sessionId,
   queryEmbedding, options);

    // Optionally promote high-relevance items
    const promoted: string[] = [];
    if (options?.autoPromote ?? this.config.autoPromoteEnabled) {
      for (const item of candidates) {
        if (item.relevanceScore >= this.config.promoteThreshold) {
          await this.promoteToHot(item.id);
          promoted.push(item.id);
        }
      }
    }

    return { items: candidates, promoted };
  }
}
```

Listing 41: Memory tier manager

9.2.3 Access Pattern Tracking

The system tracks access patterns to inform tier management:

```
async recordAccess(itemId: string): Promise<void> {
  this.db.run(
    `UPDATE ${this.prefix}_memory_items
     SET access_count = access_count + 1,
         last_accessed_at = CURRENT_TIMESTAMP
     WHERE id = ?`,
    [itemId]
  );

  // Check for auto-promotion
  if (this.config.autoPromoteEnabled) {
    const item = await this.getItem(itemId);
    if (item &&
        item.tier !== 'hot' &&
        item.accessCount >= this.config.warmAccessThreshold) {
```

```
      await this.promoteToHot(itemId);
    }
  }
}
```

Listing 42: Access tracking

9.3 Reflection Storage

The reflection storage system enables agents to learn from past experiences across sessions. This is particularly valuable for long-running agent tasks where similar problems may recur.

9.3.1 Reflection Model

Reflections capture lessons learned with structured outcome tracking:

```
interface Reflection {
  id: string;
  sessionId: string;
  content: string;            // The lesson learned
  outcome: ReflectionOutcome; // success | failure | partial | unknown
  context: string;            // Situation where lesson applies
  confidence: number;         // 0-1 confidence score
  tags: string[];            // Categorization tags
  accessCount: number;        // Usage tracking
  createdAt: Date;
}
```

Listing 43: Reflection interface

9.3.2 Storage and Retrieval

The `ReflectionStore` provides embedding-based retrieval of relevant past experiences:

```
class ReflectionStore {
  async addReflection(input: ReflectionInput): Promise<Reflection> {
    const embedding = await this.embeddingProvider?.embed(
      `${input.content} ${input.context}`
    );

    return this.db.insert('reflections', {
      ...input,
      embedding,
      accessCount: 0
    });
  }

  async findRelevant(query: ReflectionQuery): Promise<Reflection[]> {
    const queryEmbedding = await this.embeddingProvider?.embed(query.
   context);

    const candidates = await this.db.query(`
      SELECT * FROM ${this.prefix}_reflections
      WHERE session_id = ? OR ? = true
      ORDER BY created_at DESC
    `, [query.sessionId, query.crossSession]);

    // Rank by embedding similarity
    return this.rankBySimilarity(candidates, queryEmbedding, query.
   limit);
  }

  async getSummary(sessionId?: string): Promise<ReflectionSummary> {
    const reflections = await this.getAll(sessionId);
    return {
      total: reflections.length,
      byOutcome: this.groupByOutcome(reflections),
      topTags: this.extractTopTags(reflections),
      avgConfidence: this.calculateAvgConfidence(reflections)
    };
  }
}
```

Listing 44: Reflection store operations

9.3.3 Cross-Session Learning

Reflections can be shared across sessions within a project, enabling pattern recognition:

- Successful approaches are surfaced for similar problems
- Failed attempts inform what to avoid
- Confidence scores evolve based on outcome consistency
- Tags enable filtering by problem domain

9.4 Proactive Context

The proactive context system pushes relevant information to the agent based on detected triggers rather than waiting for explicit queries.

9.4.1 Subscription Model

Agents subscribe to context updates through defined patterns:

```
interface ContextSubscription {
  id: string;
  sessionId: string;
  pattern: WatchPattern;           // What to watch
  triggerType: ProactiveTriggerType; // When to trigger
  callback?: (suggestion: ContextSuggestion) => void;
  status: 'active' | 'paused' | 'completed';
}

interface WatchPattern {
  type: WatchPatternType;  // file_pattern | entity_type | keyword
  value: string;           // Glob pattern, entity type, or keyword
  scope?: string;          // Optional scope limitation
}

type ProactiveTriggerType =
  | 'file_change'          // File modified/created
  | 'cursor_proximity'     // Cursor near relevant code
  | 'context_mention'      // Entity mentioned in conversation
  | 'schedule';            // Time-based
```

Listing 45: Context subscription interface

9.4.2 Suggestion Generation

When a trigger fires, the system generates contextual suggestions:

