Linköping University | Department of Science and Technology
Master's thesis, 30 ECTS | Datateknik
2025 | LIU-ITN/LITH-EX-A--25/001--SE

Agentic RAG & Knowledge Graphs for Test Scope Analysis

Agentisk RAG & Kunskapsgrafer för Analys av Testomfattning

Berkay Orhan

Supervisor : Not yet chosen (Internal)
Examiner : Not yet chosen

External supervisor : Dimitris Rentas, Ericsson AI Technology Leader

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

The abstract resides in file `Abstract.tex`. Here you should write a short summary of your work.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque in massa suscipit, congue massa in, pharetra lacus. Donec nec felis tempor, suscipit metus molestie, consectetur orci. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Curabitur fermentum, augue non ullamcorper tempus, ex urna suscipit lorem, eu consectetur ligula orci quis ex. Phasellus imperdiet dolor at luctus tempor. Curabitur nisi enim, porta ut gravida nec, feugiat fermentum purus. Donec hendrerit justo metus. In ultrices malesuada erat id scelerisque. Sed sapien nisi, feugiat in ligula vitae, condimentum accumsan nisi. Nunc sit amet est leo. Quisque hendrerit, libero ut viverra aliquet, neque mi vestibulum mauris, a tincidunt nulla lacus vitae nunc. Cras eros ex, tincidunt ac porta et, vulputate ut lectus. Curabitur ultricies faucibus turpis, ac placerat sem sollicitudin at. Ut libero odio, eleifend in urna non, varius imperdiet diam. Aenean lacinia dapibus mauris. Sed posuere imperdiet ipsum a fermentum.

Nulla lobortis enim ac magna rhoncus, nec condimentum erat aliquam. Nullam laoreet interdum lacus, ac rutrum eros dictum vel. Cras lobortis egestas lectus, id varius turpis rhoncus et. Nam vitae auctor ligula, et fermentum turpis. Morbi neque tellus, dignissim a cursus sed, tempus eu sapien. Morbi volutpat convallis mauris, a euismod dui egestas sit amet. Nullam a volutpat mauris. Fusce sed ipsum lectus. In feugiat, velit eu fermentum efficitur, mi ex eleifend ante, eget scelerisque sem turpis nec augue.

Vestibulum posuere nibh ut iaculis semper. Ut diam justo, interdum quis felis ac, posuere fermentum ex. Fusce tincidunt vel nunc non semper. Sed ultrices suscipit dui, vel lacinia lorem euismod quis. Etiam pellentesque vitae sem eu bibendum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Pellentesque scelerisque congue ullamcorper. Sed vehicula sodales velit a scelerisque. Pellentesque dignissim lectus ipsum, quis consectetur tellus rhoncus a.

Nunc placerat ut lectus vel ornare. Sed nec dictum enim. Donec imperdiet, ipsum ut facilisis blandit, lacus nisi maximus ex, sed semper nisl metus eget leo. Nunc efficitur risus ac risus placerat, vel ullamcorper felis interdum. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Duis vitae felis vel nibh sodales fringilla. Donec semper eleifend sem quis ornare. Proin et leo ut dolor consectetur vehicula. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Nunc dignissim interdum orci, sit amet pretium nibh consectetur sagittis. Aenean a eros id risus aliquam placerat nec ut lectus. Curabitur at quam in nisi sodales imperdiet in at erat. Praesent euismod pulvinar imperdiet. Nam auctor mattis nisi in efficitur. Quisque non cursus ipsum, consequat vehicula justo. Fusce varius metus et nulla rutrum scelerisque. Praesent molestie elementum nulla a consequat. In at facilisis nisi, convallis molestie sapien. Cras id ullamcorper purus. Sed at lectus sit amet dolor finibus suscipit vel et purus. Sed odio ipsum, dictum vel justo sit amet, interdum dictum justo. Quisque euismod quam magna, at dignissim eros varius in. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Acknowledgments

Acknowledgments.tex

Contents

List of Figures

List of Tables

1 Introduction

In the domain of large-scale enterprise software development, the velocity of feature delivery is a critical competitive advantage. Modern Continuous Integration and Continuous Deployment (CI/CD) pipelines have enabled rapid iteration cycles, but they simultaneously impose significant demands on quality assurance processes [1].

1.1 Background

A critical bottleneck in modern CI/CD workflows is *test scope analysis*. In this thesis, we define test scope analysis as the activity of identifying and recommending relevant legacy test cases for verification when a new feature, change request, or defect fix is introduced [2]. This differs from traditional Regression Test Selection (RTS), which typically focuses on code-coverage-based filtering of a test suite for automated execution [3, 4]. Instead, this work addresses the *Test Case Recommendation* problem [5]: supporting engineers in the semantic discovery of tests that may not be directly linked via static traces but are contextually relevant to the change. Evaluation of such retrieval systems typically relies on standard information retrieval metrics, such as Precision@k (accuracy of top-k results), Recall@k (proportion of relevant items found), and Mean Average Precision (MAP) [6].

1.2 Motivation

For large organizations, maintaining high test coverage while avoiding redundant execution is a complex optimization problem. When a new artifact, such as a user story or bug ticket, is introduced, test engineers must navigate vast repositories of test cases to identify existing coverage and potential gaps [7]. Historically, this retrieval process has relied on keyword-based search or the manual inspection of static trace links. However, these traditional methods are increasingly brittle; they often fail to capture the semantic nuance of natural language requirements or the complex, graph-like dependencies between system components [8, 9]. As software systems grow in complexity, the "semantic gap" between unstructured change descriptions (e.g., "fix race condition in handover") and structured test cases (e.g., "TC_HO_001") widens, leading to inefficient scoping, missed defects, and increased maintenance costs [10].

Recent advancements in Large Language Models (LLMs), which are AI models capable of understanding and generating human-like text, and Retrieval-Augmented Generation (RAG), a technique that enhances LLMs by fetching relevant information from external data sources [11], offer promising avenues for bridging this semantic gap [12, 13]. Simultaneously, Knowledge Graphs (KGs), structured representations of interconnected entities and their relationships, provide a structured means to model the intricate relationships between requirements, code, and tests [14, 15].

The primary problem addressed in this thesis is the inefficiency and inaccuracy of current test scope analysis methods in large-scale software engineering contexts. Traditional keyword search approaches lack the semantic understanding required to match high-level requirements with low-level test steps, frequently resulting in low recall (missing relevant tests) or low precision (retrieving irrelevant tests).

While standard RAG approaches improve semantic matching, they often suffer from precision errors (retrieving plausible but incorrect context) and "hallucinations" (generative errors where the model invents non-existent facts or test names) [5]. Furthermore, flat RAG retrieval fails to account for the structural context inherent in software artifacts (e.g., inheritance, trace links, execution history). Conversely, purely graph-based methods struggle to effectively process the unstructured free text found in change logs and user stories [16]. Consequently, there is a lack of integrated solutions that leverage the combined strengths of *vector-based semantic retrieval* and *graph-based structural reasoning*, often referred to as *GraphRAG* [17], to provide accurate and explainable test scope recommendations.

However, a static GraphRAG pipeline, where a query is simply converted to a vector, matched against a graph, and summarized, remains insufficient for complex software engineering tasks. Such pipelines fail when the initial user intent is ambiguous or requires multi-step investigation (e.g., "Check if this change affects the billing module, and if so, find tests for the legacy payment gateway"). A static retrieval process cannot decompose this intent, perform intermediate lookups to refine the search criteria, or reason about the absence of information. To address this, an *agentic architecture* is required [18]. An agentic architecture is defined as a system where autonomous software agents perceive their environment, reason about how to achieve a goal, and orchestrate the execution of tools to solve complex tasks. By combining these technologies into a system where autonomous AI agents orchestrate retrieval and reasoning loops, there is potential to transform test scope analysis from a manual, error-prone task into an automated, explainable, and highly accurate process [19].

1.3 Aim

The aim of this thesis is to design and evaluate an agent-orchestrated RAG and Knowledge Graph architecture for test scope analysis. This will involve the implementation of a prototype system to determine the effectiveness of this hybrid approach in automating the retrieval of relevant legacy test cases, identifying coverage gaps, and providing explainable rationales for its recommendations, thereby increasing practical utility for software practitioners, as validated through qualitative feedback from Ericsson engineers in a live or simulated environment.

1.4 Research questions

To fulfill the aim, the following research questions have been formulated:

- **RQ1:** What ontology design is required to model the semantic dependencies between unstructured requirements and structured test artifacts within a Knowledge Graph to enable semantic retrieval?

- **RQ2:** To what extent does an agent-orchestrated GraphRAG approach improve retrieval accuracy compared to standard keyword-based and vector-only RAG methods?
- **RQ3:** To what extent do software practitioners at Ericsson find the automated recommendations accurate and the provided justifications sufficient for decision-making?

1.5 Delimitations

This study is delimited to the context of Ericsson's internal software development environment.

- **Data Scope:** The study will utilize specific datasets provided by Ericsson, including structured test management data (requirements, trace links) and unstructured test case descriptions.
- **Functional Scope:** The system focuses exclusively on *Test Scope Analysis* (identifying relevant existing tests) and does not include the automatic generation of new test cases or the execution of tests.

2 Theoretical Framework

This chapter establishes the theoretical foundation for the agent-orchestrated Retrieval-Augmented Generation (RAG) and Knowledge Graph architecture developed in this thesis. It begins by grounding the problem in the context of modern software development practices and quality assurance, followed by an in-depth exploration of Large Language Models (LLMs), RAG, Knowledge Graphs, and AI Agents, including their constituent components and operational mechanisms.

2.1 Software Development Lifecycle & Quality Assurance

Modern software development is characterized by rapid iteration and continuous delivery, largely facilitated by Continuous Integration and Continuous Deployment (CI/CD) pipelines. These methodologies accelerate feature delivery but place significant demands on quality assurance, particularly in ensuring the reliability of complex systems through effective testing.

Continuous Integration and Continuous Deployment (CI/CD)

CI/CD represents a set of practices that enable rapid and reliable software releases [20]. Continuous Integration involves frequently merging code changes into a central repository, followed by automated builds and tests. Continuous Deployment extends this by automatically deploying verified changes to production. While highly efficient, CI/CD necessitates robust testing strategies to prevent the introduction of defects at an accelerated pace.

