



Enhanced Repository-Level Code Comprehension through Knowledge Graphs for Large Language Models

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Benedikt Hausberger, BSc.

Matrikelnummer 11908105

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.techn. Matthias Paul Lanzinger, BSc.

Wien, 10. Februar 2026

Benedikt Hausberger

Matthias Paul Lanzinger



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



Informatics

Enhanced Repository-Level Code Comprehension through Knowledge Graphs for Large Language Models

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Benedikt Hausberger, BSc.

Registration Number 11908105

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.techn. Matthias Paul Lanzinger, BSc.

Vienna, February 10, 2026

Benedikt Hausberger

Matthias Paul Lanzinger



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Benedikt Hausberger, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 10. Februar 2026

Benedikt Hausberger



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Diese Arbeit hätte ich ohne die Unterstützung einiger Personen nicht in dieser Weise umsetzen können. Ich bin diesen Menschen zu großem Dank verpflichtet und möchte hiermit meine tiefste Wertschätzung für die Begleitung in den letzten Jahren äußern.

Mein aufrichtiger Dank gilt in erster Linie meinem Betreuer Prof. Matthias Lanzinger. Seit dem ersten Gespräch hat er mich durch seine fachliche Expertise, seinen unermüdlichen Zuspruch und sein kritisches Feedback maßgeblich unterstützt. Ohne ihn hätte diese Arbeit nicht ihre jetzige Qualität erreicht.

Auf persönlicher Ebene danke ich von Herzen meiner Familie, die mich über Jahre hinweg motiviert und in meinen Vorhaben stets bestärkt hat. Hervorheben möchte ich meine Brüder Jakob und Michael, mit denen ich die Welt der Informatik schon früh erkunden durfte. Unsere wegweisenden Diskussionen haben diese Arbeit wesentlich bereichert.

Ein ganz besonderer Dank gilt meiner Freundin Teresa Baar. Sie stand mir in dieser intensiven Zeit stets mit Verständnis, Liebe und großem Rückhalt zur Seite.

Nicht zuletzt möchte ich meinen Freunden und Freundinnen meinen tiefsten Dank aussprechen. Sie haben mich stets ermutigt an diesem Projekt weiterzuarbeiten, aber sie haben mir auch bewusst einen gesunden Abstand zu dieser Arbeit gegeben.

Ohne euch wäre dieser Weg nicht möglich gewesen. Vielen Dank!



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would not have been able to complete this thesis in its present form without the support of several people. I am deeply indebted to them and would like to express my sincere appreciation and gratitude for all our interactions over the past few years.

My sincere thanks go first and foremost to my supervisor, Prof. Matthias Lanzinger. From our very first conversation, he has provided significant support through his professional expertise, tireless encouragement, and critical feedback. Without him, this thesis would not have reached its current level of quality.

On a personal level, I thank my family from the bottom of my heart for their motivation and constant encouragement in all my endeavors over the years. I would particularly like to mention my brothers, Jakob and Michael, with whom I was able to explore the world of computer science at an early age. Our insightful discussions have significantly enriched this work.

A very special thanks goes to my girlfriend, Teresa Baar. She stood by my side throughout this intense period with understanding, love, and unwavering support.

Last but not least, I would like to express my deepest gratitude to my friends. They not only encouraged me to continue working on this project but also ensured I maintained a healthy distance from the work when necessary.

Without you, this journey would not have been possible. Thank you!



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die Codegenerierung wurde durch jüngste Fortschritte bei Large Language Models (LLMs) maßgeblich beeinflusst. Das Verständnis auf Repository-Ebene bleibt jedoch insbesondere innerhalb von Enterprise-Java-Umgebungen, die stark auf Meta-Programming-Frameworks wie Spring und Lombok basieren, eine zentrale Herausforderung. In diesen Umgebungen werden Kontrollfluss und Logik häufig durch implizite Annotationen als durch expliziten Quellcode definiert. Infolgedessen übersehen herkömmliche textbasierte Retrieval-Augmented-Generation-Ansätze (RAG) kritische Abhängigkeiten und liefern irrelevanten Kontext.

Um diesen Einschränkungen zu begegnen, stellt diese Arbeit J-PRISM (Java Property-graph Repository-level Insight System for Meta-programming) vor, eine neue Architektur, die statische Git-Repositories in semantisch angereicherte Wissensgraphen (Knowledge Graphs, KG) transformiert. J-PRISM modelliert Code-Entitäten als Knoten und Abhängigkeiten als Kanten, wodurch einem KI-Agenten eine deterministische Schnittstelle zur Abfrage der Codebasis bereitgestellt wird. Das System implementiert eine hybride Retrieval-Strategie über das Model Context Protocol (MCP), welche symbolische Cypher-Abfragen für strukturelle Präzision mit vektorbasierter semantischer Suche zur Konzeptfindung kombiniert.

Zur Validierung dieses Ansatzes wurde ein neues Set an Benchmarks entwickelt, das speziell auf Java-Repositories mit Meta-Programming zugeschnitten ist und von einfachen CRUD-Anwendungen bis hin zu komplexen infrastrukturintensiven Architekturen reicht. Die empirische Evaluierung zeigt, dass J-PRISM herkömmliche dateibasierte Retrieval-Methoden (OpenHands) signifikant übertrifft und im Vergleich zu 89,37% bei der Baseline, einen mittleren Quality Score von 97,69% erreicht. Die Ergebnisse bestätigen ferner, dass mit zunehmender architektonischer Komplexität ein strukturiertes graphbasiertes Retrieval für ein präzises und effizientes Codeverständnis unerlässlich wird.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Recent advancements in Large Language Models (LLMs) have significantly impacted code generation. However, repository-level comprehension remains a critical challenge, particularly within enterprise Java environments that rely heavily on meta-programming frameworks such as Spring and Lombok. In these environments, control flow and logic are often defined by implicit annotations rather than explicit source code. As a result, standard text-based Retrieval Augmented Generation (RAG) approaches miss critical dependencies and retrieve irrelevant context.

To address these limitations, this thesis introduces J-PRISM (Java Property-graph Repository-level Insight System for Meta-programming), a new architecture that transforms static Git repositories into semantically enriched Knowledge Graphs (KG). J-PRISM models code entities as nodes and dependencies as edges, providing an AI agent with a deterministic interface to query the codebase. The system implements a hybrid retrieval strategy via the Model Context Protocol (MCP), combining symbolic Cypher queries for structural precision with vector-based semantic search for concept discovery.

To validate this approach, a new set of benchmarks specifically tailored to Java repositories using meta-programming was developed, ranging from simple CRUD applications to complex infrastructure-intensive architectures. Empirical evaluation demonstrates that J-PRISM significantly outperforms standard file-based retrieval methods (OpenHands), achieving a mean Quality Score of 97.69%, compared to 89.37% for the baseline. Results further confirm that as architectural complexity increases, structured graph-based retrieval becomes essential for accurate and efficient code comprehension.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
1.1 Background and Problem Statement	1
1.2 Research Objectives and Contributions	2
1.3 Methodological approach	2
1.4 Structure of this Thesis	3
2 Preliminaries and Related Work	5
2.1 Knowledge Graphs (KG)	5
2.2 AI Agents	10
2.3 Retrieval Augmented Generation (RAG)	12
2.4 State of the Art	14
2.5 Benchmarks	16
3 J-PRISM: Java Property-graph Repository-level Insight System for Meta-programming	21
3.1 System Overview	21
3.2 Knowledge Graph	22
3.3 Agent-Repository Interface	25
4 Benchmarks	31
4.1 Overview	31
4.2 Benchmark Definition	32
4.3 Evaluation Metrics	34
5 Evaluation	37
5.1 Executive Summary of Results	37
5.2 Experimental Setup	37
5.3 Response Quality Analysis	40
	xv

5.4	Efficiency and Cost Analysis	46
5.5	Ablation Study	54
5.6	Qualitative Analysis	57
5.7	Discussion	67
6	Conclusion and Future Work	69
6.1	Conclusion	69
6.2	Future Work	70
	Overview of Generative AI Tools Used	71
	List of Figures	73
	List of Tables	75
	Bibliography	77

Introduction

1.1 Background and Problem Statement

With the rise of large language models (LLMs), interest in new code-generation techniques has strengthened. The incentive to integrate the code-generation fully into the repository also emerged. The ability to fully automate every change request on a repository-level seemed promising, but new challenges appeared with this idea. Not only the requirements of the code change request, but also the knowledge over the repository needs to be added into the context window of the LLM. Currently, many AI agents, like *OpenHands* [58], obtain needed repository information via operations on the file system. Through trial and error, files are searched and are then parsed into the context of the agent. This approach has two major drawbacks. At first, it is not resistant to misspellings or vague information within the description of the prompt. Secondly, it often results in multiple retrieval steps, searching through the file system, until the needed information is found. This can lead to missing important knowledge and loading too much irrelevant information into the context.

Previous work has shown that the attention of an LLM decreases in the middle of the context window [31]. This effect is more severe when the context length is larger [31]. Consequently, keeping the context smaller and increasing the knowledge density should lead to better results in the code comprehension, and therefore also lead to a more improved code-generation.

One way to enhance the understanding of LLM is the usage of Knowledge Graphs (KG). The repository and its code structure can be mapped by a property graph. Using a domain-appropriate, schema-driven graph structure that models code-level entities and relations could improve the code-comprehension. This KG can then yield more targeted retrieval and better reflect program semantics than treating code as unstructured text.

As seen in Chapter 2, the current state of the art repository-level code comprehension and code generation approaches are focusing on simple Python repositories. Additionally, code comprehension of meta-programming techniques, like the Java framework *Spring* [21] or Lombok [41], have yet not been researched.

1.2 Research Objectives and Contributions

The primary objective of this thesis is the development and evaluation of a KG-powered AI agent. In particular, the KG will be designed to represent a *Java* repository which implements the *Spring* and the *Lombok* meta-programming framework.

The designed and implemented system is then used to investigate to what degree a KG can enhance the ability of an LLM to understand a repository-level codebase. We aim to answer the following research question (**RQ**) within this thesis:

RQ To what extent can a repository-wide Knowledge Graph enhance the code comprehension of a *Java* application using the *Spring Framework*?

To address the research question systematically, we further break it down into the following supportive questions (**SQ**):

SQ1 To what degree can the input and output tokens be reduced when retrieving the information from the KG instead of operations on the filesystem level?

SQ2 To what extent does the quality of the LLM response vary when using the KG enhanced LLM?

SQ3 How does the utilization of the KG correspond to agentic coding time?

SQ4 How significantly does the construction of the KG affect the result?

While we focus on the concrete case of *Java*, it is our goal to develop our approach and methods in a way that can naturally generalize to other similar meta-programming languages.

In addition to the supportive questions, we aim to design a new set of benchmarks to evaluate *Java* repository-level code comprehension, since no such benchmarks exist (*cf.*, Chapter 4).

1.3 Methodological approach

To achieve the before mentioned research objectives of this thesis we will design, implement and evaluate a KG enhanced LLM system. For that we will perform the following key steps:

Design KG The design of the KG will focus on abstracting the structure of a Java repository to individual nodes and their dependencies to each other. In contrast to simple RAG approaches, we will use a more complex KG schema which also reflects dependencies within the repository, e.g method calling and import dependencies.

Information Processing This KG can then be queried by the LLM through a Model Context Protocol (MCP) server. We design two different retrieval approaches, a simple *cypher* query tool and similarity search function. For the LLM AI agent we use openhands [58] as a baseline.

Design Benchmarks Currently, no benchmarks for *Java* repository-level code comprehension exist (*cf.*, Chapter 4). So, we will define a set of question-answer pairs for a number of repositories varying in their complexity. These will then be used to evaluate the presented system.

Evaluation To evaluate our approach we define two separate steps: The benchmarks tailored to different *Java* repositories will be processed and the metrics of the input and output tokens (*SQ1*), the runtime (*SQ3*) and the quality of the responses (*SQ2*) will be analyzed. In the second step an ablation study will be performed to validate how the different extensions effects the performance (*SQ4*). This ablation study will evaluate the contribution of each presented component and examines different ways of prioritizing the LLM’s system prompts toward using the KG.

1.4 Structure of this Thesis

Beyond this introduction, the thesis is organized into five main parts. In the following Chapter 2 we introduce core concepts such as KG, Retrieval Augmented Generation (RAG), and AI agents and give preliminaries. Furthermore, we provide research on existing literature and the current state of the art KG-powered RAG systems in the context of repository-level code comprehension.

Chapter 3 introduces a novel approach to generate an AI agent-based code comprehension system which is enhanced by a java repository focused KG. We discuss how the KG schema is designed and how the information can be retrieved by the LLM.

In Chapter 4 we present a new set of benchmarks which focus on testing the code comprehension of java repository using the *Spring* and the *Lombok* framework. After the generation of the benchmarks, we discuss the evaluation steps in Chapter 5. We begin to define the core setup and then analyze the result of the evaluation into detail.

Last but not least, Chapter 6 concludes the presented findings and provides recommendations for further research.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Preliminaries and Related Work

This chapter provides an overview of the needed background for this thesis. Each topic will review the basic concepts and the existing literature on the subject.

In the beginning, we focus on Knowledge Graphs (Section 2.1) and their different types. Then, in Section 2.2 we will discuss what AI agents are and how they can be structured and interact with external applications. After the core concepts of KG and AI agents, we will review common information retrieval types and how the LLM generation can be improved by Retrieval Augmented Generation (Section 2.3).

Section 2.4 will discuss the current state of the art of repository-level KG enhanced code generation and comprehension systems. In the end we dive into the topic of benchmarking LLMs and what the current benchmarks on repository-level code comprehension are (Section 2.5).

2.1 Knowledge Graphs (KG)

Although the term "knowledge graph" appears in the literature as early as 1972 [49], the modern notion of KGs became widely used after *Google* and other companies announced novel search engines that integrate a graph-based data model [25, 39, 20].

There is no single, universally accepted definition of knowledge graphs [7, 2, 13]. The literature offers a number of definitions, which are at times incompatible and vary from specific technical to more inclusive general [20]. But, within this thesis we want to align with the formulation presented by Hogan et al. (2021):

"Herein, we define a knowledge graph as a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities." [20]

This definition captures the essence of a KG as a structured, graph-based model of knowledge. The mentioned graph of data corresponds to the underlying graph-based data model which implements this abstract concept of a knowledge graph [20]. The data model can be different types of graphs. In the following, we will analyze a non-exhaustive list of concrete data models of the knowledge graph.

2.1.1 Data Model

There are numerous different data models used to implement KGs [20], but we want only to focus on the most popular models in practice [20, 3]. In the following we will explore Directed Edge-labeled (DEL), heterogeneous and property graphs.

Directed Edge-labeled Graphs

A directed edge-label (DEL) graph, sometimes also referred as a multi-relational graph [6, 8], is a simple and fundamental graph model. Such a graph can be seen in Figure 2.1. Each node is connected to another with a labeled edge. Formally, it is a graph where edges are directed, and each edge is associated with a label from a specific set of possible labels [20]. A DEL graph G can be defined as a set of triples (s, p, o) , where s (subject) and o (object) are nodes from a set of entities V , and p (predicate) is the label from a set of relations R , representing a directed edge from s to o [3].