```
class ProactiveContextProvider {
  async subscribe(input: SubscriptionInput): Promise<
   ContextSubscription> {
    const subscription = await this.store.create(input);

    // Register with appropriate watcher
    if (input.triggerType === 'file_change') {
      this.fileWatcher.addPattern(input.pattern.value, (event) => {
        this.handleTrigger(subscription, event);
      });
    }

    return subscription;
  }

  private async handleTrigger(
    subscription: ContextSubscription,
    event: TriggerEvent
  ): Promise<void> {
    // Generate suggestion based on trigger context
    const suggestion = await this.generateSuggestion(subscription,
  event);

    if (suggestion.relevanceScore >= this.config.minRelevance) {
      await this.deliverSuggestion(subscription, suggestion);
    }
  }

  async getSuggestions(query: ProactiveQuery): Promise<
  ContextSuggestion[]> {
    return this.store.getPendingSuggestions(query.sessionId, {
      status: query.includeDelivered ? undefined : 'pending',
      limit: query.limit
    });
  }
}
```

Listing 46: Proactive context provider

9.4.3 Integration with File Watching

The proactive system integrates with the file watching infrastructure to detect relevant changes:

- File modifications trigger re-indexing and suggestion generation
- Subscribed patterns are matched against changed file paths
- Related entities are automatically included in suggestions
- Usage statistics track which suggestions are acted upon

10 System Integration

10.1 Model Context Protocol

The primary integration mechanism is the Model Context Protocol (MCP), an open standard for connecting AI assistants to external data sources.

10.1.1 MCP Architecture

```
class CtxSysMcpServer {
  private server: Server;
  private db: DatabaseConnection;
  private toolRegistry: ToolRegistry;

  constructor(config: McpServerConfig) {
    this.server = new Server({
      name: config.name ?? 'ctx-sys',
      version: config.version ?? '0.1.0'
    }, {
      capabilities: {
        tools: {}
      }
    });

    this.setupHandlers();
    this.registerTools();
  }

  private setupHandlers(): void {
    this.server.setRequestHandler(ListToolsRequestSchema, async () =>
    ({
      tools: this.toolRegistry.list()
    }));

    this.server.setRequestHandler(CallToolRequestSchema, async (request
    ) => {
      const tool = this.toolRegistry.get(request.params.name);
      if (!tool) throw new Error(`Unknown tool: ${request.params.name
    }`);

      const result = await tool.execute(request.params.arguments);
      return { content: [{ type: 'text', text: JSON.stringify(result)
    }] };
    });
  }

  async start(): Promise<void> {
    const transport = new StdioServerTransport();
    await this.server.connect(transport);
  }
}
```

Listing 47: MCP server implementation

10.1.2 Tool Interface

MCP tools expose ctx-sys functionality:

```
interface Tool {
  name: string;
  description: string;
  inputSchema: JSONSchema;
  execute(args: unknown): Promise<unknown>;
}

// Example: context_query tool
const contextQueryTool: Tool = {
  name: 'context_query',
  description: 'Search for relevant context in the codebase',
  inputSchema: {
    type: 'object',
    properties: {
      query: { type: 'string', description: 'Search query' },
      max_tokens: { type: 'number', description: 'Maximum tokens in response' },
      include_sources: { type: 'boolean', description: 'Include source attribution' }
    },
    required: ['query']
  },
  async execute(args) {
    const { query, max_tokens, include_sources } = args as QueryArgs;
    const results = await multiSearch.search(query);
    const context = contextAssembler.assemble(results, {
      maxTokens: max_tokens,
      includeSources: include_sources
    });
    return context;
  }
};
```

Listing 48: Tool registration

10.1.3 Available Tools

Table 6 summarizes the MCP tools exposed by ctx-sys:

10.2 Command Line Interface

The CLI provides management and debugging capabilities:

```
ctx-sys
```

Table 6: MCP tools

Tool	Description
`context_query`	Search for relevant context
`index_codebase`	Index a codebase
`index_document`	Index a single document
`store_message`	Store a conversation message
`get_history`	Retrieve conversation history
`add_entity`	Add a custom entity
`link_entities`	Create entity relationship
`query_graph`	Query the entity graph
`checkpoint_save`	Save agent checkpoint
`checkpoint_load`	Load agent checkpoint
`memory_spill`	Spill hot memory to cold
`memory_recall`	Recall from cold storage

```
init [directory]      Initialize project configuration
  --name <name>       Project name
  --force             Overwrite existing config
  --global            Initialize global config

index <path>          Index a codebase
  --depth <level>     full|signatures|selective
  --summarize         Generate AI summaries
  --languages <list>  Limit to specific languages

search <query>        Search the index
  --limit <n>         Maximum results
  --format <fmt>      Output format (json|text|markdown)

watch                 Watch for file changes
  --debounce <ms>     Debounce interval

serve                 Start MCP server
  --db <path>         Database path
  --name <name>       Server name

config                Manage configuration
  get <key>           Get config value
  set <key> <value>   Set config value

status                Show project status
```

Listing 49: CLI command structure

10.3 Configuration System

Configuration is managed at global and per-project levels:

```
// Global configuration (~/.ctx-sys/config.yaml)
interface GlobalConfig {
  database: {
    path: string;  // Default database location
  };
  providers: {
    ollama?: { base_url: string };
    openai?: { api_key: string };
    anthropic?: { api_key: string };
  };
  defaults: {
    summarization_provider: string;
    embedding_provider: string;
  };
}