Regression Testing and Test Scope Analysis

Central to quality assurance in CI/CD environments is regression testing, which ensures that new code changes do not adversely affect existing functionalities [3]. However, re-running all tests for every small change is resource-intensive and impractical in large-scale systems [4]. This gives rise to the critical challenge of *test scope analysis*: the activity of intelligently determining the minimal set of legacy test cases relevant for verification when a new feature, change request, or defect fix is introduced. Effective test scope analysis balances the

need for high test coverage with the efficiency of execution, preventing redundant tests while identifying potential gaps.

Traceability in Software Engineering

Software traceability refers to the ability to link related artifacts throughout the software development lifecycle, such as requirements to design, design to code, and code to test cases. Robust traceability is fundamental for effective test scope analysis, as it provides the explicit connections necessary to understand the impact of changes and identify corresponding verification activities. However, maintaining accurate and comprehensive traceability in large, evolving systems is a significant challenge.

Observability, Evaluation, and Deployment

Beyond the core development and testing activities, the successful operation of complex software systems, especially those incorporating AI, relies on robust practices for observability, evaluation, and deployment.

- **Observability:** Refers to the ability to infer the internal states of a system by examining its external outputs [21]. In AI-driven systems, this is crucial for understanding how models and agents make decisions, identifying potential biases, and diagnosing issues in real-time.
- **Evaluation:** Encompasses the methods and metrics used to assess a system's performance, accuracy, and overall effectiveness. For test scope analysis systems, this includes quantitative metrics such as precision and recall, as well as qualitative assessments of practical utility and explainability.
- **Deployment:** Is the process of making the developed system available for use, ranging from local integration to large-scale production rollouts. For agentic systems, deployment strategies must consider integration with existing workflows, scalability, and maintenance.

2.2 Large Language Models

Large Language Models (LLMs) form the cognitive backbone of modern AI-driven applications, including agentic systems. These models are advanced neural networks trained on vast datasets of text, enabling them to understand, generate, and process human language with remarkable fluency.

Foundational Principles

At their core, LLMs are built upon the *transformer architecture*, which allows them to process sequences of data in parallel, capturing long-range dependencies more effectively than previous architectures. However, the field is rapidly evolving beyond standard dense transformers. To improve efficiency and scalability, modern architectures often incorporate techniques like *Mixture-of-Experts (MoE)*, where only a subset of parameters ("experts") are activated for a given input. This approach is explicitly utilized in open-weights models like DeepSeek-R1 [22]. Other frontier models, such as Gemini 3 Pro [23], are explicitly built upon a sparse Mixture-of-Experts (MoE) transformer-based architecture, offering native multimodal support for text, vision, and audio inputs. Similarly, Claude 4.5 Sonnet [24] exhibits advanced "hybrid" reasoning and multimodal capabilities, implying significant architectural innovations, though its specific internal structure remains proprietary.

Attention Mechanism

The **Attention Mechanism** [25] is a pivotal component of the transformer architecture, enabling LLMs to weigh the importance of different parts of the input sequence when processing each token. *Self-Attention*, in particular, allows the model to capture dependencies between tokens regardless of their distance in the input, forming a rich contextual representation. This mechanism computes three vectors for each token: a Query (Q), a Key (K), and a Value (V). The attention score for a given Query token is calculated by its dot product with all Key tokens, followed by a softmax function to produce weights, which are then applied to the Value vectors.

Connection to Integration: The Attention Mechanism is the fundamental "engine" that allows the agent to maintain context over long reasoning chains (Section 2.6) and complex interactions with tools (Section 2.4). It enables the agent to effectively recall relevant parts of the conversation history, tool outputs, and retrieved information, making coherent decisions about subsequent actions and query analysis (Section 2.4). This capability is crucial for preventing "Context Rot" (Section 2.6) and ensuring the agent's effectiveness in dynamic environments.

Despite these structural variations, the primary training objective remains *next-token prediction*: given a sequence of input tokens, the model learns to predict the most probable subsequent token. This simple task, when scaled across massive datasets and parameter counts, yields emergent capabilities in reasoning, coding, and general problem-solving [26, 27].

Reasoning and Tool Use Capabilities

Beyond basic text generation, advanced LLMs exhibit significant capabilities relevant to agentic systems [28]:

- **Reasoning:** LLMs can engage in various forms of reasoning, from simple fact retrieval to complex logical deduction. This capability is enhanced through two primary paradigms:
  - **Prompted Reasoning:** Techniques like *Chain-of-Thought (CoT)* prompting guide standard models to break down problems into intermediate steps [29].
  - **Inference-Time Compute:** Recent "reasoning models" (e.g., DeepSeek-R1, OpenAI o1) employ Reinforcement Learning to internalize this "thinking" process. These models generate hidden chains of thought before producing an output, effectively trading increased inference latency for higher accuracy on complex tasks like dependency analysis.
- **Tool Use:** A critical emergent capability is the ability of LLMs to interact with external tools or APIs. By learning to generate structured outputs (e.g., JSON function calls) in response to prompts, LLMs can extend their functionalities beyond their training data, enabling them to perform calculations, query databases, or access real-time information.

2.3 Knowledge Graphs in Software Engineering

Knowledge Graphs (KGs) provide a structured way to represent complex, interconnected data, making them particularly suitable for modeling the dependencies in software systems [30, 14]. Unlike vector stores, which capture semantic similarity, KGs capture explicit structural relationships.

Graph Data Models: RDF vs. LPG

Two primary data models exist for implementing Knowledge Graphs:

- **Resource Description Framework (RDF):** A W3C standard based on triples (Subject, Predicate, Object). RDF is ideal for semantic web applications and interoperability but can be verbose for traversing complex property-rich graphs typical in software engineering.
- **Labeled Property Graphs (LPG):** Used by graph databases like Neo4j, LPGs allow nodes and relationships to have internal properties (key-value pairs). This model is often preferred in industry for software analytics because it allows for efficient storage of metadata (e.g., a "calls" relationship can have properties like `frequency` or `latency`). This thesis utilizes the LPG model to support the rich metadata requirements of test artifacts [17].

LPG Mechanics: Index-Free Adjacency

Central to the performance of Labeled Property Graphs, particularly for traversal-heavy workloads, is the concept of **Index-Free Adjacency**. In this model, each node directly stores pointers to its adjacent nodes and relationships. This contrasts with index-intensive models (like RDF triple stores) where relationships are discovered through lookup tables or global indices. Index-free adjacency means that the cost of traversing a relationship is constant, $O(1)$, regardless of the total size of the graph.

Connection to Integration: This architectural choice makes the `graph_traverse()` tool (Section 2.4) computationally feasible for the agent's multi-hop reasoning. When the agent needs to analyze the "blast radius" of a change by exploring dependencies (e.g., from a modified function to affected tests), index-free adjacency ensures that even deep traversals complete within acceptable timeframes, preventing bottlenecks in the dynamic retrieval process.

Ontologies in Software Engineering

An *ontology* defines the schema or the "mental model" of the domain. In software engineering KGs, the ontology dictates the types of entities (e.g., `Class`, `Method`, `TestCase`, `Requirement`) and the allowed relationships between them (e.g., `inherits_from`, `verifies`, `traces_to`). A well-defined ontology is crucial for traversing the "semantic gap," allowing an agent to reason that if a `Requirement` is changed, the `TestCases` that `verify` it are candidates for regression testing [9, 15].

Graph Retrieval and Traversal

Retrieving information from a Knowledge Graph differs fundamentally from text retrieval. Instead of calculating similarity scores, it involves traversing the graph's structure to discover connections.

- **Pattern Matching (Declarative Querying):** The most common method involves expressing a desired subgraph structure using a query language like Cypher [31]. In test impact analysis, this allows for the precise selection of verification artifacts based on structural dependencies, such as retrieving "all `TestCases` that cover `Functions` called by the modified `Class`."
- **Multi-Hop Traversal:** Complex dependencies often require traversing multiple edges (hops) to find relevant information (e.g., Requirement → Function → Test). Algorithms like Breadth-First Search (BFS) or Depth-First Search (DFS) are used to explore the neighborhood of a starting node up to a specified depth.
- **Graph Algorithms:** Beyond simple traversal, advanced algorithms can be applied for global analysis. *Community Detection* (e.g., Leiden algorithm [32]) clusters densely connected nodes to identify functional software modules. For the agent, detecting these

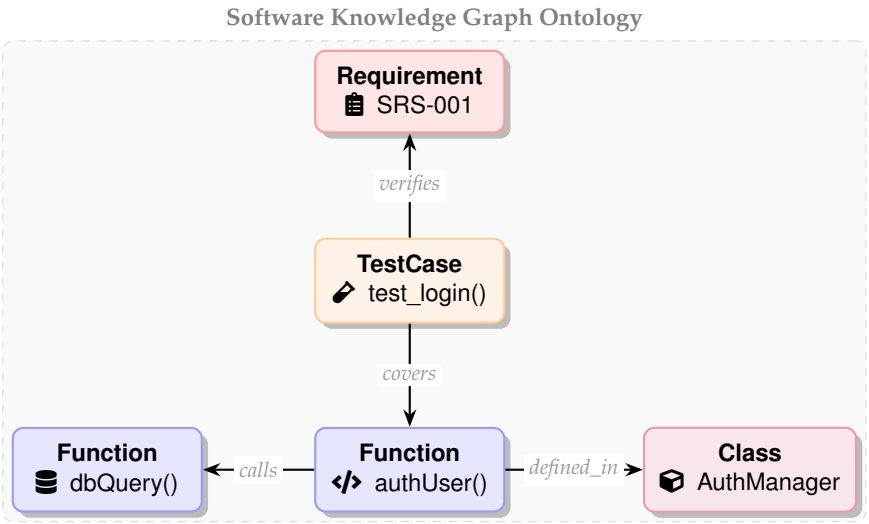


Figure 2.1: A representative Knowledge Graph Ontology for Software Engineering, illustrating the semantic relationships (e.g., `verifies`, `calls`) between Requirements, Test Cases, Functions, and Classes.

communities helps in understanding the broader "blast radius" of a change, suggesting that if one component in a tightly coupled cluster is modified, the entire module's test suite may require execution.

2.4 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is a powerful technique that enhances the capabilities of LLMs by enabling them to fetch and incorporate relevant information from external knowledge sources during the generation process [11]. This mitigates issues like *hallucinations* (generative errors where the model invents facts) and improves precision by grounding the LLM's responses in factual, up-to-date data, although it introduces the risk of retrieval errors (fetching irrelevant context). In the context of test scope analysis, RAG serves as the bridge between the static knowledge contained in software artifacts (code, requirements, tests) and the dynamic reasoning capabilities of the LLM.