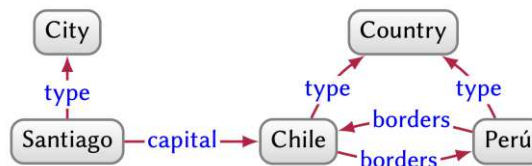


Figure 2.1: DEL graph [20]

Due to its simplicity, this model is not merely a theoretical construct but is ubiquitous in modern graph database systems and real-world knowledge graphs, such as Wikidata [61]. The most prominent and standardized instantiation of the DEL model is the Resource Description Framework (RDF), a $W3C$ standard [45]. In RDF each node can be one of three types: An Internationalized Resource Identifier (IRI), denoting a global identifiable entity and relations on the Web; a literal, used as simple datatype values like string, integers or dates; or a blank node, used to represent the existence of an entity [45, 20].

Heterogeneous Graphs

Heterogeneous graphs are similar to DEL graphs, but they extend the graph by assigning each node and edge a certain type [23, 57]. Therefore, the type is not presented by an additional relation, but is rather a direct part of the graph model [20]. Figure 2.2 shows a heterogeneous graph.

If an edge is shared by two nodes of the same type it is called *homogeneous* and

heterogeneous if not [20]. Unlike DEL graphs, heterogeneous graphs typically presume exactly one type (one-to-one relations between types and nodes) [20].



Figure 2.2: Heterogeneous graph [20]

Property Graphs

Property graphs extend the basic idea of labeled edges and heterogeneous graphs even further. Both nodes and edges are allowed not only to carry a label but also an arbitrary set of key-value properties [20]. Rather than using auxiliary nodes to store simple properties in a DEL graph, the property graph allows facts to be annotated directly to a node or an edge [20]. In practice, this modeling freedom has made property graphs popular in industrial graph databases (e.g., *Neo4j* [38, 3]), even though the model itself is not yet standardized and thus varies across systems [20]. This advantage of added flexibility however comes with a trade-off: property graphs typically need richer query languages and more intricate formula semantics than simple DEL or heterogeneous graphs [20].

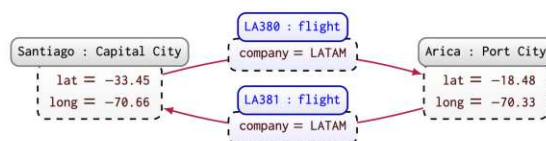


Figure 2.3: Property graph [20]

2.1.2 Retrieval

One core discipline of knowledge graphs is the retrieval of the stored information. There are many different ways to extract data from a knowledge graph: from exact, symbolic querying to similarity-based (vector) retrieval. Within this section we will investigate different retrieval processes. We will outline the most widely used query languages and then cover vector-space search and complementary retrieval forms.

SPARQL

SPARQL 1.1 is the *W3C* standard query language for RDF and operates primarily by matching basic graph patterns (BGPs), a set of triple patterns, against an RDF graph under multi-set semantics [51]. The language presents core constructs like

SELECT, CONSTRUCT, FILTER, OPTIONAL, UNION, MINUS, aggregation, grouping and subqueries [51]. In addition, property paths allow navigational queries using regular-expression-like operations to bind nodes connected via paths of arbitrary length [51].

The following Listing 2.1 shows a simple SPARQL query which extracts the information on a flight connecting two cities. Note that the prefix represents the full URI of the example RDF graph.

```
PREFIX ex: <http://example.org/air#>

SELECT ?flightNo ?fromName ?toName
WHERE {
  ?from a ex:CapitalCity ; ex:name ?fromName .
  ?to   a ex:PortCity     ; ex:name ?toName .

  ?f a ex:Flight ;
    ex:origin ?from ;
    ex:destination ?to ;
    ex:flightNo ?flightNo ;
    ex:company ?company .

  FILTER (lcase(?company) = "latam")
}
ORDER BY ?fromName ?toName ?flightNo
```

Listing 2.1: SPARQL query to retrieve information on the cities connected by a flight from LATAM company.

Cypher

Cypher on the other hand is a declarative pattern-matching language that originated in Neo4j and now adopted across multiple systems [3]. The pattern-matching queries are expressed with the MATCH clause followed by a pattern in "ASCII art" form, such as $(s) - [e] \rightarrow (n)$ where s represents a starting node, e the edge and n the end node [14]. Each node and edge in the query can then be extended by labels and/or properties to match entities within the knowledge graph. Cypher uses further SQL-similar clauses to fetch and format the information, e.g. SELECT, RETURN, UNION, and so on [14]. While this work utilizes Cypher, it is important to note that the industry is moving toward the ISO GQL (Graph Query Language) standard [24]. As GQL is heavily based on Cypher, the implementation logic presented below remains relevant and easily adaptable to the new standard.

Listing 2.2 presents the same previously mentioned SPARQL query using a Cypher query.

```
MATCH (s:Capital City)-[f:flight {company: "LATAM"}]->(a:Port City)
RETURN s.name AS fromName, a.name AS toName, f.flightNo
```

```
ORDER BY flightNo;
```

Listing 2.2: Cypher query to retrieve information on the cities connected by a flight from LATAM company.

Vector Space Retrieval

The aforementioned symbolic queries assume the exact identifiers and edges exist. In many applications we also need approximate retrieval, especially when dealing with vaguely defined user requests. To address imprecise or natural language queries, vector space retrieval represents queries and information as points in a high-dimensional vector space [20]. The nearest points in regard to the query are then retrieved. This approach descends from the classic vector space model in information retrieval, which already framed matching as geometric similarity in a term-weighted space [48].

Embeddings An embedding is a function $f : X \rightarrow \mathbb{R}^d$ that maps objects $x \in X$ to dense vectors. The objects can be for instance tokens, nodes, subgraphs and sentences. The semantic or structural similarity in X is reflected by proximity in \mathbb{R}^d . Early distributional embeddings were learned from co-occurrence statistics, e.g. *word2vec* [37]. Modern embeddings are typically derived from deep transformers and can represent sentences, passages, and code with strong semantic fidelity [11, 46]. For knowledge graphs specifically, knowledge graph embeddings (KGEs) learn representations for entities and relations that encode graph structure and relation semantics, e.g. *TransE*, *DistMult*, *ComplEx*, *RotatE* [8, 54, 52, 62].

Similarity measures When comparing now two vectors, one for the query element and one for an element in the knowledge graph, a similarity score will be determined. Given these two vectors $u, v \in \mathbb{R}^d$, the similarity score can be calculated differently. One way would be by calculating the cosine similarity:

$$\cos(v, u) = \frac{u \cdot v}{\|u\| \cdot \|v\|}$$

Another approach would be by calculating the Euclidean distance, which measures the straight-line distance between two vectors in the embedding space [8]:

$$d(u, v) = \|u - v\|_2 = \sqrt{\sum_{i=1}^d (u_i - v_i)^2}$$

While cosine similarity focuses on the angle between vectors, i.e. their directional alignment, Euclidean distance captures magnitude differences as well. In practice, cosine similarity is often preferred for textual or semantic embeddings, as it normalizes for vector length and emphasizes relational meaning rather than absolute scale [63].

In large-scale retrieval systems, these similarity computations are optimized through

approximate nearest neighbor (ANN) algorithms [36]. This enables efficient search in high-dimensional space [36]. Combined with embedding models, these methods form the backbone of modern semantic search, question answering, and retrieval augmented generation (RAG) systems. In Section 2.3 we provide a more detailed discussion of RAG systems.

2.2 AI Agents

In recent years, the notion of AI agents has moved from classical symbolic AI into the context of large language models (LLMs). While an LLM on its own is essentially a powerful conditional text model, an AI agent couples such a model with an environment in which it can perceive, reason, and act [47, 27]. In contrast to a single stateless prompt-response interaction, an agent is designed to operate over multiple steps, to maintain and improve the response. Additionally, the agent can select actions that affect external systems based on its evolving goals and observations [27].

Often systems are denoted as "agentic" [59] since many researchers see the definition of AI agents more as a spectrum than as binary [27]. The following three factors that cause systems to be more agentic were described by Sayash Kapoor et al. [27]:

- **Environment and goals:** The more agentic a system is the better it can handle a more complex environment and pursue more complex goals without external instructions on how to achieve the goal [50, 27, 15].
- **User interface and supervision:** The less supervision from the user and the more autonomously the AI can handle the task, the more agentic it is [27, 15, 50].
- **System design:** More agentic systems are using design patterns like tool/function calling, planning or dynamic control flow [27, 60, 56].

Building on these dimensions, AI agents can be seen as systems that embed an LLM into a control loop that observes an environment. The agent reasons about intermediate steps, and issues actions via tools or APIs, rather than producing a single reply to a prompt. A practical motivation for using agents is that they enable LLMs to solve tasks that exceed the context window. Recent surveys emphasize tool use, planning, and feedback learning as central design paradigms for LLM-based agents [30, 27, 56].

From an architectural perspective, many agent frameworks decompose the system into a small number of recurring components. A policy component interprets the current goal and decides on the next action, including tool invocation. The memory component preserves information across steps, including summarized prior episodes or user-specific knowledge. The environment is abstracted as a set of observable states and available actions. Although concrete implementations differ substantially, this general pattern of observe-reason-act-reflect is shared by many recent agent systems [30].

Within this thesis, AI agents are primarily relevant as an interface between LLMs and structured information sources such as knowledge graphs and code repositories. Instead of embedding all relevant information directly into the model prompt, an agent can decide when and how to retrieve additional context. In the following we discuss a concrete and increasingly adopted approach to structuring interactions between the AI agent and the environment.

2.2.1 Model Context Protocol (MCP)

To abstract the environment from the agent and to standardize the interaction between AI agents and external tools and resources, the Model Context Protocol (MCP) was introduced by *Anthropic* in 2024 [25]. Motivated by the Language Server Protocol (LSP) [19], MCP exposes tools and contextual data dynamically to LLMs and acts as a flexible framework for AI agents [22]. It even allows a human-in-the-loop mechanism [22].

At the core of MCP is a client-server architecture. The AI application acts as the MCP client, while one or more independent MCP servers implement the protocol and expose concrete capabilities [22]. An MCP server can wrap local components or remote services. The server advertises available tools, resources, and prompts in a standardized schema. It responds to requests in a structured JSON-RPC format [22]. The client discovers these tools, forwards the agent's tool calling request to the appropriate server, and returns the results back to the LLM as additional context [22].

In Figure 2.4 from Xinyi Hou et al. [22] the difference of tool invocation with and without MCP is shown. Without MCP each external tool needs to be connected independently to the agent. MCP on the other hand introduces an intermediate server layer, with the AI application acting as an MCP client. External tools still expose their own specific APIs, but they are now connected to the MCP server, which standardizes their capabilities and communication with the agent. This framework abstraction enables more scalable, flexible, and reusable tool connectivity [22].

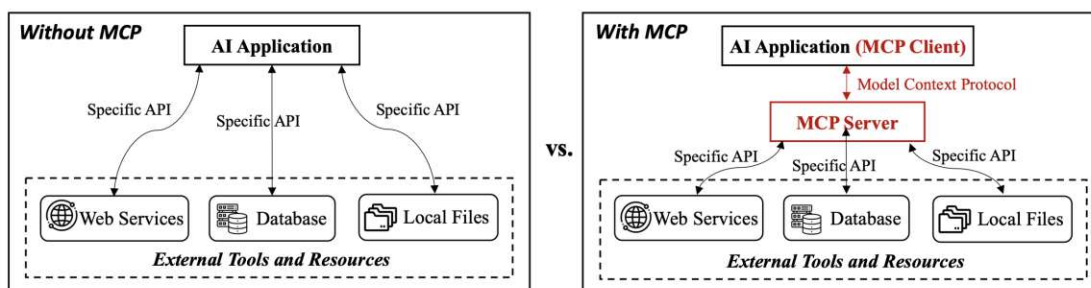


Figure 2.4: Tool invocation with and without MCP. [22]

2.2.2 OpenHands

One concrete instantiation of an LLM-based agent framework for software repositories is *OpenHands*, recently introduced by Wang et al. (2025) [58]. The central idea is to let agents interact with the code base the same way a human developer would [58]. It can inspect and edit files, run commands in the shell, execute code, and browse the web, all inside a controlled environment [58]. In contrast to purely API-based tool calling, *OpenHands* treats the operating system and the repository itself as the primary interface.

At the architectural level, *OpenHands* defines an agent as a function over a state [58]. The state bundles all information that is relevant for decision making, most prominently the event stream [58]. This includes the chronological history of past actions, their observations, user messages and metadata, such as cost accounting or multi-agent delegation traces [58]. Based on this state, the agent produces the next action that is executed in a docker-sandbox runtime [58]. Results of the action are then returned as observations and appended to the event stream [58].

For repository-level tasks, information retrieval is realized through this action space rather than a separate retrieval API. Each task spins up a dedicated docker container in which the target code base is mounted into a configurable workspace directory [58]. The agent can then list directories, open and inspect files, search for identifiers, and execute tests using ordinary shell commands [58]. In addition, Python actions provide a higher-level programmatic interface to the repository. This can be done, for example, by running analysis scripts or inspecting data structure during debugging [58].

It is worth noting that *OpenHands* was only evaluated with benchmarks of Python-repositories [58]. Furthermore, no meta-programming techniques within the repository, e.g. the Spring Framework, were considered and evaluated.

2.3 Retrieval Augmented Generation (RAG)

Large language models are powerful in-context reasoners but remain constrained by a fixed training corpus, a limited context window and a tendency towards hallucinated answers. Retrieval Augmented Generation (RAG) addresses these limitations by explicitly coupling a retriever with a generator. In a typical RAG pipeline, a user query is first mapped into a representation that can be matched against an external knowledge store [17, 16]. The retriever selects a small set of information that is estimated to be relevant for the query. These results are then concatenated with the original query and provide additional context to the LLM. This decoupling of retrieval and generation makes it possible to inject fresh and domain specific information at inference time without the need to retrain the model [17, 16].

Within the general RAG framework, different retrieval options can be categorized into the three paradigms of Naive, Advanced and Modular RAG [17]:

Naive RAG The simplest retrieve and read pipeline is used by the Naive RAG approach. The documents are preprocessed into chunks which are then embedded and stored in the vector space. At query time, the user’s question is encoded with the same model and used to retrieve the top chunks with the highest similarity. We explained this approach in more detail already in the Section 2.1.2. The retrieved information is then parsed into the LLM context. Naive RAG is easy to implement and often surprisingly effective, but still has notable drawback with precision, recall and information quality [17].

Advanced RAG To refine this basic pipeline Advanced RAG inserts additional processing steps before and after the retrieval step [17]. On the pre-retrieval step, indexing is improved through more careful chunking strategies such as sliding windows, finer segmentation and hierarchical indices [40, 64, 17]. Also, queries are rewritten, expanded or decomposed into sub questions so that the retriever operates on clearer and more informative inputs [34, 17]. After the retrieval, the candidate chunks are re-ranked, filtered and possibly compressed so that only the most informative evidence is presented to the LLM [28, 17]. So, Advanced RAG increases recall and precision without changing the core generator but by optimizing the retrieval process [17].

Modular RAG Modular RAG goes a step further and treats the retrieval step as one module in a larger, configurable architecture [17]. In addition to a retriever, the system can include dedicated search modules that call external APIs, knowledge graphs, routing modules, or many more. Retrieval itself can be iterative, recursive or adaptive rather than a single one shot operation [17]. This modular view allows different retrieval strategies to be composed and changed dynamically in the process [17].

Despite its success, RAG also introduces new challenges. The overall quality of the system is strongly bounded by the quality of the retrieval step [16]. If the retriever fails to surface the necessary evidence, the generator can not compensate and may still hallucinate confident but incorrect answers [16]. Aligning the behavior of the LLM with the retrieved context, e.g. by ensuring that it faithfully uses evidence instead of ignoring it, requires careful prompt design [17, 16].