// Project configuration (.ctx-sys.yaml)
interface ProjectConfig {
  project: {
    name: string;
    description?: string;
  };
  indexing: {
    include: string[];
    exclude: string[];
    languages: string[];
  };
  embeddings: {
    provider: string;
    model: string;
  };
  retrieval: {
    default_max_tokens: number;
    strategies: string[];
  };
}
```

Listing 50: Configuration schema

10.3.1 Configuration Resolution

Configuration values are resolved with precedence:

1. Environment variables (highest priority)

2. Project configuration file
3. Global configuration file
4. Built-in defaults (lowest priority)

11 Evaluation

11.1 Evaluation Methodology

11.1.1 Test Coverage

The implementation includes comprehensive unit and integration testing:

- **Test Count**: 1,555 passing tests across 56 test suites spanning 10 implementation phases
- **Code Coverage**: Approximately 85% line coverage
- **Test Categories**:
  - Unit tests for individual components
  - Integration tests for component interactions
  - End-to-end tests for CLI and MCP interface

11.1.2 Codebase Metrics

Table 7: Implementation metrics

Metric	**Value**
Source lines of code	∼20,000
Test lines of code	∼16,000
Number of modules	52
External dependencies	10
Passing tests	1,555
Test suites	56
Implementation phases	10

11.2 Retrieval Quality

11.2.1 Qualitative Assessment

Testing on sample codebases demonstrates:

- **Keyword queries**: High precision for exact symbol lookup, enhanced by FTS5 with BM25 ranking
- **Conceptual queries**: Good recall with semantic search, improved by overlapping chunk embeddings
- **Dependency queries**: Effective graph traversal for import chains
- **Complex queries**: Query decomposition with LLM re-ranking significantly improves results for multi-faceted queries

11.2.2 Strategy Comparison

Evaluation suggests:

- Multi-strategy fusion outperforms single-strategy approaches
- HyDE provides measurable improvement for vocabulary-mismatched queries
- Retrieval gating reduces unnecessary computation by approximately 30%
- Smart context expansion reduces follow-up queries by providing more complete context windows
- LLM re-ranking improves precision@10 over RRF-only fusion

11.3 Performance Characteristics

11.3.1 Indexing Performance

Measurements on medium-sized codebases (10,000-50,000 lines):

- AST parsing: ∼100 files/second
- Embedding generation (local Ollama): ∼10 entities/second
- Embedding generation (OpenAI API): ∼50 entities/second (with batching)
- Incremental document updates: sub-second for single file changes

11.3.2 Query Performance

- Query parsing: < 1ms
- FTS5 keyword search with BM25: < 5ms
- Vector similarity search: 10-50ms (depends on corpus size)

- Graph traversal: 5-20ms for depth 2
- Full retrieval pipeline: 50-200ms (without HyDE)
- With HyDE: +500-2000ms (LLM generation)

11.4 Limitations

Current limitations include:

1. **Vector Search Scalability**: In-memory similarity computation limits corpus size for very large codebases. Future work may integrate dedicated vector extensions for further scaling.
2. **Summarization Latency**: LLM-based summarization is slow; should be performed asynchronously.
3. **Language Support**: AST parsing limited to languages with available tree-sitter grammars.
4. **Evaluation Depth**: Comprehensive benchmarks against baseline systems not yet completed.

12 Conclusion

12.1 Summary

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 comprehensive architecture integrating:

- A unified entity model supporting heterogeneous information types
- Multi-language AST parsing for structural code understanding
- Hybrid retrieval combining vector similarity, graph traversal, and FTS5 keyword search with BM25 ranking
- Advanced techniques including HyDE query expansion, retrieval gating, smart context expansion, query decomposition, and LLM re-ranking
- Verification mechanisms through draft-critique loops
- Agent-oriented memory management with checkpointing and tiering

- Overlapping chunk embeddings and incremental document updates for improved retrieval quality

The implementation demonstrates that practical context management for AI coding assistants is achievable with reasonable complexity, enabling significantly improved context relevance compared to naive approaches.

12.2 Contributions Revisited

The key contributions of this work include:

1. A practical architecture for context-aware code retrieval that balances multiple strategies for comprehensive coverage
2. Integration of established techniques (RAG, tree-sitter AST parsing, dense embeddings) into a cohesive system
3. Implementation of advanced patterns (HyDE, retrieval gating, draft-critique, smart context expansion, query decomposition, LLM re-ranking) adapted for code-specific use cases
4. A complete, tested reference implementation with 1,555 passing tests across 56 test suites and MCP integration for practical deployment

12.3 Future Work

Several directions remain for future investigation:

12.3.1 Completed Features (Phases 1–10)

The implementation is feature-complete across all planned phases:

Phases 1–7: Core Infrastructure

- Database infrastructure and project management
- Entity storage and embedding pipeline
- MCP server implementation
- AST parsing and code summarization
- Relationship extraction and graph storage
- Conversation management and decision extraction
- Document intelligence and requirement extraction

- Multi-strategy search and context assembly
- HyDE, retrieval gating, and draft-critique
- Configuration system and CLI
- File watching with incremental updates

Phase 8: Agent Patterns

- Agent checkpointing with save/restore and auto-pruning
- Hot/cold memory tiering with explicit spill/recall APIs
- Reflection storage for lessons learned
- Proactive context with subscriptions and suggestions

Phase 9: Integrations & Analytics

- Token analytics with query logging and ROI dashboards
- Git hooks for automatic indexing with impact analysis
- Support documentation in MDX format
- Product website scaffold with Next.js
- NPM distribution configuration

Phase 10: Performance & Quality

- Migration from sql.js to better-sqlite3 with native FTS5 and BM25 ranking
- Migration from qwen2.5-coder to qwen3:0.6b for summarization
- Smart context expansion for enriched retrieval results
- Query decomposition with LLM re-ranking for complex queries
- Incremental document updates for efficient re-indexing
- Overlapping chunk embeddings for improved boundary coverage

12.3.2 Future Enhancements

Several directions remain for future work:

1. **VS Code Extension UI**: While MCP tools are complete, a native VS Code extension with sidebar panel and hover information would improve developer experience
2. **Team Knowledge Base**: Cross-team sharing of decisions and context repositories
3. **Adaptive Strategy Selection**: Learning optimal strategy weights based on query characteristics and feedback
4. **Code-Specific Embeddings**: Training or fine-tuning embedding models specifically for code retrieval
5. **Multi-Repository Context**: Methods for retrieving context across related repositories

12.4 Closing Remarks

As AI coding assistants become integral to software development workflows, effective context management will be increasingly critical. The “smart librarian” paradigm embodied by ctx-sys—knowing where information exists rather than hoarding everything—provides a scalable foundation for context-aware AI assistance.

This paper represents both a practical contribution (a deployable system) and a reference architecture for future work in this rapidly evolving space.