Embeddings and Semantic Search

The core mechanism enabling RAG is the concept of *embeddings*. Embeddings are numerical representations (vectors) of text, code, or other data, where semantically similar items are mapped closer to each other in a high-dimensional vector space [33]. This allows for efficient comparison and retrieval of semantically related information, surpassing keyword-based search by capturing intent and context.

Code vs. Natural Language Embeddings

While generic language models excel at natural language understanding, they often **fail to capture the semantic nuance of source code** or specialized software engineering terminology. This is known as the "Semantic Gap" [10]. Failure modes include:

- **Tokenization Issues:** Code identifiers like 'calculateTotalPrice' are often split into 'calculate', 'total', 'price' by natural language tokenizers, losing the original meaning.

- **Structural vs. Sequential Semantics:** Natural language is largely sequential, while code has rich structural relationships (e.g., class hierarchies, function calls) that are not captured by linear text embeddings.
- **Domain-Specific Vocabulary:** Codebases use highly specialized terms (e.g., 'mutex', 'endianness', '0xdeadbeef') that are rare or have different meanings in general text corpora.

Therefore, for effective test scope analysis, the choice of embedding model is critical. This thesis utilizes models like **CodeBERT** [34] or **Qwen** [35], which are pre-trained on vast datasets of source code, enabling them to understand code syntax and semantics more effectively than general-purpose LLMs.

Connection to Integration: *Because* semantic search, even with code-specific embeddings, has these inherent blind spots (e.g., struggling with precise structural queries like "all functions called by class X"), the agent must have the autonomy (Section 2.4) to *bypass* pure vector search in favor of graph traversal when a query indicates structural intent (Section 2.4). This adaptive selection ensures robust retrieval across diverse query types.

For technical domains, generic language models often fail to capture the semantic nuance of code or specialized terminology. Therefore, the choice of embedding model is critical. While seminal work like BERT [36] and Sentence-BERT [37] laid the foundation for contextual embeddings, the field has evolved towards massive general-purpose models. Currently, flagship proprietary models such as OpenAI's `text-embedding-3` [38] and Google's `gemini-embedding-001` [39] offer state-of-the-art performance. However, in sensitive software engineering contexts where data privacy is paramount, high-performing open-weight models are often preferred as they allow for secure, on-premise deployment. Notable examples include Qwen3-Embedding-8B [35] and BGE-M3 [40], which provide comparable retrieval quality while maintaining data sovereignty.

Text Retrieval Strategies

Effective RAG systems rely on robust mechanisms to locate relevant textual information. These strategies are generally categorized by how they represent and match data.

Dense Retrieval (Vector Search)

Dense retrieval operates on the semantic vector space created by embedding models. It calculates similarity metrics, typically *Cosine Similarity* or *Euclidean Distance*, between the query vector and document vectors.

- **Mechanism:** $score(q, d) = \cos(\vec{v}_q, \vec{v}_d) = \frac{\vec{v}_q \cdot \vec{v}_d}{\|\vec{v}_q\| \|\vec{v}_d\|}$
- **Strengths:** Captures semantic meaning, synonyms, and intent (e.g., mapping "login issue" to "authentication failure").
- **Weaknesses:** Struggles with precise keyword matching, rare entities, or specific technical identifiers (e.g., error codes like `0x8004`).

Sparse Retrieval (Keyword Search)

In the context of software repositories, sparse retrieval is indispensable for locating exact string matches. Unlike natural language, source code relies on precise identifiers. A developer searching for a specific error constant (e.g., `ERR_TIMEOUT_503`) or a unique function signature requires an exact lexical match, which dense retrievers often fail to capture due to tokenization or semantic abstraction. The seminal approach to term weighting is TF-IDF (Term Frequency-Inverse Document Frequency) [41], which assigns higher importance

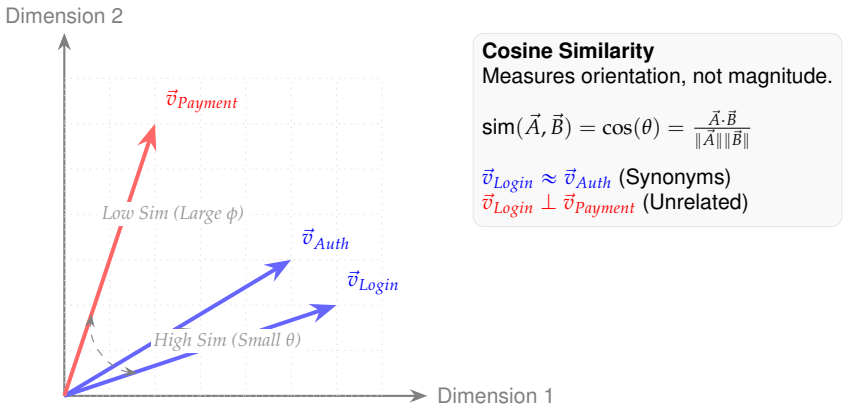


Figure 2.2: Visualizing Embeddings and Cosine Similarity: Semantically similar terms (e.g., Login, Auth) have vectors with small angles θ, while unrelated terms (e.g., Payment) have large angles ϕ.

to terms that appear frequently in a document but rarely across the entire collection. Building upon this, the industry standard algorithm is **BM25** (Best Matching 25), a probabilistic relevance framework that improves upon TF-IDF [42, 43]. Its scoring function for a document d given a query Q (composed of terms $q_1, \ldots, q_n$) is:

$$\text{BM25Score}(d, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, d) \cdot (k_1 + 1)}{f(q_i, d) + k_1 \cdot \left(1 - b + b \cdot \frac{\text{len}(d)}{\text{avgdl}}\right)}$$

where:

- $f(q_i, d)$ is the term frequency of query term q_i in document d.
- $\text{len}(d)$ is the length of document d in words.
- avgdl is the average document length in the collection.
- k_1 is a positive tuning parameter that calibrates term frequency saturation (typically $1.2 \leqslant k_1 \leqslant 2.0$). A higher k_1 means term frequency continues to increase relevance more strongly.
- b is a parameter ($0 \leqslant b \leqslant 1$) that controls document length normalization. If $b = 0$, no length normalization is applied; if $b = 1$, full normalization is applied, penalizing longer documents more heavily. For code, where file lengths vary significantly, appropriate b selection is crucial.

Connection to Integration: BM25 provides the lexical precision tool that the agent exposes as `keyword_search()` (Section 2.4). This tool is critical when the agent detects specific, precise identifiers (e.g., error codes, exact function names) in the user's query (Section 2.4), ensuring that structurally relevant but semantically distant artifacts are retrieved effectively.

Hybrid Search

For effective test scope analysis, the retrieval system must bridge the gap between high-level intent and low-level implementation. Hybrid search [44, 45] addresses this by combining the strengths of semantic search (mapping "user login" to authentication modules) with keyword

search (identifying specific variables involved in a change). By fusing these results, the system ensures that relevant tests are surfaced whether they share semantic concepts or explicit code references with the modified artifacts.

A challenge in combining these disparate results is **Score Fusion**, where combining raw scores from algorithms with different scales (e.g., Cosine similarity is 0-1, BM25 is unbounded) is non-trivial. Therefore, this thesis adopts **Reciprocal Rank Fusion (RRF)** [46] as the fusion mechanism. RRF is a robust, parameter-free method that sorts documents based on their rank rather than raw score:

$$RRFscore(d) = \sum_{r \in R} \frac{1}{k + r(d)}$$

where $r(d)$ is the rank of document d in result set r, and k is a smoothing constant (typically 60). This ensures that documents appearing consistently high in both lists are prioritized, effectively bypassing the challenges of score normalization.

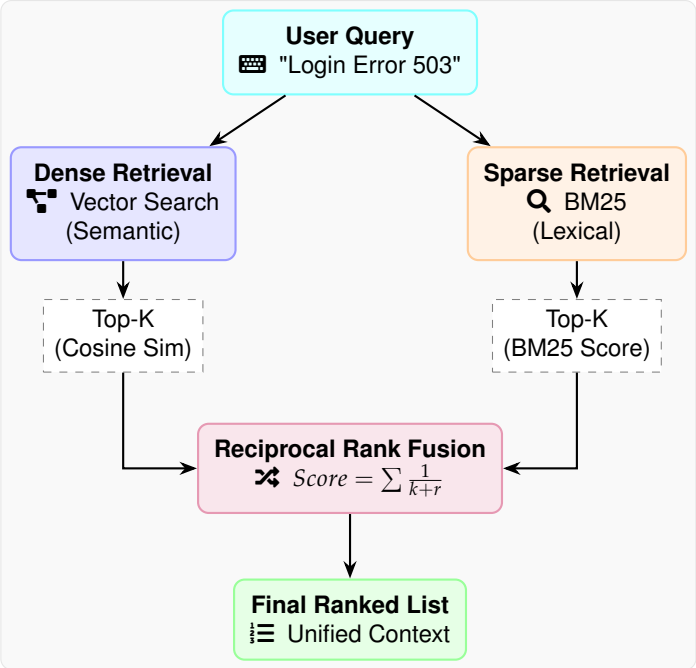


Figure 2.3: The Hybrid Search process: Parallel execution of Dense (Vector) and Sparse (Keyword) retrieval, followed by Reciprocal Rank Fusion (RRF) to merge and rank results.

The RAG Pipeline Components

An effective RAG system for software engineering comprises several specialized components working in concert to ingest, process, and store data:

Document Loaders (Data Integration)

Document loaders are responsible for ingesting data from diverse software development lifecycle (SDLC) sources [47]. In a corporate environment, this requires "connectors" capable of interfacing with:

- **Version Control Systems:** Ingesting raw source code and commit history (e.g., Git).
- **Documentation Platforms:** Parsing structured specifications (e.g., Markdown, PDF, Internal Wikis).

- **Issue Trackers:** Extracting bug reports, user stories, and acceptance criteria (e.g., Jira, Linear).

These loaders normalize disparate data formats into a unified document structure suitable for processing.

However, standard text extraction often fails for visually rich formats like PDF, where information is encoded in layout (e.g., tables, dual-column text) rather than linear character streams. To address this, **Document Layout Analysis (DLA)** is required. DLA involves using computer vision models (e.g., DocLayNet) to detect and classify layout elements (headers, paragraphs, tables) before extraction. This is critical for preserving the "structural context" of requirements and specifications, ensuring that a table cell containing a test parameter is explicitly linked to its column header rather than being flattened into a meaningless string of text.