Furthermore, feeding the entire information of a subject into the context can reduce the quality [31]. Currently, the largest context window spans from 10 [53] to 100 million tokens [1]. Previous work from Nelson F. Liu et al. [31] has shown that the understanding of the input context decreases in the middle. This accuracy drop is shown in Figure 2.5. So, it does not seem reasonable to ingest the total information available into the LLM context, but rather choose and filter the information before parsing. This token reduction also relates to the computational cost of a request.

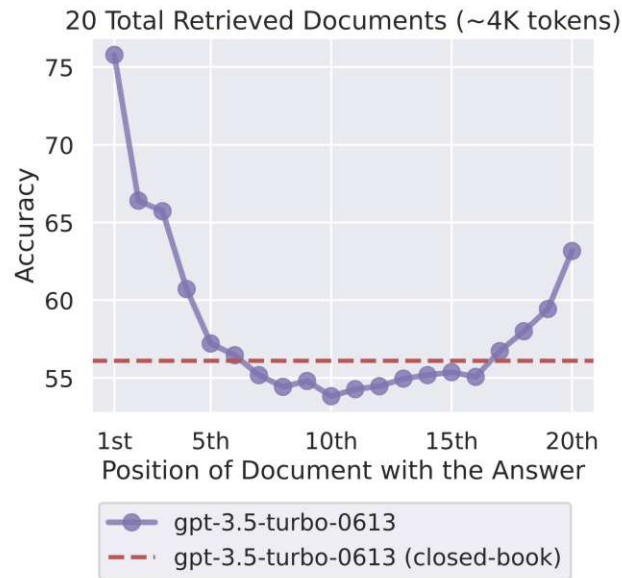


Figure 2.5: Changing the location of relevant information (in this case, the position of the passage that answers an input question) within the language model’s input context results in a U-shaped performance curve—models are better at using relevant information that occurs at the very beginning (primacy bias) or end of its input context (recency bias), and performance degrades significantly when models must access and use information located in the middle of its input context. [31]

2.4 State of the Art

Within this section we dive deeper into the current state of the art repository-level knowledge graph enhanced LLMs. We will discuss their architecture and possible improvements.

2.4.1 CodeXGraph - Liu et al. (2024)

A recent line of work that is closely related to our approach is *CodeXGraph* by Liu et al. [32]. The presented system connects an LLM agent with the code repository via a graph database interface [32]. Instead of relying on similarity search retrieval or on a predefined toolbox, *CodeXGraph* enables the LLM to interact with the knowledge graph through self-generated Cypher queries.

Knowledge Graph Structure The core of the architecture is a schema-driven code graph. A task-agnostic schema is used in which nodes represent code symbols such as MODULE, CLASS, FUNCTION, METHOD, FIELD and GLOBAL_VARIABLE. Edges encode structural relations between these symbols, including CONTAINS, HAS_METHOD, HAS_FIELD, INHERITS and USES. Additional information is stored within these entities’

metadata. The schema is tailored to Python repositories only. Python files are translated into nodes and edges and are then stored in a *Neo4j* [38] database. This schema enables multi-granularity navigation, from modules down to individual fields, while keeping code snippets stored outside the graph.

Retrieval The interaction between the KG and the LLM is organized as an iterative pipeline. Rather than trying to solve the task in a single step, the agent repeatedly issues one or several graph queries. As already mentioned the LLM designs Cypher queries which are then executed on the graph.

The system was evaluated by three different Python repository-level benchmarks. It achieves competitive performance across all three tasks, especially when combined with stronger models. At the same time, the authors note that the current implementation is limited to python [32]. Furthermore, a more comprehensive schema and construction efficiency are potential improvements [32].

2.4.2 Knowledge Graph Based Repository-Level Code Generation - Athale et al. (2025)

A further recent approach that connects repository-level generation with knowledge graphs is the framework presented by Athale et al. [5]. Unlike approaches that rely solely on vector-based similarity or agentic query generation, this method focuses on constructing a dense, semantically rich representation of the repository to enable precise subgraph retrieval [5].

Knowledge Graph Structure The architecture relies on a property graph that models the code base at a fine-grained level. The schema is designed to capture not only the syntactic structure but also the semantic dependencies between code elements. The graph schema can be seen in Figure 2.6. The construction of the graph is typically automated using static analysis tools. A distinguishing feature of this approach is the generated description of the class, method and function definition. Code snippets are described by an additional LLM and are then inserted into the knowledge graph to enhance the contextual depth and make the KG more informative [5]. This generated description and the documentation are then used to calculate the vector embeddings with an embedding model. These embeddings can then be used for the similarity search retrieval. [5]

Retrieval Athale et al. designed a hybrid code retrieval system. The system can handle a query in natural language. The LLM can generate Cypher queries from the natural language query along with the knowledge graph schema. Alternately, the embedding model mentioned before can generate embeddings from the natural language query which can then be used to perform a similarity search on the knowledge graph. The results of both retrieval steps can then be expanded by traversing the knowledge graph or can be filtered by prioritizing the top k nodes. This step is used to improve computational

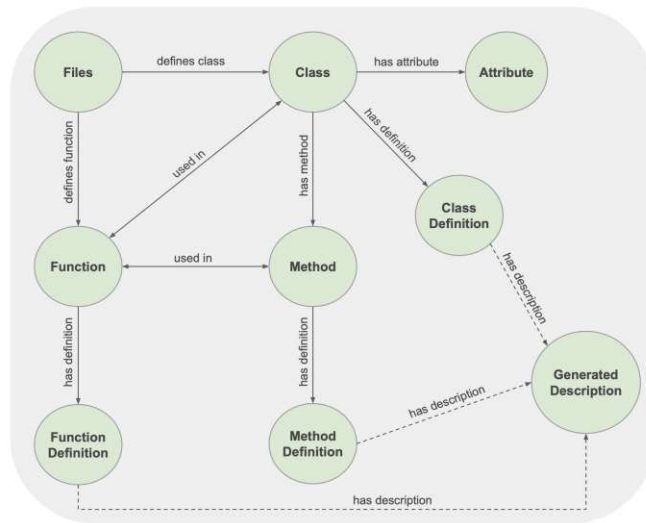


Figure 2.6: A sample schema that defines the node types and relations between them. Every node in the knowledge graph will be one of these types linked by a relevant relation to another node. [5]

efficiency and to enrich the context provided to the LLM. [5]

The evaluation shows that using a repository-level knowledge graph to enhance LLM-based code generation improves results compared to standard prompting and local file retrieval. The authors test their approach on the *EvoCodeBench* [29], which covers function- and method-level code generation in real Python repositories [5]. On these benchmarks, their system achieves higher accuracy than the original *EvoCodeBench* baseline, which depends on local file completion or no retrieval [5]. Furthermore, the approach from Athale et al. outperformed *CodeXGraph* slightly when both use a strong LLM [5]. This suggests that a richer knowledge graph and a hybrid retrieval strategy can provide more relevant context to the model and lead to more correct and repository-consistent code.

To conclude, Athale et al. show that a repository-wide property graph with hybrid symbolic-semantic retrieval can substantially improve code generation. At the same time, their work remains limited to a Python-only schema and does not yet address meta-programming frameworks.

2.5 Benchmarks

Proper evaluation of Knowledge Graph-enhanced AI agents on a repository level requires robust benchmarks. Traditional datasets like *HumanEval* focus on generating single functions in isolation [9]. However, real-world software engineering requires understand-

ing an entire repository. AI models must navigate cross-file dependencies, understand complex class hierarchies, and respect architectural patterns. This section reviews the state-of-the-art benchmarks for this domain. These repository-level benchmarks test an AI model’s ability to understand the codebase. They are essential for measuring the performance of RAG systems and agentic systems.

2.5.1 EvoCodeBench - Li et al. (2024)

A significant advancement in this domain is *EvoCodeBench*, introduced by Li et al. (2024) [29]. This benchmark was developed to address the misalignment of existing evaluation datasets and real-world repositories. Unlike previous benchmarks that primarily feature standalone functions using only built-in libraries, *EvoCodeBench* is constructed from comprehensive real-world repositories [29]. This ensures that the distribution of code types mirrors actual development practices.

EvoCodeBench specifically distinguishes between standalone and non-standalone code. As shown in the benchmark’s analysis, a standalone function solely uses built-in or public libraries, while a non-standalone one contains context-aware dependencies [29]. These context-aware dependencies are invocations of code elements defined within the current repository. This forces the LLM to understand the broader codebase rather than just the local function scope. Three distinct levels of dependencies are defined by *EvoCodeBench* which the AI agent must successfully resolve [29]:

- **Intra-class** dependencies: Methods or fields are referenced within the same class.
- **Intra-file** dependencies: Methods, fields or classes are referenced within the same file but not class.
- **Cross-file** dependencies: Methods, fields or classes are referenced which requires accessing code defined in different files across the repository.

To evaluate performance on these tasks, *EvoCodeBench* utilizes two primary metrics: Pass@k and Recall@k [29]. Pass@k measures functional correctness by executing test cases to check if the generated code behaves as expected [29]. Recall@k, on the other hand, specifically evaluates the model’s ability to identify and invoke the correct reference dependency from the repository context [29]. Furthermore, to prevent the model from simply memorizing open-source code seen during the training process, *EvoCodeBench* is designed as an evolving benchmark. It automatically updates the dataset using the latest repositories every few months [29].

One major limitation of the *EvoCodeBench* dataset is that it is monolingual, consisting only of Python repositories [29].

2.5.2 Java Specific Benchmarks

Python dominates the landscape of code generation benchmarks due to its dynamic nature and popularity in AI research. However, the Java ecosystem has seen the formation of specialized repository-level evaluation frameworks. These benchmarks attempt to capture the verbose, strictly typed, and object-oriented nature of Java development.

A prominent multilingual benchmark is the dataset *CrossCodeEval* by Ding et al. (2023) [12]. The benchmark explicitly targets cross-file completion [12]. Unlike benchmarks that allow models to guess code based on local context alone, *CrossCodeEval* uses static analysis to identify and mask samples where the completion strictly requires information from other files in the repository [12]. It includes a significant subset of Java repositories, testing whether an LLM can correctly import and utilize classes defined elsewhere in the project [12]. The evaluation relies on an exact match of the completed code and identifier matching to verify if the model successfully retrieved the correct cross-file context [12].

Another relevant and more recent contribution is *RepoClassBench* by Deshpande et al. (2024), which has been introduced to evaluate class-level generation [10]. In Java, where the class is the fundamental unit of logic, evaluating a single method often fails to capture the complexity of inheritance. *RepoClassBench* tests the ability of an LLM to generate entire classes that adhere to interfaces and inheritances defined in the provided repository [10]. The dataset includes multiple code repositories using Java, Python or C#. Within these repositories 130 Java classes of real-world repositories are available [10]. The performance of the AI agent is measured using three specific metrics [10]:

- **Pass@K** indicates the percentage of tasks where the model generates a correct solution in their first K attempts.
- **Test Rate** measures the average fraction of test cases passed across all generated samples.
- Finally, the **Compilation Rate** tracks the percentage of tasks where the code compiles successfully. This checks for syntactic correctness, which is critical in strongly typed languages like Java.

2.5.3 Meta-Programming Benchmarks

Despite the progress in repository-level benchmarks, a critical gap remains in the evaluation of enterprise-style Java development. Modern Java applications heavily rely on meta-programming and framework-specific abstractions, most notably the Spring framework and Lombok. These technologies shift logic from explicit code to compile-time generation. Thus creating a "magic" layer that current benchmarks fail to capture. For example, existing benchmarks rarely test if an LLM understands that it should not write a getter method because the class is annotated with `@Data` from the Lombok library

which dynamically generates the getter methods at compile time [41]. The same holds true for the dependency injection used within the Spring framework [21]. Consequently, while LLMs are frequently used to write enterprise Java code, their state-of-the-art performance metrics in this specific domain remain largely unmeasured.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

J-PRISM: Java Property-graph Repository-level Insight System for Meta-programming

To address the research question of enhancing code comprehension for Java repositories using meta-programming frameworks, we developed the Java Property-graph Repository-level Insight System for Meta-programming (J-PRISM). This chapter describes the methodological approach taken to design and implement the system. The architecture is divided into two primary components: the Knowledge Graph, which serves as the structured knowledge base of the repository, and the Agent-Repository Interface, which acts as the communication layer between the LLM agent and the stored knowledge of the repository.

3.1 System Overview

The J-PRISM system is designed as a modular pipeline that transforms a static Git repository into a dynamic, queryable knowledge graph. As seen in Figure 3.1, the architecture functions as a middleware layer between raw source code and an autonomous AI agent. The design is divided into three logical phases: ingestion, storage, and interaction.

The workflow begins on the left side of the figure with the ingestion layer. The system targets remote Git repositories. It initiates by cloning the repository. Once the files are retrieved, they enter the central J-PRISM processing unit. Here, a specialized Extract-Transform-Load (ETL) process acts as the ingestion engine. This component parses the raw Java files and applies preprocessing steps. This process ensures that meta-programming constructs are fully expanded before static analysis occurs and is also responsible for semantical enrichment.

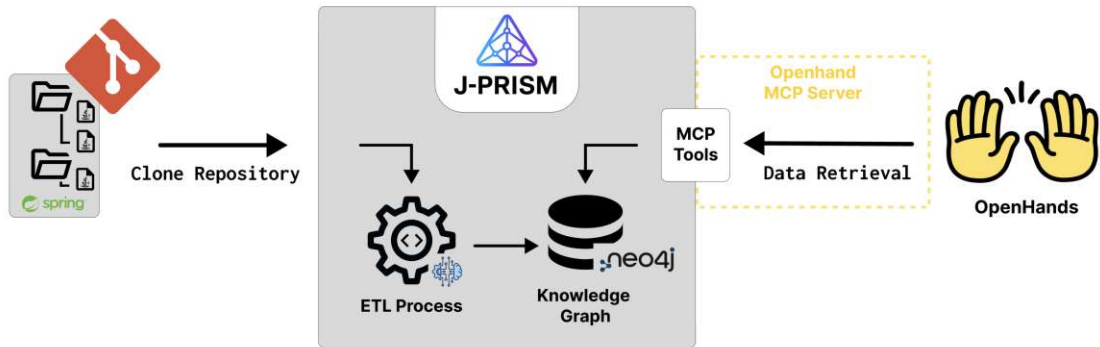


Figure 3.1: The architecture of the J-PRISM system.

Following the transformation, the data is loaded into the Knowledge Graph. The system utilizes a *Neo4j* graph database for this storage layer. Unlike traditional relational databases, the graph model natively stores the complex interlinked dependencies inherent in Object-Oriented Programming (OOP). This allows the system to persist not just the code syntax, but also the semantic relationships between classes and methods.

The final phase, shown on the right side of the architecture, simplifies the interaction with the AI agent. Communication is governed by a MCP server already integrated and designed by OpenHands. This server acts as an interface that exposes specific retrieval tools to the outside world. The AI agent is represented by OpenHands, which connects to this server as a client. Through this connection the agent executes data retrieval tools to pull precise structural data or perform semantic searches.

3.2 Knowledge Graph

The core of the proposed system is a domain-specific Knowledge Graph designed to abstract the structure of a Java repository into individual nodes and dependencies. Unlike simple RAG approaches that treat code as unstructured text, this system utilizes a Labeled Property Graph (LPG) to model code-level entities and relations.