Text Splitters (Semantic Chunking)

Standard text splitting (e.g., every 500 characters) is detrimental in software contexts where maintaining logical integrity is paramount, especially given the "lost-in-the-middle" phenomenon where LLMs struggle to recall information from the center of long contexts [48]. Instead, *structure-aware splitting* strategies are employed:

- **Code-Aware Splitting:** Parsing the Abstract Syntax Tree (AST) of source code to split based on functional boundaries (classes, methods, functions) rather than arbitrary line counts.
- **Header-Based Splitting:** processing requirements documents by hierarchy (e.g., Section 1.2, 1.2.1), ensuring that the parent context (headers) is preserved with each child chunk.

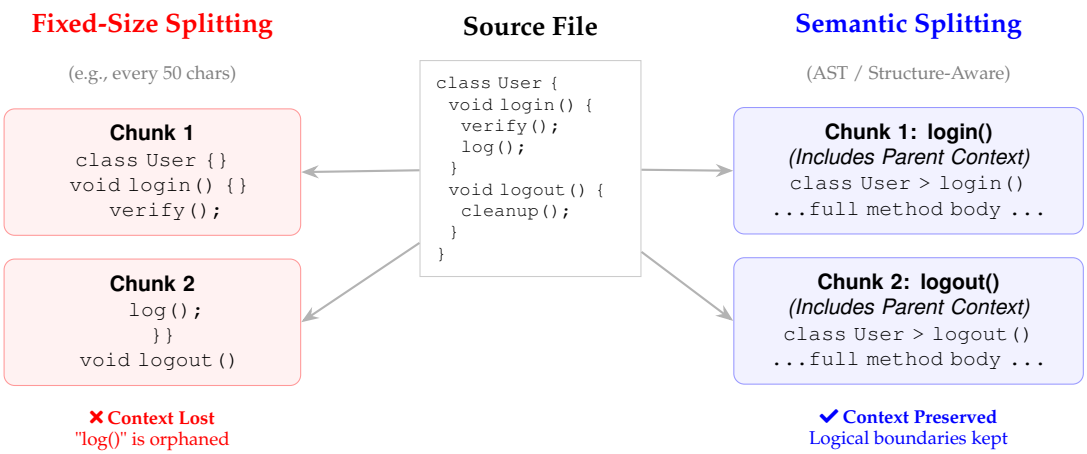


Figure 2.4: Comparison of Fixed-Size vs. Semantic (Structure-Aware) Chunking. Semantic splitting respects code boundaries (classes, methods), preserving logical context that is often lost in arbitrary fixed-size splits.

Knowledge Stores

To effectively map the test scope, the storage layer must capture both the *semantics* (meaning) and the *structure* (dependencies) of the data.

- **Vector Indices:** Store the high-dimensional embeddings, enabling efficient similarity search (e.g., "Find tests related to authentication failures") [49].
- **Knowledge Graphs (KGs):** Explicitly model the relationships between entities (e.g., `Requirement-A` $\xrightarrow{verifies}$ `Test-B` $\xrightarrow{covers}$ `Function-C`) [30].

Vector Search (HNSW)

Efficient retrieval from large vector indices relies on **Approximate Nearest Neighbor (ANN)** algorithms, which sacrifice a small amount of accuracy for significant gains in search speed. This thesis employs **Hierarchical Navigable Small Worlds (HNSW)** graphs [50] for vector search. It is important to note that the "graph" in HNSW refers to a specialized indexing structure used purely for accelerating vector similarity search. This is distinct from the semantic Knowledge Graphs (Section 2.3), which explicitly model relationships between entities. HNSW constructs a multi-layer graph where each layer is a navigable small-world graph. Queries start at the top layer (sparsest graph), quickly navigating to a local optimum, then descending to denser layers to refine the search. This hierarchical approach offers a logarithmic time complexity ($O(\log N)$) for search operations, balancing query speed with recall performance.

Connection to Integration: HNSW is crucial for the `vector_search()` tool (Section 2.4) because it enables sub-second retrieval times over potentially massive code and documentation embeddings. This low-latency performance is a hard constraint for the agent's interactive Thought-Action-Observation loop, ensuring that semantic search steps do not introduce unacceptable delays in the test scope analysis process.

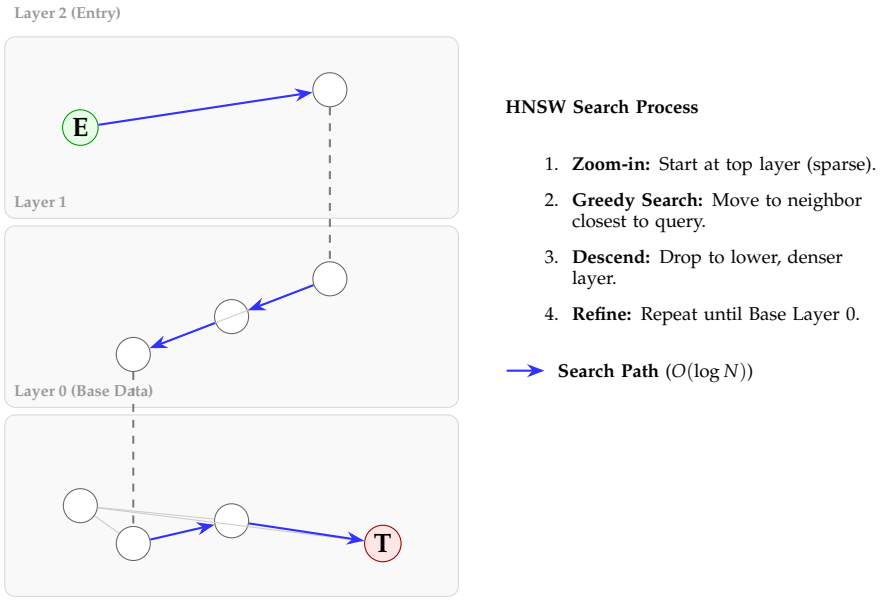


Figure 2.5: Conceptual structure of the Hierarchical Navigable Small Worlds (HNSW) graph. The multi-layer architecture allows the search algorithm to quickly "zoom in" from the sparse top layer to the dense data layer, achieving logarithmic search complexity.

Integration Architecture: Dynamic Agentic Orchestration

This thesis introduces a novel agent-orchestrated architecture that dynamically integrates RAG and Knowledge Graph components for test scope analysis. Unlike static pipelines, which follow a predetermined sequence of retrieval steps, our approach leverages an AI agent

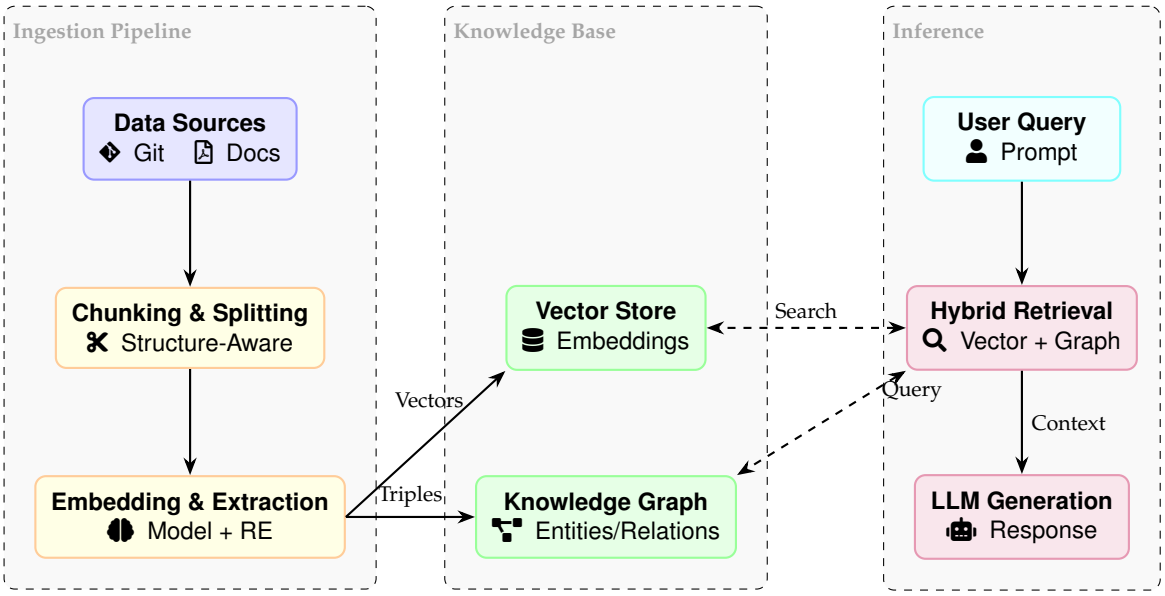


Figure 2.6: The Retrieval-Augmented Generation (RAG) Pipeline Architecture, showing data ingestion, processing, and storage components.

to adaptively select and combine retrieval strategies based on the nuances of the user's query and the evolving information scent.

Static vs. Dynamic Retrieval Architectures

In a typical *Static RAG pipeline*, a user query undergoes a fixed sequence of operations—e.g., embedding, vector search, graph traversal, then summarization. While predictable and robust for homogeneous query types, this rigidity is ill-suited for the heterogeneous information landscape of software engineering. Code entities (functions, classes), issue descriptions (natural language, error codes), and trace links (structural dependencies) each demand different retrieval mechanisms.

Conversely, *Dynamic Agentic Orchestration* empowers an LLM-based agent to serve as a reasoning engine that autonomously decides the most effective retrieval path. This approach is grounded in **Information Foraging Theory** [51], which posits that agents optimize their search strategies by following "information scent"—cues in their environment that suggest the proximity and value of desired information. In our context, the agent follows the "scent" of the query to adaptively choose between semantic, structural, or hybrid retrieval modes.

Empirical motivation for this dynamic approach stems from the diverse nature of test scope analysis queries:

- **Semantic-Only Queries:** A query like "*Tests related to authentication failures*" → Best served by **Vector Search**, identifying semantically similar code snippets or documentation. A graph walk alone might miss relevant tests if they aren't explicitly linked to "Auth" nodes.
- **Structural-Only Queries:** For "*All tests calling 'updateUser()'*", **Graph Traversal** is optimal, leveraging explicit call graph dependencies. Vector search might struggle with lexical variations or miss tests that call wrappers of this function.

- **Hybrid Queries:** Complex requests such as "*Tests for user login with database timeouts*" necessitate a **Hybrid Search**, combining semantic understanding of "login" with structural analysis of "database" interactions.

A static pipeline would either always use both strategies (inefficient, adding noise) or choose one (failing on queries requiring the other), thereby underperforming across the full spectrum of user intents. Dynamic selection is thus a practical necessity for achieving comprehensive and precise test scope analysis.

Query Analysis for Strategy Selection

The agent performs intent recognition and strategy selection primarily through **Chain-of-Thought (CoT)** reasoning within its ReAct "Thought" step. This process leverages internal reasoning and prompt patterns to classify query types:

- **Structural Signal:** Queries containing phrases like "*all tests calling X*" or "*tests affected by class Y*" signal a need for graph-based structural reasoning, leading the agent to invoke `graph_traverse()`.
- **Semantic Signal:** Phrases such as "*tests related to Z*" or "*find issues about W*" indicate a semantic search, prompting the agent to use `vector_search()`.
- **Hybrid Signal:** Queries combining descriptive language with specific entities, e.g., "*tests for X with Y*", suggest a need for `hybrid_search()`.

This emergent classification behavior, guided by the agent's system prompt and tool descriptions (detailed in Section 2.6), allows for flexible adaptation without an explicit, separate classification model.

Tool-Mediated Integration

The agent interacts with the underlying retrieval components through a set of well-defined tools, abstracting away their complexity. These tools serve as the operational interface for the agent's dynamic orchestration:

- `vector_search(query: str, k: int) -> List[Document]`: Queries the vector index for semantically similar documents.
- `graph_traverse(start_node: str, relation: str, depth: int) -> List[Node]`: Explores the Knowledge Graph based on specified starting nodes, relationship types, and traversal depth.
- `hybrid_search(query: str) -> List[Result]`: Combines vector and keyword search, fusing results using techniques like RRF.

These tool signatures hide the underlying algorithmic complexities (e.g., HNSW layers, BM25 tuning) from the agent, allowing it to reason at a higher level of abstraction about *which* tool to use rather than *how* to use it. This modular design also enables independent refinement of individual retrieval components without altering the agent's reasoning logic.

2.5 GraphRAG

While traditional RAG systems rely primarily on vector similarity to retrieve disjoint chunks of text, they often struggle with queries that require global reasoning or traversing complex dependencies. *GraphRAG* (Graph-based Retrieval-Augmented Generation) addresses this limitation by integrating Knowledge Graphs into the retrieval process.

Traditional RAG vs. GraphRAG

Traditional RAG, or *Baseline RAG*, follows a "retrieve-then-read" paradigm where the system fetches top-k semantically similar documents and feeds them to the LLM [11]. This approach is effective for explicit fact retrieval but often fails when the answer requires synthesizing information across multiple, indirectly connected documents (e.g., "How does a change in the billing module affect the reporting service?").

GraphRAG, specifically the approach formalized by Edge et al. [17], augments this by using the structural connections within a Knowledge Graph. It allows the system to:

- **Traverse Relationships:** Moving from a retrieved entity to its neighbors to gather relevant context (e.g., finding all tests linked to a modified function).
- **Synthesize Global Context:** Using community detection or path traversal to generate answers that reflect the broader system architecture rather than just isolated snippets.

In the context of test scope analysis, GraphRAG enables the retrieval system to "reason" about the software's structure, identifying test cases that are not textually similar to a code change but are structurally dependent on it.

Entity Linking and Grounding

A prerequisite for effective GraphRAG is **Entity Linking** (or Grounding). This is the process of mapping mentions of artifacts in unstructured text (e.g., user queries or bug reports) to their corresponding unique nodes within the Knowledge Graph. Without accurate linking, the system cannot locate the correct 'start nodes' for traversal, rendering the graph structure inaccessible. In agentic systems, this is often performed via hybrid search (using dense and sparse retrieval to find candidate nodes) or by leveraging the LLM's reasoning capabilities to disambiguate entity names before traversal [**shen2015entity**].

2.6 AI Agents

AI Agents represent a paradigm shift from simple LLM interactions to autonomous, goal-oriented systems. An AI Agent can be conceptualized as an **LLM equipped with Memory, Tools, Reasoning, and Planning capabilities**. This architecture enables agents to break down complex tasks, interact with their environment, and learn from feedback.

Key Components of an AI Agent

- **LLM Integration:** The Large Language Model serves as the agent's brain, interpreting inputs, generating thoughts and plans, and deciding on actions. It translates high-level goals into executable steps and understands the outputs from tools.
- **Memory Systems:** Agents require both short-term and long-term memory to maintain context and accumulate knowledge.
  - **Internal State/Short-Term Memory:** This includes the current conversational context, scratchpad for intermediate thoughts, and temporary variables. Strategies like *Compaction* are used here to prevent context rot [52].
  - **External/Long-Term Memory:** Rather than a static database, long-term memory in advanced agents acts as a "curated playbook" [53]. The agent actively synthesizes lessons from past interactions, storing refined strategies and domain facts in the RAG system (Knowledge Graph), enabling it to improve over time without retraining.

- **Tools:** Tools are external functions, APIs, or scripts that an agent can invoke to interact with its environment, perform specific operations (e.g., search a database, execute code, call a RAG pipeline), or gather information.
- **Guardrails:** These are mechanisms implemented to ensure agents operate safely, reliably, and within defined ethical and operational boundaries. In the context of this thesis and deployment within a corporate environment like Ericsson, guardrails are critical for protecting proprietary intellectual property and ensuring operational integrity. They can be implemented using two complementary approaches:
  - **Deterministic Guardrails:** Use rule-based logic to enforce strict compliance [54]. For test scope analysis, this includes verifying that suggested test files actually exist in the repository before recommendation, or enforcing the redaction of Personally Identifiable Information (PII), such as masking sensitive user IDs in bug reports, before they are processed by the model.
  - **Model-based Guardrails:** Utilize LLMs to evaluate the semantic quality of inputs and outputs. Techniques such as *Constitutional AI* [55] use AI feedback to align models with safety principles, while specialized models like *Llama Guard* [56] classify content risks. In this system, model-based guardrails validate the reasoning behind test scope recommendations, ensuring the agent does not hallucinate connections between unrelated code modules.

  Additionally, *Human-in-the-loop (HITL)* mechanisms provide a critical layer of oversight for high-stakes actions, consistent with the interactive machine learning principles outlined by Amershi et al. [57]. This architectural pattern involves pausing the agent's execution flow (interrupts) when a sensitive action is proposed. The system's state is persisted, allowing a human expert to review the request and exercise supervisory control by either *approving* the action, *editing* the parameters (e.g., refining a generated test case), or *rejecting* the proposal entirely with feedback.

  Agents also operate along a spectrum of autonomy, often categorized as **Human-in-the-Loop (HITL)** (where human approval is required for actions), **Human-on-the-Loop (HOTL)** (where humans supervise but intervene only in exceptions), and **Human-out-of-the-Loop** (full autonomy). For high-stakes software engineering tasks, navigating this spectrum is crucial: strictly enforcing HITL (e.g., "Safe Mode" in our implementation) ensures correctness during exploration, while allowing higher autonomy (e.g., "YOLO Mode") facilitates speed for trusted workflows.

Reasoning and Planning

Agents possess mechanisms to plan a sequence of actions to achieve a goal. The foundational paradigm is the *ReAct* (Reason+Act) framework [18], which enables LLMs to interleave reasoning traces ("Thoughts") with task-specific "Actions" and "Observations."

ReAct Loop and Prompt Engineering for Tool Use

The core of the ReAct paradigm is its iterative Thought-Action-Observation loop. The agent, driven by the LLM, first generates a 'Thought' based on the current goal and available information. This 'Thought' guides the subsequent 'Action', which is typically a call to an external tool. The 'Observation' is the result of that tool's execution, which then feeds back into the loop to inform the next 'Thought'.

Effective tool use in this loop is heavily reliant on precise **Prompt Engineering**. The LLM is provided with a **System Prompt** that acts as its operational manual, defining its persona, overall objective, and crucially, the interface to its available tools. This system prompt typically includes:

- **Agent Persona and Goal:** Setting the context for the agent's role (e.g., "You are an expert test scope analysis agent...") and its primary objective.
- **Available Tools and Descriptions:** A structured list of callable functions (e.g., `vector_search`, `graph_traverse`, `hybrid_search`), along with clear, concise descriptions of their purpose, input parameters, and expected output types. These tool definitions are critical; they enable the agent to understand *when* and *how* to invoke each tool.
- **Output Format Constraints:** Guiding the LLM to generate tool calls in a specific, parseable format (e.g., JSON).

This detailed prompt structure enables the emergent classification behavior described in Section 2.4. By analyzing the user's query and the current state, the agent's 'Thought' process identifies the most appropriate tool or sequence of tools, operationalizing the dynamic retrieval strategy. The quality of these tool definitions directly impacts the agent's ability to accurately map user intent to effective retrieval actions.

The ACE Cycle

Building on this, advanced systems employ the ACE cycle [53] for self-improvement: *Generation* (of the plan), *Reflection* (critiquing the plan/result), and *Curation* (saving the lesson to memory).

Context Engineering

Effective RAG and Agentic systems are not merely about retrieving data, but about *adapting* it to maximize the LLM's reasoning capabilities. Context Engineering represents a systematic framework for this adaptation, treating context not as a static buffer but as an evolving playbook for the agent. It is distinct from *Prompt Engineering*, which focuses on optimizing the instructions given to the model. Context Engineering, conversely, focuses on the architecture and state management of the information (context window) supplied to the model to support complex reasoning tasks [52, 53].

The Challenge of Context Rot

As systems scale, simply retrieving more data leads to "Context Rot," a phenomenon where the model's ability to recall and reason about specific information degrades as the context window fills with distractors [52]. This aligns with the concept of "Brevity Bias," where critical domain insights are lost amidst noise [53]. Therefore, treating context as a finite, high-value resource is essential.

The ACE Framework

To address these limitations, we adopt principles from the Agentic Context Engineering (ACE) framework [53]. ACE treats context construction as a modular process involving:

- **Generation:** Retrieving and drafting initial context blocks based on the query.
- **Reflection:** Analyzing the retrieved data to identify gaps or inconsistencies.
- **Curation:** Iteratively refining and structuring the context to create a coherent narrative or "playbook" for the agent.