3.2.1 Graph Schema

The graph schema is strictly typed to represent the hierarchical and relational nature of OOP in Java. The data model follows the Property Graph paradigm, allowing both nodes and edges to store key-value properties. The schema centers around four primary node types that capture the essence of the code base. The *CodeClass* node represents Java classes and contains properties such as the class name, access modifiers, package name, class type (e.g. *interface* or *enum*), and associated documentation. Within this classes, the executable logic is encapsulated in *CodeMethod* entities. These store the

method name, return type, modifiers, input parameters, possible exceptions, method documentation, and the actual source code itself. Data fields within the *CodeClass* are represented by *Variable* nodes, which hold details regarding the variable's name, type, modifier, and initial value. Finally, to specifically support the *Spring* framework and *Lombok* targeted by this thesis, *Annotation* nodes are explicitly modeled to capture Java annotations, storing the text of the annotation as a property.

Note that the prefix *Code* was adopted for the method and class entities to avoid conflicts with reserved keywords in Java and Neo4j.

As seen in Figure 3.2, these mentioned entities are connected by directed relationships, which map the logical flow and dependencies of the application. Structural containment is defined by relationships such as *HAS_METHOD*, *HAS_VARIABLE*, and *HAS_ANNOTATION*, which link classes to their constituent parts. Dependency relationships, including *EXTENDS*, *IMPLEMENTS*, and *IMPORTS*, map the inheritance hierarchy and external references between classes. Furthermore, the execution flow is captured through the *IS_CALLING* relationship, which maps method invocation chains. This connection between methods allows the graph to represent not just the static structure but also the dynamic behavior of the program.

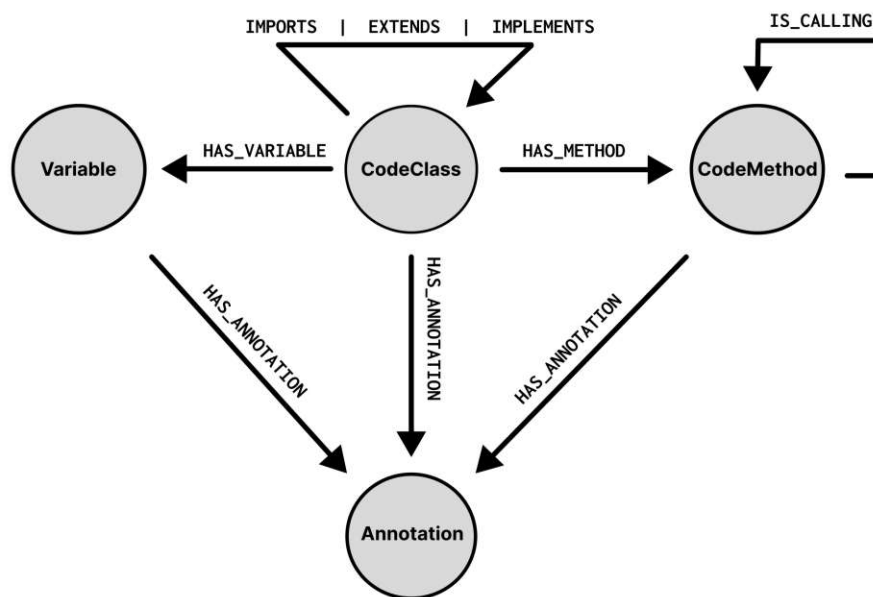


Figure 3.2: Architecture of the main nodes in the KG.

3.2.2 Data Integration

To populate this graph, we implemented an application that functions as a control system to clone, parse, and evolve the KG. The ingestion pipeline follows a strict ETL process. The process begins with the cloning of a remote Git repository. Following this, the system addresses the specific challenges of meta-programming through a preprocessing step involving Lombok. A "delomboked" version of the repository is generated to ensure that dynamically generated code, such as *getters*, *setters* and *constructors*, is explicitly visible to the parser.

The core parsing is handled by the *JavaParser* [26] library, which generates an Abstract Syntax Tree (AST) of the repository and the "delomboked" version. To support accurate symbol resolution, particularly for the Spring framework, we configured the parser with a *TypeSolver* of the *JavaParser* library. By referencing the project's dependency JARs, the solver allows the system to correctly identify types and hierarchies that are defined externally in libraries rather than the source code.

The system then traverses this AST to extract classes, methods, and variables. This data will be mapped directly to the schema defined previously. During the insertion of data into the graph, logical constraints are strictly enforced to maintain data integrity. For instance, if a class definition indicates that it extends another class, the system validates this connection and creates the corresponding *EXTENDS* or *IMPORTS* relationships.

Figure 3.3 shows the Knowledge Graph of an example java repository. The first 300 nodes are represented.

3.2.3 Semantic Enrichment

A raw syntactic graph often lacks the semantic density required for effective LLM retrieval. To address this, we implemented an "Evolution" phase that enriches the graph using Artificial Intelligence. One major limitation in legacy repositories is the lack of meaningful documentation and thus limited semantic information. To mitigate this, the system identifies methods missing a *Javadoc* comment and utilizes an LLM to generate synthetic documentation. This process uses a One-Shot prompt to produce summaries, parameter descriptions, and return value details based on the method's source code. For classes, the previously generated documentation of methods and the information on the variables are used to generate the class documentation. The documentation is then stored as properties to each *CodeMethod* and *CodeClass* node.

In addition to textual enrichment, the system generates vector embeddings to support semantic search. For every *CodeClass* and *CodeMethod* node an embedding will be calculated and stored within the KG. These embedding vectors are then stored into separate nodes within the current KG schema as seen in Figure 3.4. For *CodeMethod* nodes, all properties are serialized into a structured JSON string. This composite representation is then processed by an embedding model to yield a high-dimensional vector. This vector

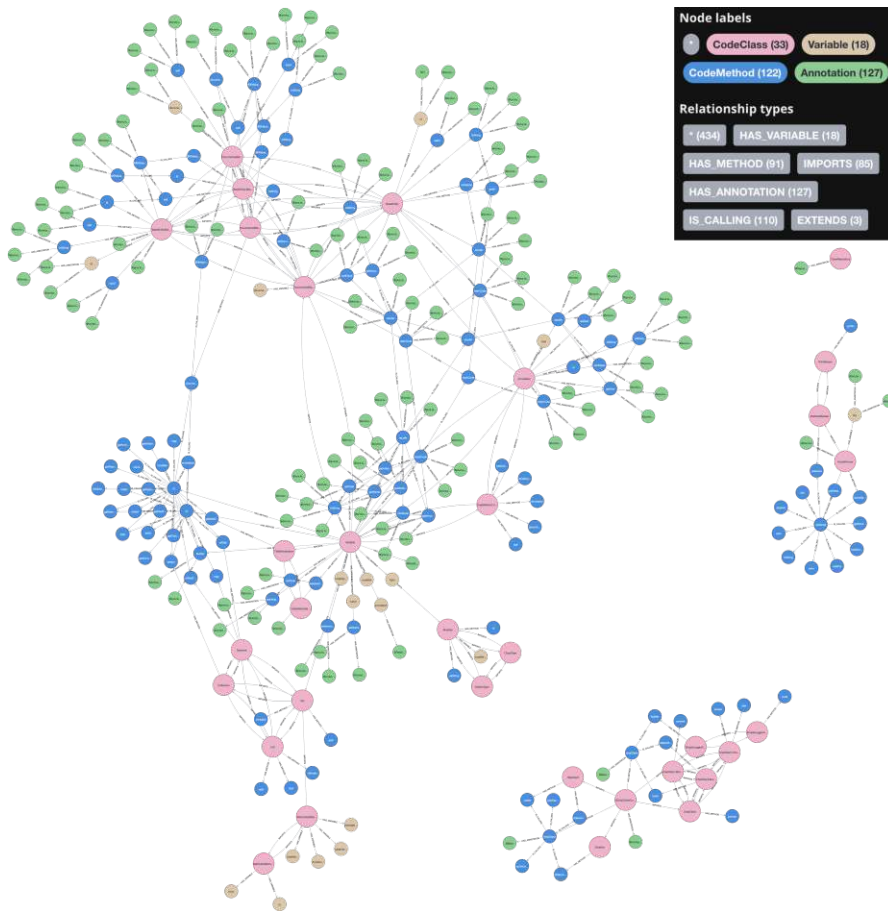


Figure 3.3: Subgraph of the first 300 nodes of an example Java repository using the *Spring* framework and *Lombok*.

captures both the syntax and semantics of the method. On the other hand, *CodeClass* embeddings are derived from the class-level documentation alongside the signature and documentation of its constituent methods.

3.3 Agent-Repository Interface

Once the Knowledge Graph is constructed and semantically enriched, it transitions from a passive storage unit to an active knowledge base for the AI agent. The Agent-Repository Interface is the critical abstraction layer that bridges the gap between the stochastic nature of the LLM and the deterministic structure of the repository data. In this section we will describe the communication protocol and the hybrid retrieval strategy. This retrieval strategy is designed to overcome the limitations of flat file-system search by

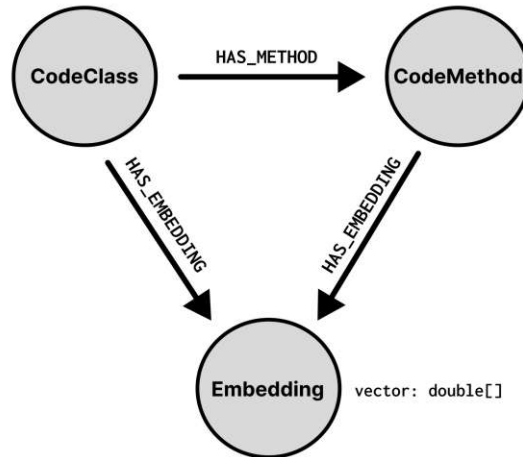


Figure 3.4: Integrate vector representation into existing schema

providing structured, queryable access to the code base.

3.3.1 MCP Integration

To standardize the interaction between the autonomous agent and the J-PRISM backend, we leverage the MCP standard. Rather than implementing a proprietary server architecture, we extended the existing MCP server architecture provided by the OpenHands framework.

In this configuration, OpenHands operates as the client, while the extended MCP instance acts as the host for J-PRISM’s retrieval tools. This integration strategy offers two distinct advantages: By registering J-PRISM’s capabilities as function tools within the native OpenHands environment, we reduce architectural friction and latency. And secondly, the agent interacts with the repository through high-level function calls, abstracting away the underlying complexities of database connection management and query parsing.

This system exposes specific executable functions defined within the MCP server that allow the agent to pull data from the *Neo4j* graph on demand.

3.3.2 Hybrid Retrieval Strategy

A strictly code-based search is often insufficient for high-level comprehension tasks, while a purely semantic search lacks precision for structural analysis. To address this duality, J-PRISM implements a hybrid retrieval strategy. Thus combining symbolic and semantic search mechanism.

Cypher Query Tool Description

Run a cypher query against the repository knowledge of classes, methods, variables, etc.
IMPORTANT: Use the following schema when defining the query!

```

-- RELATIONS --
(:CodeClass) -- [:EXTENDS] -> (:CodeClass)
(:CodeClass) -- [:HAS_ANNOTATION] -> (:Annotation)
(:CodeClass) -- [:HAS_METHOD] -> (:CodeMethod)
(:CodeClass) -- [:HAS_VARIABLE] -> (:Variable)
(:CodeClass) -- [:IMPORTS] -> (:CodeClass)
(:CodeMethod) -- [:HAS_ANNOTATION] -> (:Annotation)
(:CodeMethod) -- [:IS_CALLING] -> (:CodeMethod)
(:Variable) -- [:HAS_ANNOTATION] -> (:Annotation)

-- ENTITIES --
CodeClass: Properties: [packageName, name, classType, enumValues,
isExternal, modifier, documentation]
Annotation: Properties: [text]
CodeMethod: Properties: [methodName, isExternal, possibleExceptions,
modifier, documentation, inputParameters, returnType, code]
Variable: Properties: [documentation, name, type, initialValue,
modifier]

-- RELATIONS --
EXTENDS: Properties: []
HAS_ANNOTATION: Properties: []
HAS_METHOD: Properties: []
HAS_VARIABLE: Properties: []
IMPORTS: Properties: []
IS_CALLING: Properties: []

--END OF SCHEMA--

```

Figure 3.5: Cypher query tool description which is visible to the LLM. The relations are presented in a "ASCII art" format which is used by the Cypher language.

Symbolic Search For precise, structural queries, the agent can utilize a Cypher query endpoint. This endpoint is provided through a MCP tool. The LLM can generate and execute raw Cypher queries directly against the KG data base. This method leverages the strict logical structure of the graph, making it ideal for answering questions regarding inheritance or specific method annotations.

A critical component of this integration is schema awareness. To prevent the agent from "guessing" the database structure or hallucinating node labels, we integrated the KG schema directly into the tool description. All available nodes, properties and relations are presented to the LLM in a simple format. This ensures that the LLM is aware of the schema when generating new Cypher queries and the produced queries are syntactically correct. The description of the tool presented to the LLM can be seen in Figure 3.5.

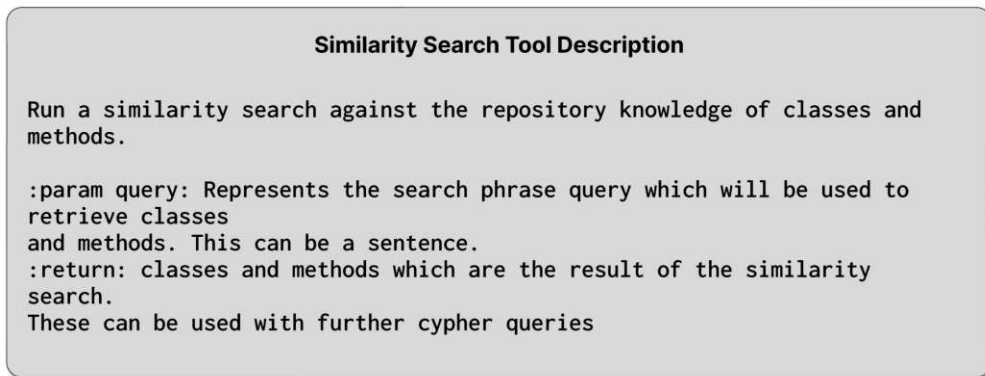


Figure 3.6: Similarity search tool description which is visible to the LLM.

Semantic Search To retrieve unknown or vague concepts the system offers a MCP tool to run a similarity search against the stored vector space. This tool addresses scenarios where the agent needs to find code related to a concept, such as "parsing Java files", without knowing the exact method names. The mechanism works by converting a natural language query into a vector using the same embedding model used to embed the existing repository (cf., Subsection 3.2.3). The LLM must formulate natural language queries derived from the user request. The description which will be displayed to the AI agent can be seen in Figure 3.6.

The execution of the semantic search relies on *Neo4j*'s internal vector indexing mechanism. When the LLM generates a natural language query, it is compared against the stored knowledge graph embedding entities using cosine similarity. This metric was chosen for its effectiveness in high dimensional space. This allows the system to retrieve repository artifacts that are conceptually aligned with the query, even if they lack lexical overlap.

Since the embeddings are generated for both Class and Method nodes, the retrieval mechanism is designed to be type-agnostic. It produces a heterogeneous result set that may contain a mix of classes and methods, rather than being restricted to a single structural level. This ensures that semantically relevant information is captured regardless of whether it is encapsulated within a class or a method. Thereby allowing the LLM to access the precise level of granularity required by the user's request.