Optimization Strategies

Practical implementation relies on specific strategies to combat rot and maintain long-horizon coherence [52]:

- **Compaction:** Periodically summarizing conversation history or verbose tool outputs to retain only the essential state changes, freeing up tokens for new reasoning.
- **Structured Note-Taking:** Empowering the agent to explicitly "write down" key facts or decisions into a dedicated memory block (or Knowledge Graph), rather than relying on implicit recall from a long transcript.
- **Delimited Evidence:** Using clear XML-style tags (e.g., `<code_snippet>`) to demarcate external data from internal logic, preventing the model from confusing retrieved evidence with system instructions.

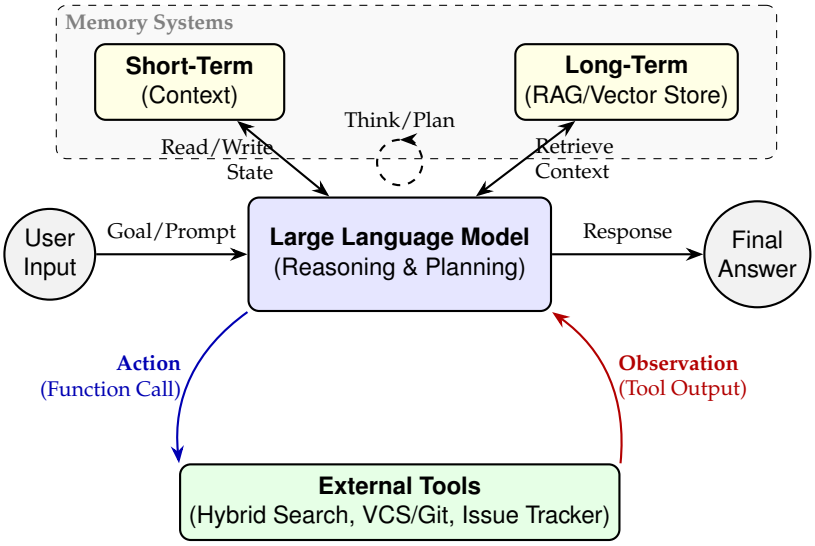


Figure 2.7: Architectural overview of an AI Agent, illustrating the interaction between the LLM "Brain," Memory systems, and External Tools via the ReAct cycle.

2.7 Evaluation Metrics

To rigorously assess the performance of the proposed retrieval system, standard metrics from the field of Information Retrieval (IR) are employed. The primary goal in test scope analysis is to present relevant tests to the engineer (high recall) while minimizing the noise of irrelevant results (high precision) within the limited window of user attention [6].

Rank-Aware Metrics

Since the system provides a recommended list of tests, the order of results is critical. A relevant test appearing at position 50 is effectively "missed" by a busy engineer. Therefore, we focus on rank-aware metrics:

- **Recall@k:** Measures the proportion of relevant items retrieved within the top k results (e.g., $k = 10$). This is the primary safety metric, indicating the system's ability to surface correct tests within the user's immediate view.

- **Precision@k:** Measures the proportion of items in the top k results that are actually relevant. High precision@k ensures that the user does not waste time reviewing irrelevant suggestions.
- **Mean Average Precision (MAP):** While Recall@k focuses on a specific cutoff, MAP provides a single-figure measure of quality across recall levels. It calculates the average precision at the position of every relevant item, rewarding systems that place relevant tests higher in the list. This metric is widely regarded as the standard for evaluating ranked retrieval systems [6].
- **Mean Reciprocal Rank (MRR):** In agentic workflows, the system often acts on the *first* relevant result found. MRR measures the average of the reciprocal ranks of the first relevant item for a set of queries (1/rank). An MRR of 1.0 implies the first result is always correct, which is ideal for autonomous agents.
- **F1-Score (@k):** The harmonic mean of Precision@k and Recall@k. This metric is essential for balancing the trade-off between coverage and noise, ensuring the system does not achieve high recall simply by retrieving an excessive number of documents.

3 Method

This chapter details the implementation of the agentic RAG system and the experimental methodology used to evaluate it. The system integrates a dual-storage architecture with an autonomous agent capable of dynamic retrieval strategy selection. To bridge the theoretical concepts established in Chapter 2 with the concrete implementation, Table 3.1 provides a mapping of each concept to its corresponding technology.

Table 3.1: Mapping of Theoretical Concepts to Implementation Technologies

Concept	**Technology**	**Role in System**
Agent Orchestration (§2.6)	LangChain & LangGraph	Provides agent abstractions and state machine runtime
Graph Database (§2.3)	Neo4j	Stores structural relationships between software artifacts
Vector Search (§2.4)	PostgreSQL + pgvector	Stores embeddings and performs HNSW-based similarity search
Keyword Search (§2.4)	PostgreSQL + pg_search	Provides BM25-based lexical retrieval
Document Layout Analysis (§2.4)	Docling	Extracts structure-preserving text from PDF documents
AST-based Splitting (§2.4)	Tree-sitter	Parses code into syntax trees for semantic chunking
Hybrid Fusion (§2.4)	Reciprocal Rank Fusion	Merges vector and keyword results by rank position

3.1 System Architecture

The system architecture follows a layered design, separating data storage, retrieval mechanisms, and agentic control logic (see Figure **??**).

- **Storage Layer:** Implements a "Dual Storage" pattern that separates *structural* and *semantic* concerns. A graph database stores explicit relationships between software arti-

facts (e.g., `TestCase` $\xrightarrow{VERIFIES}$ `Requirement`), enabling the structural traversal described in Section 2.3. A relational database with vector and full-text extensions stores textual representations of these entities, indexed for both semantic similarity and keyword matching.

- **Retrieval Layer:** Exposes four tools that abstract the underlying storage complexities: `vector_search` for semantic queries, `keyword_search` for exact identifier matching, `graph_traverse` for structural exploration, and `hybrid_search` for queries requiring both approaches.
- **Agent Layer:** An orchestration layer that uses an LLM configured as a ReAct agent (see Section 2.6) to dynamically select and chain retrieval tools based on the user's query intent. This layer implements the dynamic strategy selection described in Section 2.4.

3.2 Data Processing Pipeline

To construct a comprehensive Knowledge Graph for test scope analysis, the system ingests data from three primary sources: unstructured documentation, source code repositories, and structured test execution logs.

Document Ingestion and Layout Analysis

A key challenge in processing technical documentation is that standard PDF extractors flatten visual layouts into plain text streams, losing critical structural information such as section hierarchies and table relationships. To address this, the system employs Document Layout Analysis (DLA), as introduced in Section 2.4.

The implementation uses an open-source document conversion library [58] that internally applies a multimodal vision model [59] to detect and classify layout elements (headers, paragraphs, tables, lists) before extraction. For tables—which are particularly problematic in PDF formats—a specialized table structure recognition model [60] reconstructs cell relationships, ensuring that a test parameter value remains linked to its column header rather than being extracted as disconnected text.

Code Analysis and AST Parsing

As discussed in Section 2.4, naive text splitting (e.g., every 500 characters) disrupts the logical integrity of code, severing function bodies mid-statement or separating method signatures from their implementations. To preserve semantic coherence, the system performs *structure-aware splitting* by first parsing the codebase into an Abstract Syntax Tree (AST).

The implementation uses an incremental parsing library [61] that supports multiple programming languages through a unified grammar interface. By traversing the resulting syntax tree, the system identifies natural boundaries (functions, classes, methods) and creates chunks that each represent a complete, self-contained unit of logic. This ensures that when a code fragment is embedded and later retrieved, it contains sufficient context for meaningful analysis.

Test Data Ingestion

The third data source is structured test execution metadata from the organization's test management systems. In this study, this data originates from Ericsson's internal Test Governance Framework (TGF), which exports test results as CSV files containing fields such as test identifiers, execution outcomes, timestamps, and—crucially—explicit links to requirements and code functions.

A dedicated loader component parses these exports, performing data normalization (e.g., mapping variant result strings like "passed" or "ok" to a canonical "PASS" status) and creating the ground-truth relationships in the Knowledge Graph:

- `TestCase` $\xrightarrow{VERIFIES}$ `Requirement`
- `TestCase` $\xrightarrow{COVERS}$ `Function`

These explicit trace links serve as the "golden standard" for evaluating the system's retrieval performance, enabling deterministic calculation of precision and recall metrics.

3.3 Retrieval Implementation

To enable a comparative analysis of retrieval strategies, four distinct retrieval mechanisms were implemented as callable tools.

Vector and Keyword Search

- **Vector Search:** Semantic retrieval is implemented using a relational database extended with vector operations [62]. Document embeddings (768 dimensions) are stored in an HNSW index, as described in Section 2.4, enabling sub-second approximate nearest-neighbor queries using cosine distance.
- **Keyword Search:** Lexical retrieval uses the BM25 algorithm [42] via a full-text search extension [63]. This tool is optimized for exact identifier matching—essential for locating specific error codes (e.g., `ERR_TIMEOUT_503`) or function signatures that semantic search might miss due to tokenization issues.

Graph Traversal

The `graph_traverse` tool queries the Knowledge Graph using parameterized pattern-matching queries (see Section 2.3). It requires a specific starting point (a node identifier) and allows the agent to explore that entity's neighborhood. The tool's interface—`graph_traverse(start_node, relationship_type, depth)`—constrains the agent to valid traversals, preventing unbounded graph walks that could degrade performance. This tool is the primary mechanism for answering structural queries such as "which tests verify requirement X?"

Hybrid Search and Fusion

The `hybrid_search` tool combines the results of vector and keyword searches using **Reciprocal Rank Fusion (RRF)** [46], as described in Section 2.4. By normalizing the rank positions of documents from both retrieval lists (rather than their raw scores), RRF produces a single ranking that prioritizes items appearing prominently in both semantic and lexical results. This strategy is designed to handle complex queries that contain both domain concepts and technical identifiers.

Agentic Orchestration

The agent is implemented as a state machine following the ReAct paradigm (Section 2.6) and the tool-mediated integration pattern (Section 2.4). Its behavior is governed by a **System Prompt** that defines a "Tool Selection Strategy," instructing the model to classify queries into four types:

- **Identifier Queries:** "What tests cover REQ-123?" → Prioritize `keyword_search`.

- **Conceptual Queries:** "Tests for latency issues?" → Prioritize `vector_search`.
- **Structural Queries:** "Dependencies of module X?" → Prioritize `graph_traverse`.
- **Complex Queries:** "Login tests with timeout errors?" → Prioritize `hybrid_search`.