To manage the LLM's comprehension and context load, we implemented a retrieval limit ($k = 5$). By returning only the top-5 results, we ensure that the agent is provided with high-precision context without being overwhelmed by irrelevant information. The search results are returned as structured JSON object, where each entity is paired with its calculated similarity search. This score serves as a confidence metric, enabling the agent to evaluate the relevance of the return.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Benchmarks

4.1 Overview

The evaluation of repository-level code comprehension has historically been biased toward languages with explicitly defined logic, with a particular emphasis on Python. As discussed in Chapter 2, current state-of-the-art benchmarks such as EvoCodeBench provide robust frameworks for evaluating Python code generation [29]. However, these existing datasets fail to address the unique challenges posed by enterprise Java development. Modern Java ecosystems heavily utilize meta-programming frameworks. Spring Boot and Lombok for example introduce a layer of abstraction that effectively hides the application’s control flow and data structure from standard static analysis.

Rather than explicit constructor calls or instantiation logic, the Spring framework relies on dependency injection where the connections between classes are defined by annotations. Similarly, in a standard Java file utilizing Lombok, for instance, getter and setter methods are never explicitly written in the source text. They are generated at compile time via the `@Data` annotation. A standard RAG system, which relies on text-based similarity search, often fails to retrieve the dynamically generated logic because relevant keywords simply do not exist in the raw file content.

To bridge this gap and answer our primary research question (RQ), we designed a novel set of benchmarks specifically tailored to evaluate code comprehension in a Java repository that rely on these meta-programming techniques. Since no such standardized benchmarks currently exist for this specific domain, we constructed a custom evaluation dataset. This set consists of question-answer pairs derived from real-world open-source repositories. These pairs are designed to evaluate the agent’s ability to navigate the layer of the frameworks, thus verifying whether the Knowledge Graph successfully exposes the implicit dependencies that standard text retrieval misses.

The benchmarks consist of natural language queries that a developer might ask when onboarding to a new codebase, paired with the ground-truth answers derived from an analysis of the code. Unlike generation benchmarks that rely on unit test execution, our focus is on comprehension and retrieval accuracy. Therefore, the evaluation is currently performed manually.

4.2 Benchmark Definition

To ensure the robustness of our evaluation, we selected three distinct open-source Java repositories. These repositories were chosen to represent a spectrum of complexity, ranging from simple Create-Read-Update-Delete (CRUD) operations to complex, infrastructure-heavy enterprise systems. This diversity allows us to test the *J-PRISM* system against the standard file-based retrieval tools that serve as the current default for coding agents, such as *OpenHands*.

4.2.1 Repository Archetype and Selection

Archetype I: Standard Layered CRUD Architecture The first repository category represents the foundational baseline of modern Java web development. Applications falling into this category strictly use the Model-View-Controller (MVC) design pattern, specifically the *Controller-Service-Repository* layering common in Spring Boot. The control flow is deterministic and linear, characterized by low complexity and standard relational database interactions.

The first selected benchmark of this archetype is the `springboot-blog-rest-api` repository by RameshMF [44]. This repository functions as the control variable for the experiment. Its primary purpose is to establish a baseline qualification. By querying this codebase, we verify the agent's ability to resolve standard syntactic structures, handle basic Java Persistence API (JPA) relationships, and identify explicit REST annotations without hallucination. Failure at this level would indicate fundamental defects in the information retrieval rather than limitations in the LLM's reasoning capabilities.

Archetype II: Domain-Centric Application This category shifts the complexity from the technology stack to the business domain. Such repositories are defined by "Logic Heavy" implementations where the primary challenge lies in resolving inter-dependent business rules rather than configuration management. These systems typically implement strict external specification. Thus, complex data flow traversals across multiple domain entities are needed to determine system behaviors.

This archetype is represented by the `spring-boot-realworld-example-app` by `gothinkster` [18]. This repository is utilized to evaluate the inferential capabilities of the agent. Unlike CRUD applications, the logic here is defined by state-dependent

visibility. To achieve successful outcomes the retrieval system is required to expose the semantic dependencies between entities, enabling the agent to reason about the received information.

Archetype III: Infrastructure-Intensive Architecture The final archetype represents high-density enterprise architecture. These applications are characterized as "Stack Heavy," featuring a high degree of "Dependency Density". The complexity originates from the integration of multiple disparate infrastructure components within a single monolithic codebase. Configuration is frequently decoupled from the business logic, relying heavily on implicit wiring.

This category is represented by the `mall-tiny` repository by `macrozheng` [35]. This repository serves as a stress test for context retention and cross-boundary linking. The evaluation metrics here focus on whether the system can understand and model the "invisible" infrastructure often obscured by annotations and configuration classes.

4.2.2 Comprehension Areas

To systematically analyze the performance of the *J-PRISM* system, the benchmarks are further sub-divided into four specific domains of code comprehension.

Annotation Comprehension This domain assesses the system's capacity to resolve the "magic" inherent in frameworks like Spring and Lombok. Since these frameworks rely on compile-time generation and runtime reflection, the logic is not present in the raw text. Questions in this category evaluate whether the system successfully models annotations as functional nodes. It also validates if the agent can identify implicit behaviors such as auto-generated methods or dependency injection targets.

Functionality and Execution Flow This category evaluates the agent's ability to simulate the runtime call stack. The questions require the agent to trace the execution path of specific user requests through the business logic. Furthermore, it tests the agent's understanding of general functionality of certain methods and what their purpose is.

Method Identification and Specificity This focus area targets the precision of the retrieval mechanism. Beyond simple keyword matching, these tasks require the agent to locate methods based on their semantic utility rather than their identifier. This validates if the agent can distinguish between syntactically similar but semantically distinct functions.

Structural and Architectural Insight The final category validates the agent's holistic understanding of the software architecture. These tasks move beyond file-level analysis to query the structural relationships of the entire repository. The agent must

accurately map class inheritance hierarchies, identify global components, and navigate package structures.

4.3 Evaluation Metrics

To quantitatively and qualitatively assess the effectiveness of a Java code comprehension system, we established a set of evaluation metrics. These metrics are designed not only to measure the correctness of the agent’s output but also to analyze the computational efficiency. For each question-answer pair in the benchmark suite, we capture five distinct data points:

- Input Tokens
- Output Tokens
- Execution Time
- MCP Tool Usage
- Response Quality Score

This multi-dimensional approach allows us to answer our supportive research questions (SQ1, SQ2, and SQ3) regarding cost, speed and accuracy.

4.3.1 Efficiency Metrics

Input and Output Tokens Within this benchmark suite the number of tokens consumed by the prompt and the generated output by the model will be measured. This metric is crucial for evaluating the efficiency of the system. Note that the retrieved information from MCP tools will count as input tokens since the LLM will receive the information as context in the input. In standard RAG or full-context approaches, the entire relevant file content is often dumped into the context window. This leads to high token usage. By monitoring these metrics, we can determine if a novel approach allows more concise context injection. A reduction in input tokens without a loss in quality would indicate a higher information density. This would validate the hypothesis that the J-PRISM approach provide a more efficient context to the LLM than raw file retrieval.

Execution Time The runtime is measured from the moment the agent receives the initial prompt to the moment it delivers the final answer (SQ3). This includes the latency of the LLM, the network overhead of the MCP server, and the query execution time against the KG. Tracking this metric helps us understand and validate a possible time trade-off between reasoning enabled by the graph and by a solely raw file injection.

MCP Tool Usage To quantify the agentic nature of the session, we record the integer count of MCP tool calls made during a single task. This metric simply counts how many times the agent felt the need to perform a query on the KG or the file system. A higher number suggests an iterative discovery process. Whereas a lower number might indicate a more powerful, complex query that retrieves precise data.

4.3.2 Response Quality

The most critical metric is the Quality Score, which measures the functional correctness and completeness of the answer (SQ2). Unlike binary pass/fail metrics used in code generation, comprehension tasks require a more nuanced scoring system. We employ a precision-recall based scoring logic, normalized to a percentage between 0% (0.0) and 100% (1.0).

The score is calculated by comparing the set of information items provided by the agent (I_{gen}) against the set of ground-truth items required by the benchmarks (I_{target}). The missing-information error is defined as:

$$E_{miss} = |I_{target} \setminus I_{gen}|$$

This E_{miss} error represents the number of required items not retrieved by the agent. The hallucination error is defined as:

$$E_{hall} = |I_{gen} \setminus I_{target}|$$

This E_{hall} error counts the number of generated items not present in the ground-truth. The total error is given by the cardinality of the symmetric difference:

$$E_{total} = |I_{gen} \Delta I_{target}| = E_{miss} + E_{hall}$$

Figure 4.1 shows this symmetric difference. Where the target set I_{target} and the generated set I_{gen} do not overlap will be counted as error. Consequently, the error increases additively by one for each missing or hallucinated information item. This error is then used to compute a score which is relative to the absolute number of elements in the ground-truth. The Quality Score Q is defined as:

$$Q = 1 - \min\left(\frac{E_{total}}{|I_{target}|}, 1\right) \in [0, 1]$$

The minimum function ensures that the Quality Score can not decrease beyond 0. Therefore, if the generated output of the agent is equal to the target items the quality measure is 1.0, i.e. 100%. For example, if the agent would generate 3 target items out of 5 and hallucinates 1, the overall score would be:

$$\begin{aligned} E_{total} &= 2 + 1 \\ Q &= 1 - \frac{3}{5} = 0.4 \end{aligned}$$

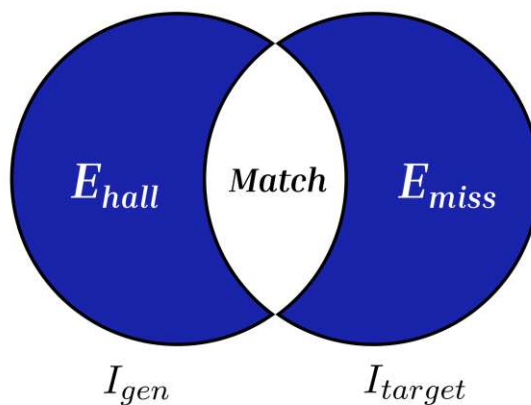


Figure 4.1: Visual representation of the intersection between the target item set I_{target} and the generated item set I_{gen} .

$$Quality\ Score = 40\%$$

This strict scoring mechanism ensures that high quality is only attributed to answers that are both comprehensive and precise. Thus, directly addressing the challenge of hallucination in LLM-based systems.

Evaluation

5.1 Executive Summary of Results

To provide an immediate understanding of the system’s effectiveness, this section presents a high-level illustration of the experimental data. The core objective of this research was to determine if the J-PRISM architecture could enhance code comprehension without a significant increase in computational costs. Figure 5.1 visualizes the relationship between operational cost and functional value for every individual benchmark run.

The X-axis represents the total token consumption (Input and Output) on a logarithmic scale. The Y-axis represents the achieved Quality Score. As illustrated, the two systems occupy distinct behavioral clusters. The positioning indicates a simultaneous optimization of two typically conflicted metrics. The system achieves a mean Quality Score of 97.69% while reducing the global token consumption by 52.40%.

In contrast, the Control baseline displays significant variance. Its distribution is more scattered. The mean Quality Score of the Control group is 89.37%. This instability confirms that raw file retrieval often introduces irrelevant noise into the context window. By retrieving precise semantic nodes, the agent reasons more effectively with less input data. The following sections define the experimental setup and provide a granular analysis of these findings.

5.2 Experimental Setup

To ensure the reproducibility of the results the evaluation was conducted in a controlled local environment. This approach isolates the performance of the J-PRISM architecture from external variables, such as network latency or API instability. This section elaborates

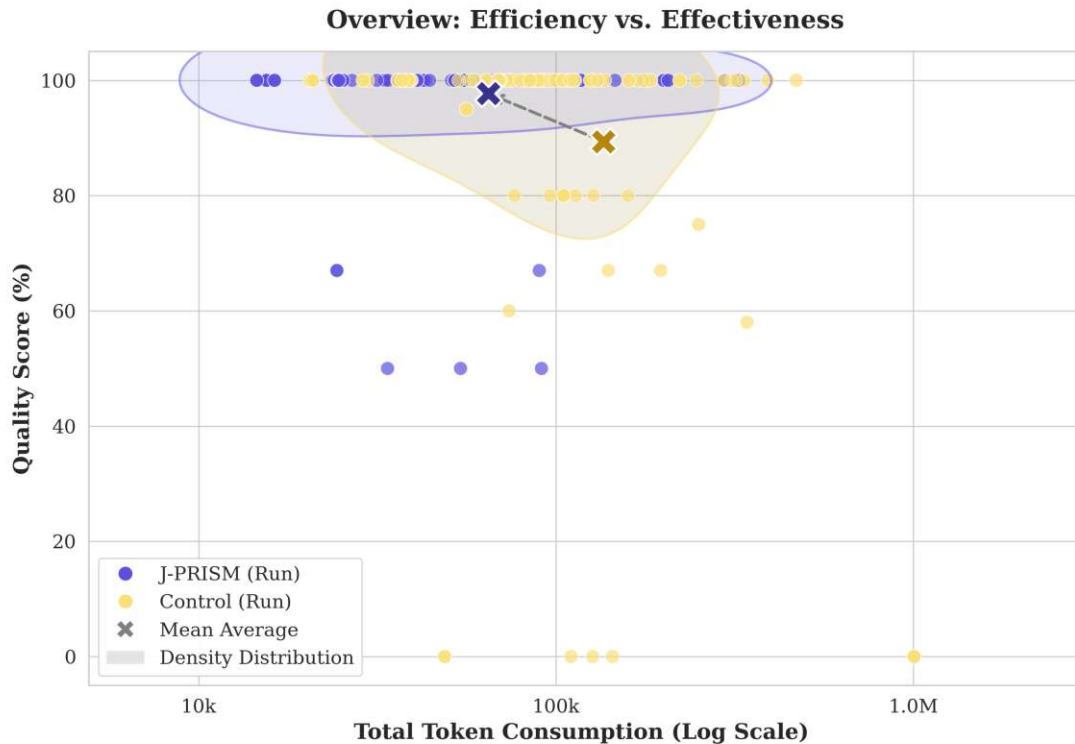


Figure 5.1: The Efficiency vs. Effectiveness landscape. The plot visualizes the specific performance of J-PRISM (purple) against the Control baseline (yellow). The density contours highlight the concentration of the J-PRISM runs in the high-quality, low-cost quadrant.

the hardware specifications, the inference configuration, and the system definitions used for the comparative analysis.

5.2.1 Hardware and Execution Environment

All experiments were executed on a workstation equipped with an ARM-based *Apple M4 Pro* chip [4] and 48GB of Unified Memory. This unified memory architecture is critical. It allows for the loading of substantial LLM weights directly into high-speed memory without offloading to slower storage tiers.

The model hosting was managed by *LM Studio* [33]. This application provided a local, OpenAI-compatible server interface. By running the inference engine locally, we eliminated network overhead as a confounding factor in the *Execution Time* metric. This setup ensured that the inference parameters remained strictly constant across all trials.

5.2.2 Model Configuration

To optimize performance for distinct computational tasks, the experimental framework utilizes a multi-model architecture. The configuration is divided into the primary inference engine, which derives the agent's reasoning, and the auxiliary models used for data ingestion and vector retrieval.

Inference Engine The core reasoning model for both the baseline and the J-PRISM system is provided by the *Mistral* architecture. This model is responsible for interpreting the prompts, executing tools, and generate the final answer during the testing phase. The model was loaded with specific quantization settings within the constraint of the local experimental environment. The following configurations for the LLM were used:

- **Model:** Devstral-Small-2507 by *Mistral AI* [55]
- **Quantization:** Q8_0
- **Context Window:** 50273 Tokens
- **Temperature:** 0,8
- **Format:** GGUF

Documentation Model In addition to the main agent, two specialized models were employed to construct and query the KG. First, a Documentation Model was utilized exclusively during the system setup phase to enhance the KG. It was used to generate missing method and class documentation for the graph nodes, enhancing the semantic searchability. The following configurations were used for this model:

- **Model:** Qwen2.5-Coder-7B-Instruct by *Qwen* [42]
- **Quantization:** 8bit
- **Context Window:** 32768 Tokens
- **Temperature:** 0,8
- **Format:** MLX

Embedding Model Second, an Embedding Model was used to vectorize the nodes in the KG. This model is active during both the system setup (indexing) and the testing phase (similarity search retrieval).

- **Model:** Qwen3-Embedding-8B by *Qwen* [43]
- **Quantization:** Q4_K_M

- **Context Window:** 4096 Tokens
- **Temperature:** 0,8
- **Format:** GGUF

The selection of this heterogeneous set of models was primarily driven by local hardware constraints and latency optimization. Smaller, specialized models were utilized for auxiliary tasks to preserve computational resources for the primary reasoning engine. Crucially, the architecture remains robust and model-agnostic. The specific combination is not a strict requirement. Consequently, similar performance outcomes are expected with alternative models that meet the necessary capability requirements.

5.2.3 System Configuration

Two distinct configurations of the OpenHands framework were evaluated to determine the performance of the proposed solution.

Baseline Configuration The control group consists of the standard OpenHands distribution running without external augmentation. In this configuration, the agent is restructured to the default toolset, primarily relying on the standard file system search. This setup simulates the current state-of-the-art for general coding agents, which treat the codebase as a collection of unstructured text files rather than a semantic graph.

J-PRISM Configuration The experimental group utilizes the modified OpenHands agent extended with the J-PRISM architecture described in Chapter 3. The primary distinction lies in the MCP integration. While the baseline relies on raw file access, the J-PRISM agent is equipped with the custom KG retrieval tools. This allows the agent to execute semantic queries on the KG.

Both systems were initialized with the exact same LLM models. Also, the exact same prompts derived from the benchmark suite defined in Chapter 4 were used. To prevent context contamination, the agent session and the shell environment were reset completely between the configuration changes.

5.3 Response Quality Analysis

This section evaluates the functional correctness and completeness of the generated outputs. This assessment directly addresses the primary research question regarding the comprehension of enterprise Java repositories. The analysis utilizes the Quality Score metric defined in Section 4.3. The evaluation follows a top-down approach. First, global performance is analyzed to determine overall system reliability. Second, the results are split by repository archetype. Finally, specific comprehension areas are examined to isolate strengths and weaknesses in the retrieval logic.

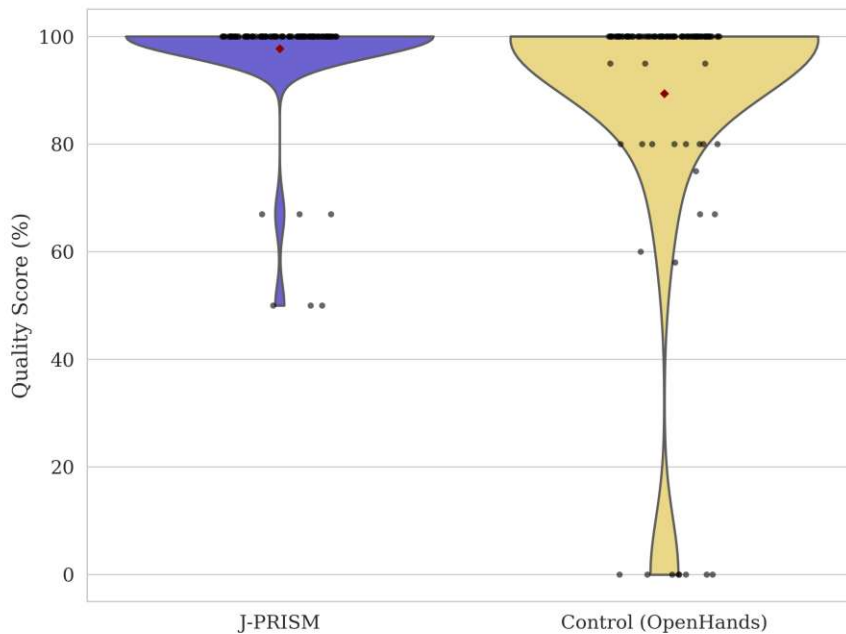


Figure 5.2: Global distribution of Quality Scores comparing J-PRISM against the Control (OpenHands) baseline.

5.3.1 Global Performance

The global performance analysis compiles the Quality Scores across all repository archetypes and all comprehension domains.

The Figure 5.2 illustrates the comparative score distributions for both systems. The J-PRISM architecture has a probability density heavily concentrated at the 100% mark with a low variance. This indicates a degree of stability. Conversely, the Control group displays a higher variance. While the baseline frequently achieves perfect precision as well, it has a "long tail" extending into the lower quartile. The mean of the Quality Scores for both systems is:

System	Quality Score
J-PRISM:	97.69%
Control (OpenHands):	89.37%

This tendency suggests that standard file-based retrieval lacks robustness against implicit complexities.

A pairwise comparison quantifies the performance delta for each benchmark task (Figure 5.3). The predominant outcome, accounting for 74.1% of the total variance, was

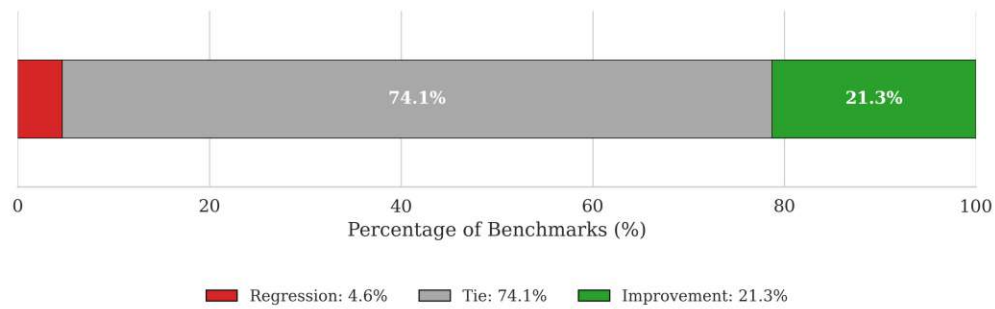


Figure 5.3: Head-to-Head Comparison: J-PRISM compared to the Control (OpenHands)

performance equivalence. This validates that the graph abstraction preserves baseline capabilities for explicit retrieval tasks. However, this high tie rate also suggests limited complexity in certain benchmark segments. A significant subset (21,3%) display a performance increase. Nevertheless, the J-PRISM approach had a decreased performance on a minor fraction (4,6%) of tasks. The specific mechanisms responsible for these rare failures are examined in the qualitative analysis in Section 5.6.

The difference between the improvement and regression rates confirms a net positive impact. The J-PRISM architecture effectively raises the performance bar.

5.3.2 Performance by Repository Archetype

To understand the specific impact of the KG integration, we must isolate performance based on system complexity. This section analyzes the Quality Score distributions across the three defined repository archetypes.

Archetype I: Standard Layered CRUD

The results for Archetype I indicate a functional equivalence between the baseline and the J-PRISM architecture. As illustrated in Figure 5.4a, the density plots for both systems are nearly identical. Both distributions are tightly clustered at the upper bound. This suggests that standard file-based retrieval is sufficient for navigating explicit, linear control flows common in simple Spring Boot applications.

This observation is quantified in Figure 5.5a. The vast majority of benchmarks (94,4%) result in a tie. Only a marginal improvement of 5,6% was recorded, with 0,0% regression. Consequently, for repositories with low complexity and explicit dependencies, the graph overhead provides minimal additional value.

Archetype II: Domain-Centric Application

The performance difference becomes visible in Archetype II. This category introduces complex business rules and implicit state dependencies. Figure 5.4b shows that the

	Archetype I	Archetype II	Archetype III
J-PRISM	100%	94.00%	99.08%
Control (OpenHands)	98.89%	78.33%	90.88%

Table 5.1: Mean comparison of J-PRISM and Control (OpenHands) across Archetypes

Control group (yellow) begins to display significant variance. Its distribution stretches downward, indicating a loss of reliability. In contrast, the J-PRISM system (purple) maintains a high density at the top of the scale.

However, this archetype also introduces volatility. Figure 5.5b reveals a more mixed outcome. While 27.8% of cases showed improvement, this category also recorded the highest regression rate at 13.9%. This suggests that while the graph helps navigate domain complexity, there are edge cases where the abstraction may obscure simple keyword matches found by the baseline.

Archetype III: Infrastructure-Intensive Architecture

The most significant impact is observed in Archetype III. This category represents high-complexity enterprise environments. Figure 5.4c demonstrates a clear collapse in the baseline’s stability. The Control group displays a long tail extending into the lower quartiles, reflecting frequent retrieval failures. The J-PRISM architecture, however, remains robust with a stable high-scoring distribution.

The breakdown in Figure 5.5c confirms this advantage. The J-PRISM system achieved an improvement in 30.6% of the benchmarks. Notably, there were 0.0% regressions in this category. This confirms the core hypothesis: as architectural complexity increases, the semantic graph becomes essential for accurate code comprehension.

Table 5.1 summarizes the mean Quality Scores across the three distinct complexity levels. Consistent with the density analysis, J-PRISM maintains superior performance across all categories, sustaining a mean score above 94% regardless of the repository complexity.

5.3.3 Performance by Comprehension Area

To isolate the specific retrieval mechanics that contribute to the global performance, we investigate four distinct comprehension domains described in Chapter 4. This granular analysis identifies where the KG integration provides critical context and where it may introduce additional overhead.

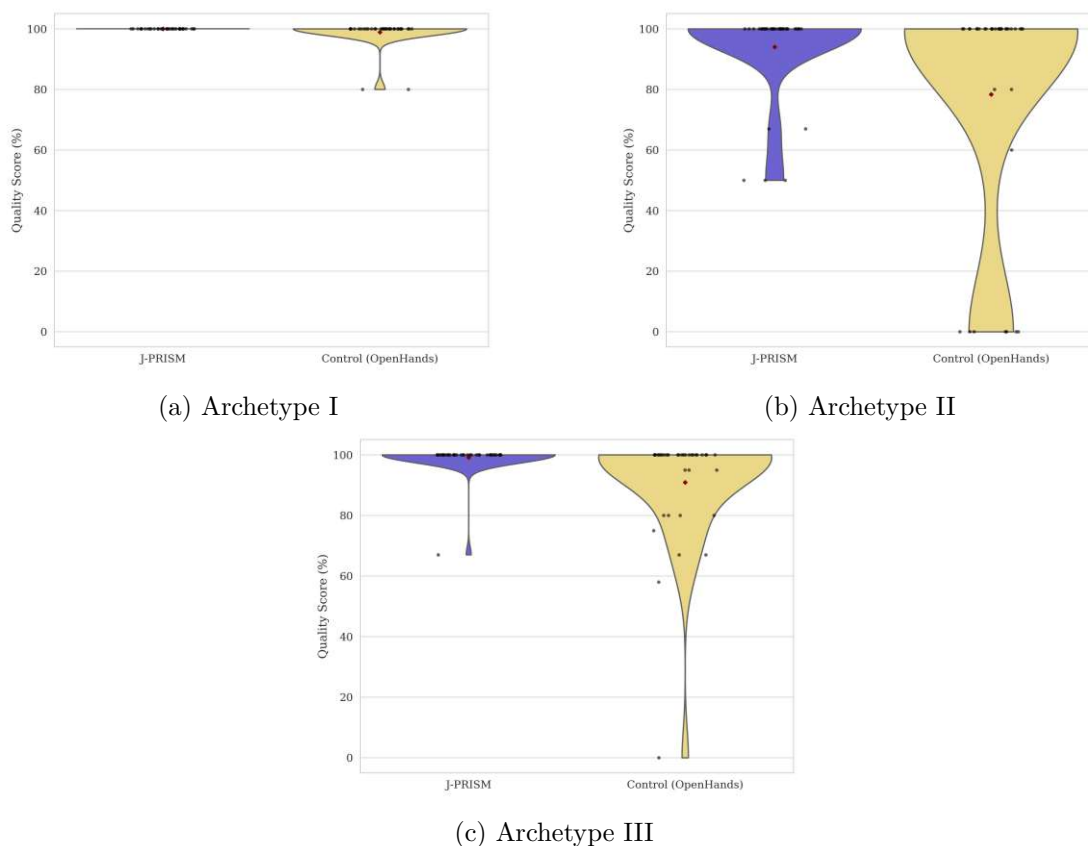
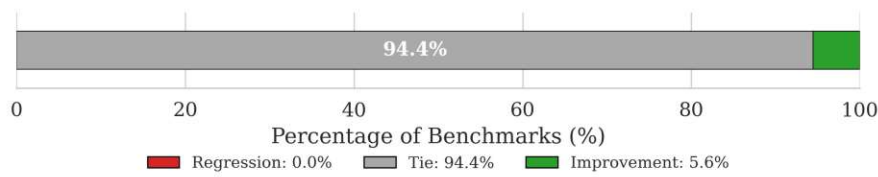


Figure 5.4: Comparative density estimation of Quality Scores for J-PRISM (purple) and the Control (yellow) across Archetypes I (a), II (b), and III (c). Red diamonds denote mean values. While Archetype I shows similar performance profiles, Archetypes II and III demonstrate that J-PRISM maintains stability at the upper bound, whereas the Control group display significant variance with distributions extending into the lower quartiles.

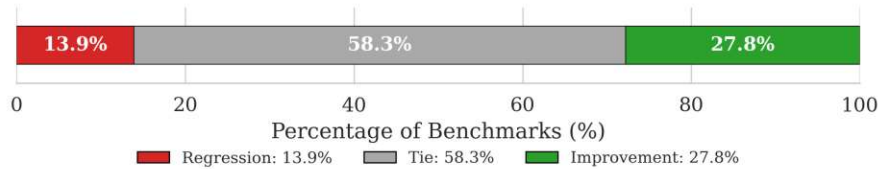
Annotation Comprehension

The result for Annotation Comprehension indicate that standard text retrieval remains highly effective for explicit metadata. As shown in Figure 5.7a, the majority of benchmarks (85.2%) resulted in a tie. This is expected, as annotations are distinct keywords markers easily indexed by standard search tools.

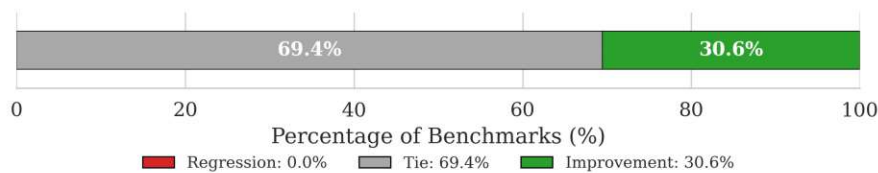
However, the density estimation in Figure 5.6a reveals a subtle distinction. While the Control group shows a long tail of variance, the J-PRISM system maintains a tighter distribution. Despite this stability, the domain recorded a 7.4% regression rate. This suggests that in rare instances the graph traversal may over-complicate the retrieval of simple, explicit text markers.