To ensure reliable operation, the agent incorporates several guardrail mechanisms:

- **Call Limits:** Hard limits on both LLM inference calls and tool executions per session prevent infinite reasoning loops and control costs.
- **Human Oversight:** An interrupt mechanism pauses execution before sensitive operations, enabling the Human-in-the-Loop workflow described in Section 3.4.

3.4 Human-in-the-Loop Implementation

To enable user control and oversight, the system implements a "Safe Mode" following the Human-in-the-Loop (HITL) principles described in Section 2.6. When enabled, the agent's execution is interrupted before any retrieval tool is invoked. The current state—including conversation history and the proposed action—is persisted to the database, allowing the session to be resumed later.

At each interrupt, the user is presented with the agent's proposed action and can:

- **Approve:** Allow the action to proceed as proposed.
- **Edit:** Modify the tool parameters (e.g., change the search query or traversal depth).
- **Reject:** Cancel the action and optionally provide corrective feedback.

This workflow enables a "Human-on-the-Loop" mode where the agent operates semi-autonomously under supervision, allowing practitioners to maintain control over high-stakes decisions while still benefiting from automated assistance.

3.5 Evaluation Methodology

The system's performance was evaluated using a synthetic dataset derived from the TGF schema, simulating real-world telecommunications testing scenarios.

Dataset Generation

To enable controlled evaluation, a synthetic "golden standard" dataset was generated following the TGF schema structure. This dataset contains a controlled set of Requirements, Functions, and Test Cases with explicitly defined trace links. The explicit linking enables deterministic relevance judgments: a retrieved test case is counted as a "hit" only if the ground-truth data explicitly connects it to the queried entity.

Experimental Setup

The evaluation compared the performance of five distinct strategies on a standard set of test scope queries:

1. **Vector-Only:** Pure semantic search.
2. **Keyword-Only:** Pure BM25 search.
3. **Graph-Only:** Pure structural traversal (assuming a correct start node).
4. **Hybrid:** RRF fusion of Vector and Keyword.
5. **Agentic:** The dynamic orchestrator described in Section 3.3.

Metrics Calculation

Retrieval quality is quantified using standard Information Retrieval metrics [6], as introduced in Section 2.7:

- **Precision@k:** The fraction of relevant items in the top-k results.
- **Recall@k:** The fraction of total relevant items retrieved in the top-k.
- **Mean Average Precision (MAP):** A rank-aware metric that averages the precision scores at the rank of each relevant document.
- **Mean Reciprocal Rank (MRR):** The average of the multiplicative inverse of the rank of the first correct answer ($1/rank$).
- **F1-Score@k:** The harmonic mean of Precision@k and Recall@k.

4 Time Plan

The following Gantt chart outlines the preliminary timeline for the thesis project. This aggressive schedule aims for completion by the end of June 2026, with technical work finishing by the end of May.

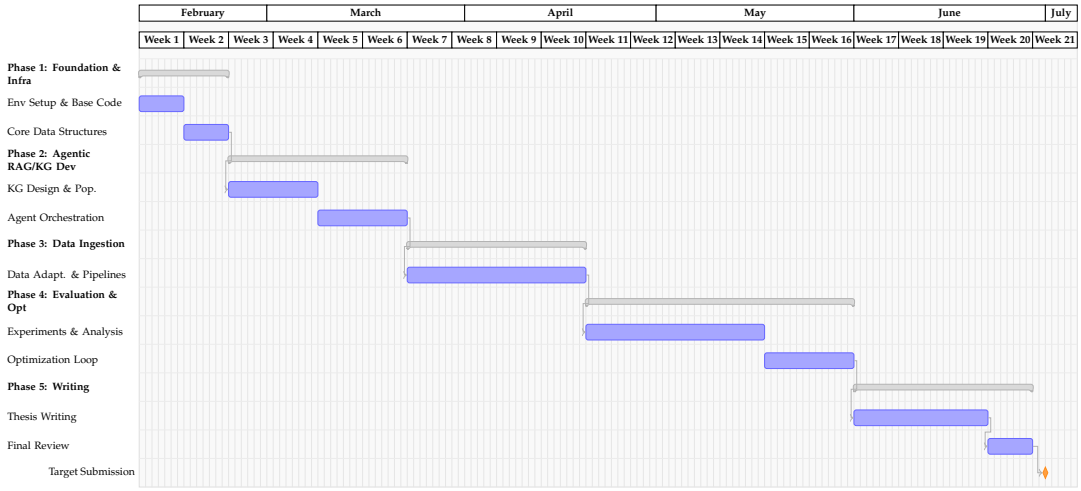


Figure 4.1: Project Timeline and Phases

5 Results

This chapter presents the results. Note that the results are presented factually, striving for objectivity as far as possible. The results shall not be analyzed, discussed or evaluated. This is left for the discussion chapter.

In case the method chapter has been divided into subheadings such as pre-study, implementation and evaluation, the result chapter should have the same sub-headings. This gives a clear structure and makes the chapter easier to write.

In case results are presented from a process (e.g. an implementation process), the main decisions made during the process must be clearly presented and justified. Normally, alternative attempts, etc, have already been described in the theory chapter, making it possible to refer to it as part of the justification.

6 Discussion

This chapter contains the following sub-headings.

6.1 Results

Are there anything in the results that stand out and need be analyzed and commented on? How do the results relate to the material covered in the theory chapter? What does the theory imply about the meaning of the results? For example, what does it mean that a certain system got a certain numeric value in a usability evaluation; how good or bad is it? Is there something in the results that is unexpected based on the literature review, or is everything as one would theoretically expect?

6.2 Method

This is where the applied method is discussed and criticized. Taking a self-critical stance to the method used is an important part of the scientific approach.

A study is rarely perfect. There are almost always things one could have done differently if the study could be repeated or with extra resources. Go through the most important limitations with your method and discuss potential consequences for the results. Connect back to the method theory presented in the theory chapter. Refer explicitly to relevant sources.

The discussion shall also demonstrate an awareness of methodological concepts such as replicability, reliability, and validity. The concept of replicability has already been discussed in the Method chapter (3). Reliability is a term for whether one can expect to get the same results if a study is repeated with the same method. A study with a high degree of reliability has a large probability of leading to similar results if repeated. The concept of validity is, somewhat simplified, concerned with whether a performed measurement actually measures what one thinks is being measured. A study with a high degree of validity thus has a high level of credibility. A discussion of these concepts must be transferred to the actual context of the study.

The method discussion shall also contain a paragraph of source criticism. This is where the authors' point of view on the use and selection of sources is described.

In certain contexts it may be the case that the most relevant information for the study is not to be found in scientific literature but rather with individual software developers and open

source projects. It must then be clearly stated that efforts have been made to gain access to this information, e.g. by direct communication with developers and/or through discussion forums, etc. Efforts must also be made to indicate the lack of relevant research literature. The precise manner of such investigations must be clearly specified in a method section. The paragraph on source criticism must critically discuss these approaches.

Usually however, there are always relevant related research. If not about the actual research questions, there is certainly important information about the domain under study.

6.3 The work in a wider context

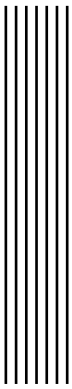
There must be a section discussing ethical and societal aspects related to the work. This is important for the authors to demonstrate a professional maturity and also for achieving the education goals. If the work, for some reason, completely lacks a connection to ethical or societal aspects this must be explicitly stated and justified in the section Delimitations in the introduction chapter.

In the discussion chapter, one must explicitly refer to sources relevant to the discussion.

7 Conclusion

This chapter contains a summarization of the purpose and the research questions. To what extent has the aim been achieved, and what are the answers to the research questions?

The consequences for the target audience (and possibly for researchers and practitioners) must also be described. There should be a section on future work where ideas for continued work are described. If the conclusion chapter contains such a section, the ideas described therein must be concrete and well thought through.



Acknowledgments

Acknowledgments.tex

Populärvetenskaplig sammanfattning på svenska i sammanfattning.tex

Bibliography

[1] D. Graham and M. Fewster, *Experiences of test automation: case studies of software test automation*. Addison-Wesley Professional, 2012.

[2] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, pp. 1–42, 2013.

[3] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996. DOI: 10.1109/32.536955.

[4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:34563682.

[5] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024. DOI: 10.1109/TSE.2024.3368208.

[6] C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press, 2008, ISBN: 978-0521865715. [Online]. Available: https://nlp.stanford.edu/IR-book/.

[7] F. Gomes De Oliveira Neto, J. Horkoff, E. Knauss, R. Kasauli, and G. Liebel, "Challenges of aligning requirements engineering and system testing in large-scale agile: A multiple case study," in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, 2017, pp. 315–322. DOI: 10.1109/REW.2017.33.

[8] B. Wang, H. Wang, R. Luo, S. Zhang, and Q. Zhu, "A systematic mapping study of information retrieval approaches applied to requirements trace recovery.," in *SEKE*, 2022, pp. 1–6.

[9] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation: A retrospective," *IEEE Transactions on Software Engineering*, vol. 51, no. 3, pp. 825–832, 2025. DOI: 10.1109/TSE.2025.3534027.

[10] R. Fauzan, D. Siahaan, S. Rochimah, and E. Triandini, "A different approach on automated use case diagram semantic assessment," *International Journal of Intelligent Engineering and Systems*, vol. 14, no. 1, pp. 496–505, 2021.

[11] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. K"uttler, M. Lewis, W.-t. Yih, T. Rockt"aschel, S. Riedel, and D. Kiela, *Retrieval-augmented generation for knowledge-intensive nlp tasks*, 2021. arXiv: 2005.11401 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2005.11401.

[12] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, IEEE, 2023, pp. 31–53.

[13] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, S. Yu, B. Zhang, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[14] V. Kesri, A. Nayak, and K. Ponnalagu, "Autokg-an automotive domain knowledge graph for software testing: A position paper," in *2021 ieee international conference on software testing, verification and validation workshops (icstw)*, IEEE, 2021, pp. 234–238.

[15] S. Radhakrishnan, A. G. Cecchettini, S. Wendler, and A. Graf, "Create, explore and analyse traceability knowledge graphs," in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, IEEE, 2023, pp. 453–454.

[16] M. Cheng, Y. Luo, J. Ouyang, Q. Liu, H. Liu, L. Li, S. Yu, B. Zhang, J. Cao, J. Ma, et al., "A survey on knowledge-oriented retrieval-augmented generation," *arXiv preprint arXiv:2503.10677*, 2025.