(a) Archetype I



(b) Archetype II



(c) Archetype III

Figure 5.5: Benchmark performance of J-PRISM compared to the Control (OpenHands) breakdown for Archetypes I, II, and III, showing the percentage of regression (red), ties (grey), and improvements (green).

Functionality and Execution Flow

The most significant performance divergence can be observed in the Functionality and Execution Flow domain. This category requires the agent to trace logic across class boundaries. Figure 5.7b highlights a substantial 44.4% improvement rate for the J-PRISM architecture, with 0.0% regression.

The impact on reliability is obvious in Figure 5.6b. The Control group displays a higher variance. In contrast, J-PRISM maintains a perfect stability at the upper bound. This indicates that file-based retrieval is insufficient for resolving implicit execution paths in Spring-based applications.

Method Identification and Specificity

In contrast, the Method Identification domain exposes a limitation in the current architecture. This area focuses on resolving specific method signatures and overloaded functions. Figure 5.7c records the highest regression rate at 11.1%.

Comprehension Area	J-PRISM	Control (OpenHands)
Annotation Comprehension	96.33%	93.85%
Functionality and Execution Flow	100%	78.52%
Method Identification and Specificity	94.44%	99.44%
Structural and Architectural Insight	100%	85.67%

Table 5.2: Mean Quality Score comparison across the four comprehension domains. While J-PRISM demonstrates superior performance in structural and functional tasks, the Control baseline retains a slight advantage in specific method identification.

The density plot in Figure 5.6c reinforces this finding. Unlike other domains, the J-PRISM distribution shows a noticeable density drop. The Control group effectively retains a higher density at the top. This suggests that for direct signature matching, raw text search occasionally outperforms the semantic node resolution. Section 5.6 will investigate this behavior in more detail. Note that even though the regression and the improvement rate is the same, both J-PRISM and the Control group struggled in distinct cases.

Structural and Architectural Insight

The final domain evaluates the agent’s understanding of high-level system components. Figure 5.7d shows a clear advantage for the graph-based approach, with a 22.2% improvement and 0.0% regression.

Similar to the functionality domain, the Control group in Figure 5.6d displays significant instability. The J-PRISM system successfully generated all required items throughout the benchmarks without hallucination.

Table 5.2 summarizes the mean Quality Scores across the four comprehension domains. J-PRISM demonstrates a distinct advantage in certain tasks. However, the data also confirms the exception in Method Identification, where the Control baseline retains a higher precision.

5.4 Efficiency and Cost Analysis

While the previous section demonstrates that the J-PRISM architecture enhances the quality of the code comprehension, these improvements must be weighed against the computational overhead. In the context of LLMs, efficiency is primarily defined by the volume of processed data and the associated execution time. This section evaluates the trade-off between the increased retrieval precision of the KG and the resulting impact on resource consumption.

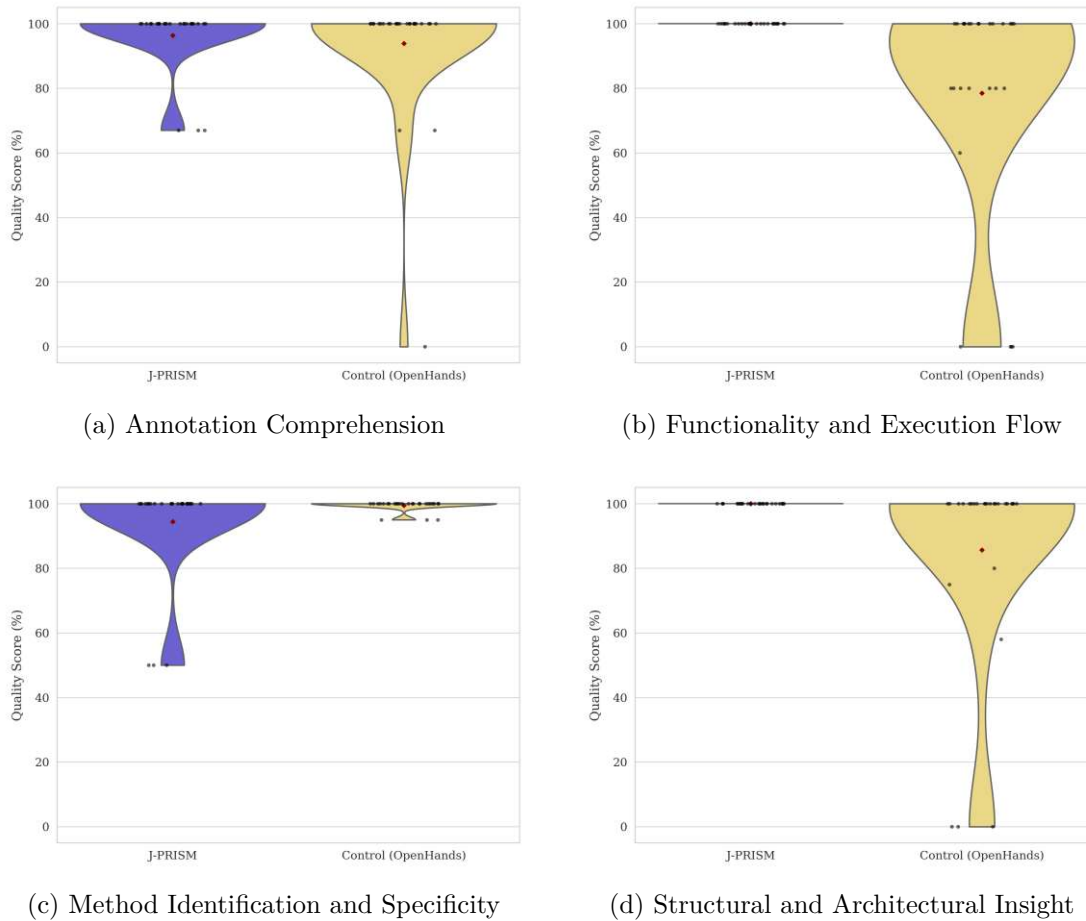
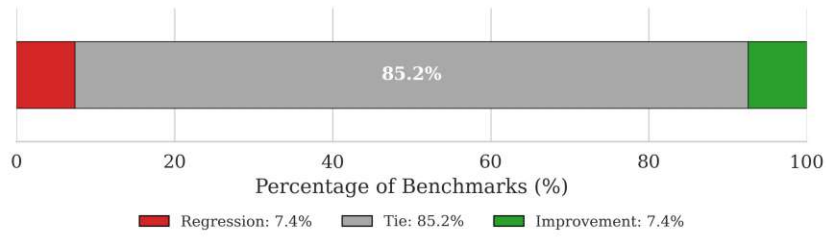
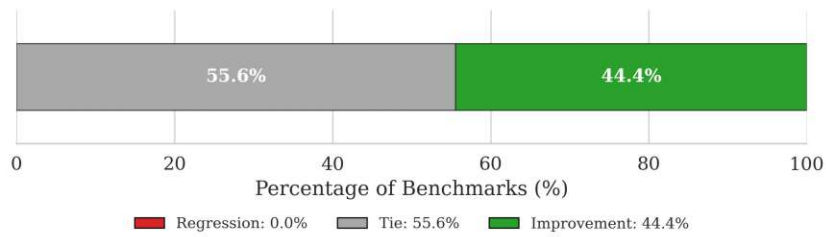


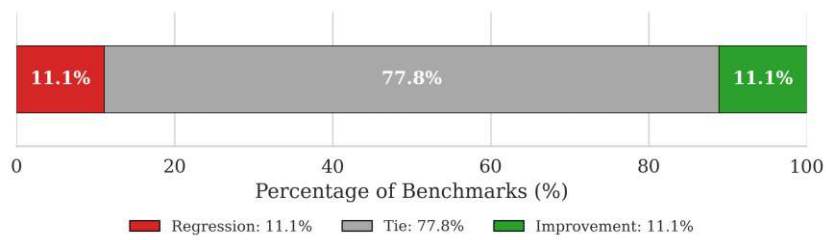
Figure 5.6: Comparative analysis of Quality Score distributions across the four distinct comprehension domains. The violin plots illustrate that J-PRISM (purple) maintains higher density at the upper bound with reduced variance compared to the Control baseline (yellow) across (a), (b) and (d). J-PRISM has a lower performance in the Method Identification and Specificity area.



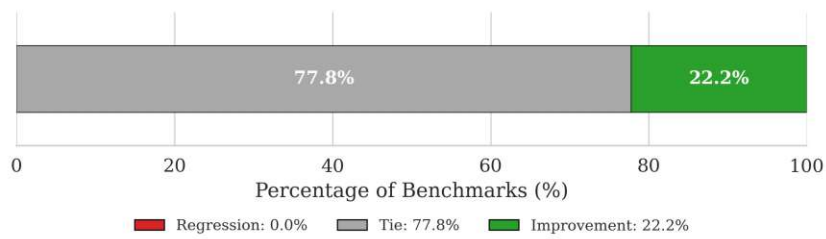
(a) Annotation Comprehension



(b) Functionality and Execution Flow



(c) Method Identification and Specificity



(d) Structural and Architectural Insight

Figure 5.7: Pairwise performance analysis categorized by comprehension domain. The stacked bar charts quantify the distribution of improvement (green), equivalent performance (ties), and regression (red) of the J-PRISM architecture relative to the Control baseline across all repository archetypes.

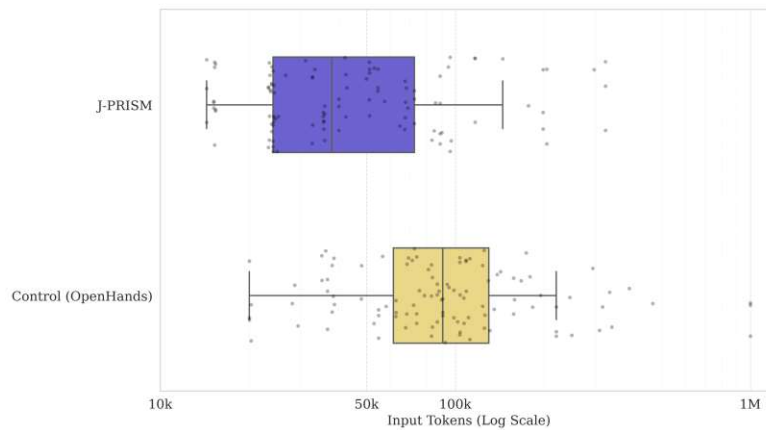


Figure 5.8: The global token consumption of the J-PRISM approach and the Control (OpenHands).

5.4.1 Token Consumption

The operational cost of the inference engine is primarily determined by token consumption. This metric combines the prompt context (Input Tokens) and the generated reasoning (Output Tokens). We analyze these two dimensions separately to understand the impact of the KG on the retrieval strategy.

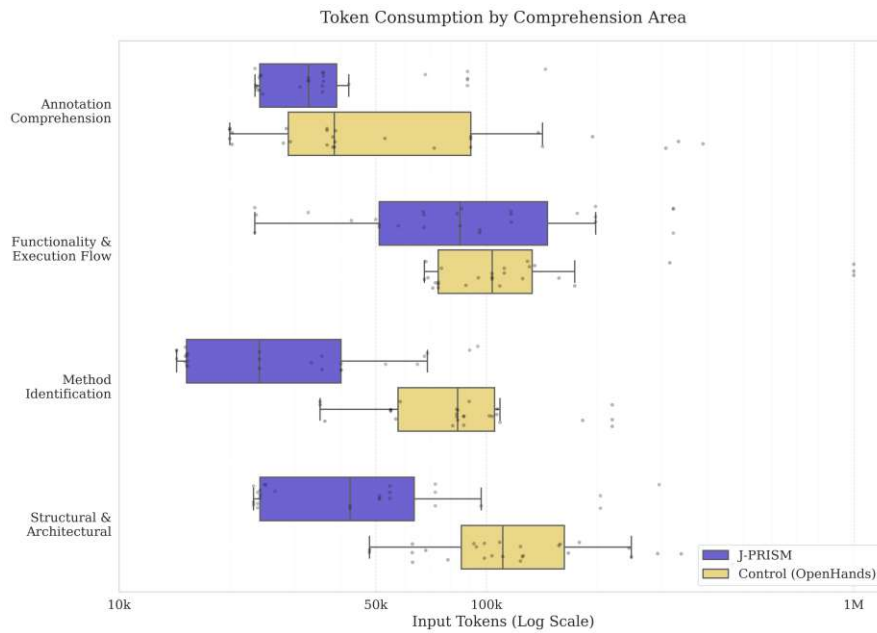
Input Tokens

Although the J-PRISM approach achieved a higher Quality Score, it also needed less input tokens. Figure 5.8 illustrates a significant reduction in token usage by the J-PRISM approach. The distribution for our system is concentrated in the lower log-scale range. In contrast, the Control group (OpenHands) consistently consumes significantly more context. This efficiency derives from the structural difference in retrieval. The baseline agent retrieves entire files to establish context. In verbose Java enterprise codebases, this results in loading thousands of irrelevant lines of code. The J-PRISM agent, however, retrieves specific semantic nodes. This retrieval approach eliminates the overhead of surrounding boilerplate code.

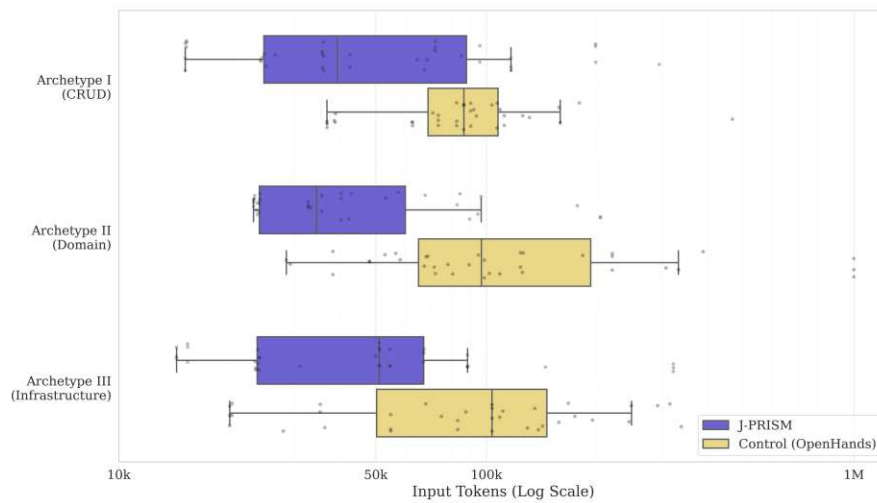
For each subgroup of Comprehension Area and Repository Archetype the J-PRISM decreased the input tokens. This can be seen in Figure 5.9. Table 5.3 compares the average input tokens between both configuration setups by each subgroup.

Output Tokens

The output token consumption serves as a metric for the efficiency of the reasoning process. A higher token count often indicates that the model requires a verbose "chain-of-thought" or iterative self correction to arrive at an answer. The global output distribution (Figure 5.10) demonstrates a substantial efficiency gain. The J-PRISM architecture



(a) Input Token Consumption by Comprehension Area



(b) Input Token Consumption by Repository Archetype

Figure 5.9: Token consumption comparison of the J-PRISM approach and the Control (OpenHands) filtered by Comprehension Area and Repository Archetype.