[17] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, and J. Larson, *From local to global: A graph rag approach to query-focused summarization*, 2025. arXiv: 2404.16130 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2404.16130.

[18] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, *React: Synergizing reasoning and acting in language models*, 2023. arXiv: 2210.03629 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2210.03629.

[19] D. Fuchß, T. Hey, J. Keim, H. Liu, N. Ewald, T. Thirolf, and A. Koziolek, "Lissa: Toward generic traceability link recovery through retrieval-augmented generation," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering. ICSE*, vol. 25, 2025.

[20] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010, ISBN: 978-0321601919.

[21] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016. [Online]. Available: http://landing.google.com/sre/book.html.

[22] DeepSeek-AI et al., *Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning*, 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2501.12948.

[23] Google DeepMind, *Gemini 3 Pro Model Card*, https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf, Released: 2025-11-18. Accessed: 2025-11-25, Nov. 2025.

[24] Anthropic, *Claude Sonnet 4.5 System Card*, https://assets.anthropic.com/m/12f214efcc2f457a/original/Claude-Sonnet-4-5-System-Card.pdf, Released: 2025-09-29. Accessed: 2025-11-25. Characterized as a hybrid reasoning model., Sep. 2025.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, 2017. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[26] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[27] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[28] Z. Shen, H. Xu, Y. Deng, S. Zhou, B. Zheng, X. Li, Y. Liu, B. Guo, Y. Zhang, W. Wang, M. Yuan, and J. Zheng, *Llm with tools: A survey*, 2024. arXiv: 2409.18807 [cs.CL].

[29] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xiong, E. H. Huang, Q. V. Gu, A. Le, J. Mograbi, et al., "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[30] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, *Unifying large language models and knowledge graphs: A roadmap*, 2024. arXiv: 2306.08302 [cs.CL].

[31] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1433–1445, ISBN: 9781450347037. DOI: 10.1145/3183713.3190657. [Online]. Available: https://doi.org/10.1145/3183713.3190657.

[32] V. A. Traag, L. Waltman, and N. J. van Eck, "From louvain to leiden: Guaranteeing well-connected communities," *Scientific Reports*, vol. 9, no. 1, Mar. 2019, ISSN: 2045-2322. DOI: 10.1038/s41598-019-41695-z. [Online]. Available: http://dx.doi.org/10.1038/s41598-019-41695-z.

[33] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Workshop at ICLR*, 2013.

[34] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: 2002.08155 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2002.08155.

[35] Y. Zhang, M. Li, D. Long, X. Zhang, H. Lin, B. Yang, P. Xie, A. Yang, D. Liu, J. Lin, F. Huang, and J. Zhou, *Qwen3 embedding: Advancing text embedding and reranking through foundation models*, 2025. arXiv: 2506.05176 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2506.05176.

[36] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: 1810.04805 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1810.04805.

[37] N. Reimers and I. Gurevych, *Sentence-bert: Sentence embeddings using siamese bert-networks*, 2019. arXiv: 1908.10084 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1908.10084.

[38] OpenAI, *New embedding models and api updates*, https://openai.com/index/new-embedding-models-and-api-updates/, Accessed: 2025-11-25, 2024.

[39] J. Lee, F. Chen, S. Dua, D. Cer, M. Shanbhogue, I. Naim, G. H. Ábrego, Z. Li, K. Chen, H. S. Vera, X. Ren, S. Zhang, D. Salz, M. Boratko, J. Han, B. Chen, S. Huang, V. Rao, P. Suganthan, F. Han, A. Doumanoglou, N. Gupta, F. Moiseev, C. Yip, A. Jain, S. Baumgartner, S. Shahi, F. P. Gomez, S. Mariserla, M. Choi, P. Shah, S. Goenka, K. Chen, Y. Xia, K. Chen, S. M. K. Duddu, Y. Chen, T. Walker, W. Zhou, R. Ghiya, Z. Gleicher, K. Gill, Z. Dong, M. Seyedhosseini, Y. Sung, R. Hoffmann, and T. Duerig, *Gemini embedding: Generalizable embeddings from gemini*, 2025. arXiv: 2503.07891 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2503.07891.

[40] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, *Bge m3-embedding: Multilingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation*, 2024. arXiv: 2402.03216 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2402.03216.

[41] K. SPARCK JONES, "A statistical interpretation of term specificity and its application in retrieval," *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, Jan. 1972, ISSN: 0022-0418. DOI: 10.1108/eb026526. eprint: https://www.emerald.com/jd/article-pdf/28/1/11/1336479/eb026526.pdf. [Online]. Available: https://doi.org/10.1108/eb026526.

[42] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, pp. 333–389, Apr. 2009, ISSN: 1554-0669. DOI: 10.1561/1500000019. [Online]. Available: https://doi.org/10.1561/1500000019.

[43] S. E. Robertson and S. Walker, "Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval," in *SIGIR '94*, B. W. Croft and C. J. van Rijsbergen, Eds., London: Springer London, 1994, pp. 232–241, ISBN: 978-1-4471-2099-5.

[44] S. Bruch, S. Gai, and A. Ingber, "An analysis of fusion functions for hybrid retrieval," *ACM Transactions on Information Systems*, vol. 42, no. 1, pp. 1–35, Aug. 2023, ISSN: 1558-2868. DOI: 10.1145/3596512. [Online]. Available: http://dx.doi.org/10.1145/3596512.

[45] Z. Rackauckas, "Rag-fusion: A new take on retrieval augmented generation," *International Journal on Natural Language Computing*, vol. 13, no. 1, pp. 37–47, Feb. 2024, ISSN: 2319-4111. DOI: 10.5121/ijnlc.2024.13103. [Online]. Available: http://dx.doi.org/10.5121/ijnlc.2024.13103.

[46] G. V. Cormack, C. L. A. Clarke, and S. Buettcher, "Reciprocal rank fusion outperforms condorcet and individual rank learning methods," in *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '09, Boston, MA, USA: Association for Computing Machinery, 2009, pp. 758–759, ISBN: 9781605584836. DOI: 10.1145/1571941.1572114. [Online]. Available: https://doi.org/10.1145/1571941.1572114.

[47] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, *Retrieval-augmented generation for large language models: A survey*, 2024. arXiv: 2312.10997 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2312.10997.

[48] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, *Lost in the middle: How language models use long contexts*, 2023. arXiv: 2307.03172 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2307.03172.

[49] J. Johnson, M. Douze, and H. Jégou, *Billion-scale similarity search with gpus*, 2017. arXiv: 1702.08734 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1702.08734.

[50] Y. A. Malkov and D. A. Yashunin, *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs*, 2018. arXiv: 1603.09320 [cs.DS]. [Online]. Available: https://arxiv.org/abs/1603.09320.

[51] P. Pirolli, "Information foraging," in *Encyclopedia of Database Systems*, L. LIU and M. T. "OZSU, Eds. Boston, MA: Springer US, 2009, pp. 1485–1490, ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_205. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_205.

[52] Anthropic, *Effective context engineering for ai agents*, Accessed: 2025-11-25, 2025. [Online]. Available: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents.

[53] Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li, U. Thakker, J. Zou, and K. Olukotun, *Agentic context engineering: Evolving contexts for self-improving language models*, 2025. arXiv: 2510.04618 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2510.04618.

[54] *Langchain guardrails*, https://docs.langchain.com/oss/python/langchain/guardrails, Accessed: 2025-11-25, Nov. 2025.

[55] Y. Bai, S. Kadavath, S. Kundu, A. Askell, J. Kernion, A. Jones, A. Chen, A. Goldie, A. Mirhoseini, C. McKinnon, C. Chen, C. Olsson, C. Olah, D. Hernandez, D. Drain, D. Ganguli, E. Tran-Johnson, E. Perez, J. Kerr, J. Mueller, J. Landau, K. Ndousse, K. Lukosuite, L. Lovitt, M. Sellitto, N. Elhage, N. Schiefer, N. Mercado, N. DasSarma, R. Lasenby, R. Larson, S. Ringer, S. Johnston, S. Kravec, S. E. Showk, S. Fort, T. Lanham, T. Telleen-Lawton, T. Conerly, T. Henighan, T. Hume, S. R. Bowman, Z. Hatfield-Dodds, B. Mann, D. Amodei, N. Joseph, S. McCandlish, T. Brown, and J. Kaplan, *Constitutional ai: Harmlessness from ai feedback*, 2022. arXiv: 2212.08073 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2212.08073.

[56] H. Inan, K. Upasani, J. Chi, R. Rungta, K. Iyer, Y. Mao, M. Tontchev, Q. Hu, B. Fuller, D. Testuggine, and M. Khabsa, *Llama guard: Llm-based input-output safeguard for human-ai conversations*, 2023. arXiv: 2312.06674 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2312.06674.

[57] "Power to the people: The role of humans in interactive machine learning," vol. 35, pp. 105–120, Dec. 2014. DOI: 10.1609/aimag.v35i4.2513. [Online]. Available: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2513.

[58] D. S. Team, "Docling technical report," Tech. Rep., version 1.0.0, Aug. 2024. DOI: 10.48550/arXiv.2408.09869. eprint: 2408.09869. [Online]. Available: https://arxiv.org/abs/2408.09869.

[59] B. Pfitzmann, C. Auer, and P. Staar, "Doclaynet: A large human-annotated dataset for document-layout segmentation," in *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 4090–4098. DOI: 10.1145/3534678.3539043.

[60] J. Yang, A. Gupta, S. Upadhyay, L. He, R. Goel, and S. Paul, "TableFormer: Robust transformer modeling for table-text encoding," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds., Dublin, Ireland: Association for Computational Linguistics, May 2022. DOI: 10.18653/v1/2022.acl-long.40. [Online]. Available: https://aclanthology.org/2022.acl-long.40/.

[61] M. Brunsfeld and Contributors, *Tree-sitter: An incremental parsing system for programming tools*, Software system, developed at GitHub, Version 0.25.2 released on September 25, 2025, development ongoing, GitHub, 2025. [Online]. Available: https://github.com/tree-sitter/tree-sitter.

[62] A. Kane and contributors, *pgvector: Open-source vector similarity search for postgres*, `https://github.com/pgvector/pgvector`, Accessed: 2025-11-30, 2024.

[63] ParadeDB Developers, *pg_search: Elastic-quality full text search inside PostgreSQL*, PostgreSQL Extension, built on Tantivy, using the BM25 algorithm, First stable release (v0.6.0) on November 14, 2023; development ongoing, ParadeDB, 2023. [Online]. Available: `github.com`.