Repository Archetype	J-PRISM	Control	Improvement
Archetype I (CRUD)	68,837	98,467	−30.1%
Archetype II (Domain)	54,544	190,899	−71.4%
Archetype III (Infrastructure)	68,732	114,200	−39.8%

Comprehension Area	J-PRISM	Control	Improvement
Annotation Comprehension	44,240	88,572	−50.1%
Functionality & Execution	114,549	208,941	−45.2%
Method Identification	33,988	96,466	−64.8%
Structural & Architectural	63,374	144,109	−56.0%

Comparison Scope	J-PRISM	Control	Improvement
Global Mean	64,038	134,522	−52.4%

Table 5.3: Mean Input Token Consumption Analysis. This table compares the average input token cost between J-PRISM and the Control group (OpenHands) across three dimensions: Repository Archetype, Comprehension Area, and Global Average.

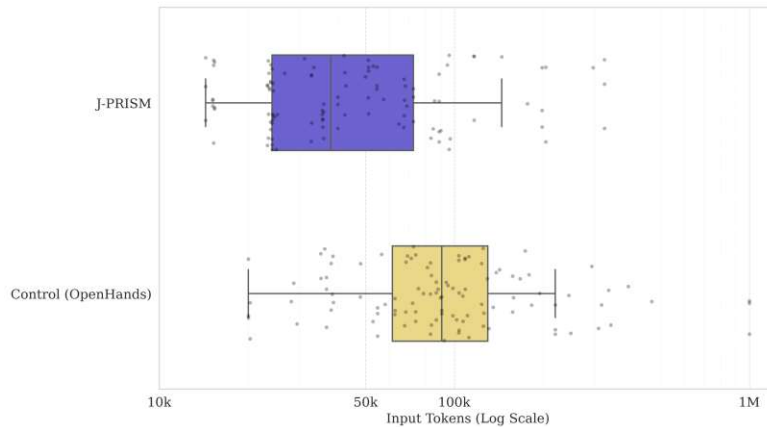
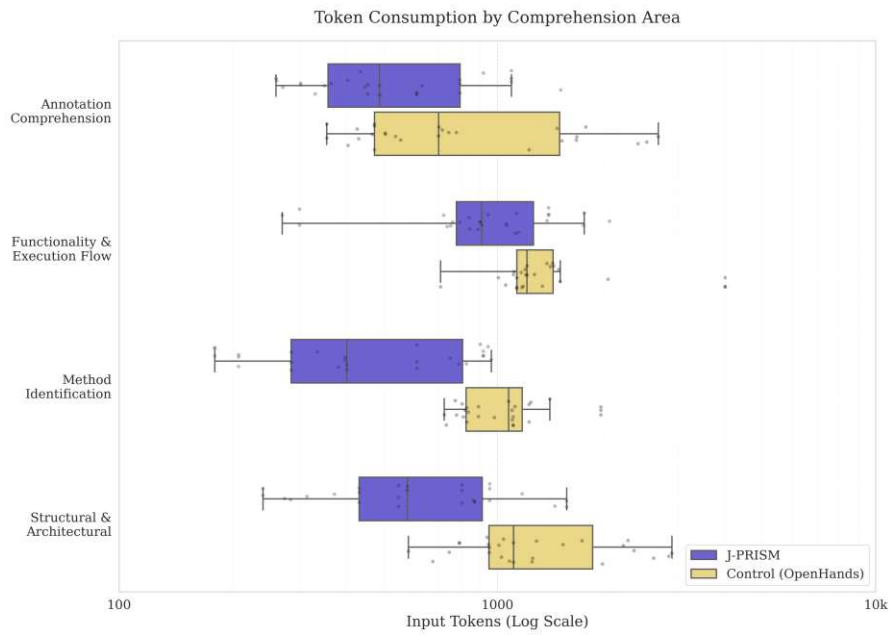
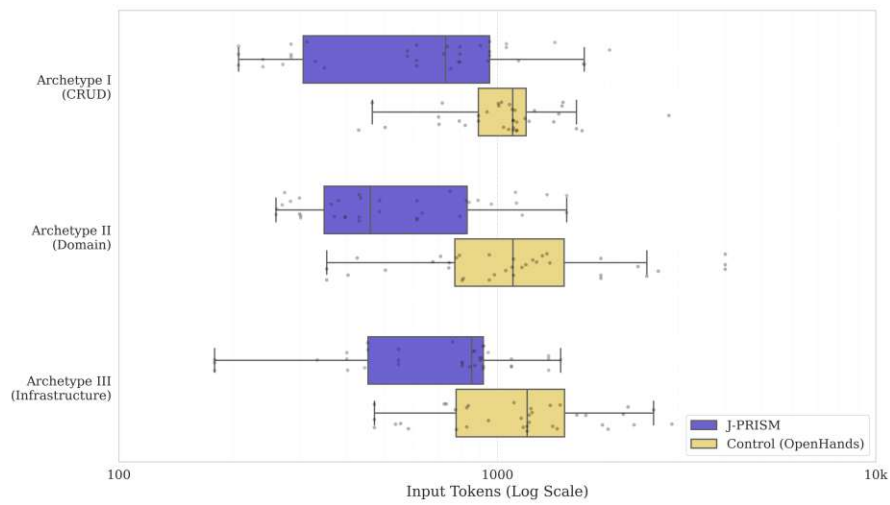


Figure 5.10: The global output token usage of the J-PRISM approach and the Control (OpenHands).

achieves a mean reduction of 42.9% compared to the Control baseline (Table 5.4). This indicates that graph-based context enables direct answer formulation, whereas the baseline agent often relies on more extended reasoning. This trend holds across all repository types (Figure 5.11b) and areas of code comprehension (Figure 5.11a).



(a) Output Token Consumption by Comprehension Area



(b) Output Token Consumption by Repository Archetype

Figure 5.11: Output Token consumption comparison of the J-PRISM approach and the Control (OpenHands) filtered by Comprehension Area and Repository Archetype.

Repository Archetype	J-PRISM	Control	Improvement
Archetype I (CRUD)	732	1,102	-33.6%
Archetype II (Domain)	625	1,372	-54.4%
Archetype III (Infrastructure)	774	1,258	-38.5%

Comprehension Area	J-PRISM	Control	Improvement
Annotation Comprehension	597	970	-38.4%
Functionality & Execution	1,006	1,540	-34.7%
Method Identification	522	1,076	-51.5%
Structural & Architectural	716	1,390	-48.5%

Comparison Scope	J-PRISM	Control	Improvement
Global Mean	710	1,244	-42.9%

Table 5.4: Mean Output Token Consumption Analysis. This table compares the average cost of the generated response between J-PRISM and the Control group (OpenHands).

5.4.2 Execution Latency

To validate the total execution latency we are comparing both configurations generation time. We examined the relationship between the total computation load (Input and Output tokens) and the resulting execution latency in seconds.

The scatter plot in Figure 5.12 reveals a strong positive linear correlation for both the J-PRISM and the Control (OpenHands) configuration. This indicating that execution time is primarily a function of the volume of text processed rather than the architectural complexity of the system. Crucially, the regression lines for both systems display a high degree of overlap. This convergence suggests that the introduction of the KG retrieval mechanism does not impose a statistically significant penalty on the system’s processing throughput.

The regression line for J-PRISM appears slightly steeper at the upper end. This divergence is driven by a sparse set of outliers rather than a systemic inefficiency. In the vast majority of operational contexts, the time-per-token cost remains invariant between the two systems. Consequently, we conclude that the J-PRISM architecture achieves its superior context retrieval without degrading the fundamental computational efficiency.

5.4.3 MCP Tool Usage

To further evaluate the operational efficiency of the agentic loop, we examine the frequency of tool invocations required to resolve a task. This metric serves as a direct

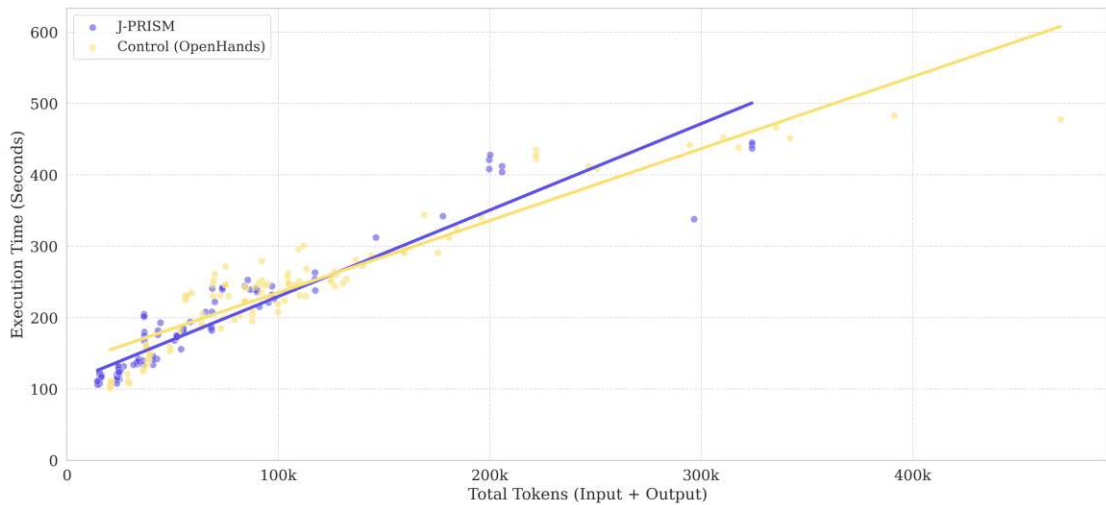


Figure 5.12: Relationship between total computation load (Input and Output Tokens) and the execution time by J-PRISM and Control (OpenHands).

indicator of the retrieval precision. A higher number of tool calls typically signifies an iterative, trial-and-error search strategy. Whereas a lower count suggests that the agent can identify the necessary context with fewer interactions.

Figure 5.13 illustrates the distribution of tool calls per task for both configurations. The results reveal a clear variance in agent behavior. The J-PRISM architecture demonstrates a significant higher efficiency, with the majority of tasks being resolved in under 12 tool calls and a median centered at approximately 5 calls. In contrast, the Control group displays a significantly higher variance and a higher median of approximately 10 tool calls. When considered alongside the findings in Section 5.4.2, this reduction does not lead to a reduction in execution latency. So, the J-PRISM system uses fewer but more execution expensive tools.

5.5 Ablation Study

5.5.1 Study Design

To validate the contribution of individual components within the J-PRISM architecture, we conducted an ablation study. We defined two stripped-down variants of the J-PRISM system to isolate the impact of Semantic Enrichment and Symbolic Retrieval.

Variant A (Vector-Only) The agent is restricted to the similarity search tool. This isolates the impact of the Semantic Enrichment component.

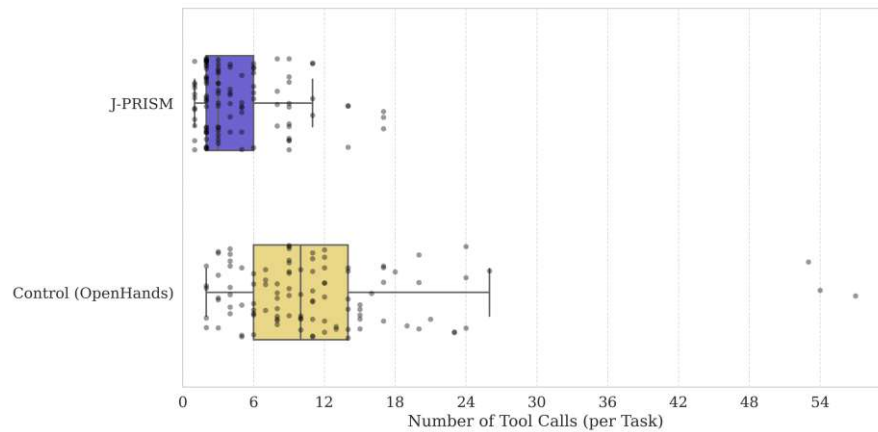


Figure 5.13: Comparing the amount of tool calls per task by J-PRISM and Control (OpenHands).

Variant B (Cypher-Only) The agent is restricted to the Cypher query tool. This isolates the impact of the Symbolic Retrieval mechanism. Also, in this variant the documentation and KG enhancement step was not performed.

We evaluate these variants against the repository Archetype III since this category demonstrated the highest complexity and the most significant performance of the J-PRISM system. The result of these variants will then be compared to the J-PRISM approach.

5.5.2 Qualitative Analysis

The quantitative results, summarized in Table 5.5, reveal a clear hierarchy in performance. The full J-PRISM system achieved a Quality Score of 99.05%, significantly outperforming both isolated variants.

Impact of Semantic Removal (Cypher-Only): Restricting the system to symbolic retrieval resulting in a Quality Score of 75.00%. While the agent successfully navigated explicit structures, it struggled with vague or natural language-based queries.

Impact of Symbolic Removal (Vector-Only): Relying solely on semantic search caused a drastic regression, yielding a Quality Score of only 47.12%. This indicates that vector similarity alone is insufficient for resolving the complex, deep dependency chains characteristic of Archetype III.

Metric	J-PRISM	Variant A	Variant B
Input Tokens	68,731	47,764	57,250
Output Tokens	774	381	561
Execution Time (s)	180	142	154
MCP Tool Calls	4.64	3.50	5.64
Quality Score	99.05%	47.12%	75.00%

Table 5.5: Comparative Performance Metrics for Ablation Variants

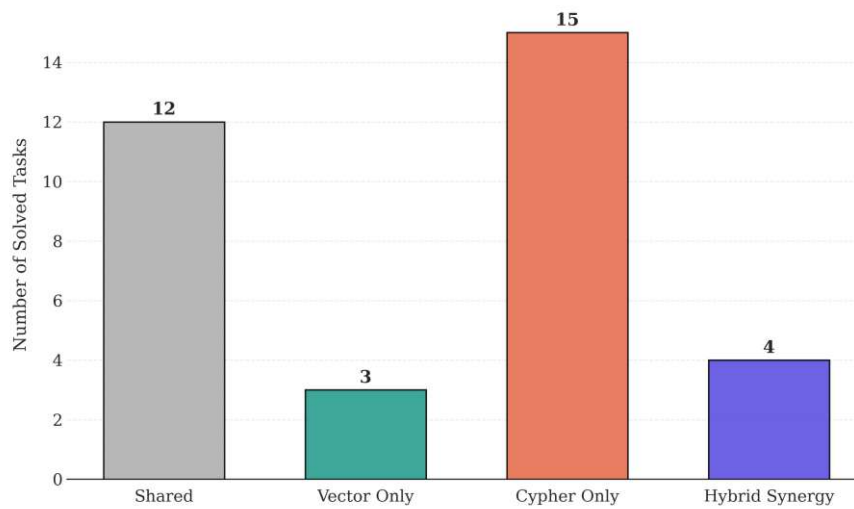


Figure 5.14: Attribution of Success Breakdown.

To further analyze the underlying drivers of this performance gap, we categorized the successful retrieval tasks based on which system variant was capable of solving them. Figure 5.14 illustrates this attribution of success.

The data demonstrates that the superiority of the J-PRISM architecture is not merely additive but synergistic. A significant portion of tasks (Hybrid Synergy) failed in both isolation variants, but were successfully resolved by the full J-PRISM system. This confirms that the hybrid architecture enables emergent reasoning capabilities. The system leverages the Semantic Enrichment to identify vague entry points and then utilizes Symbolic Retrieval to traverse the dependency graph. Furthermore, the distinct "Vector Only" and the "Cypher Only" contributions highlight the complementarity of the components.

5.5.3 Stability and Efficiency

Beyond absolute correctness, the behavioral profiles of the variants differ significantly. Figure 5.15 illustrates the stability of the Quality Scores. The Vector-Only variant displays a low-mean distribution, consistently failing to retrieve precise structural context.

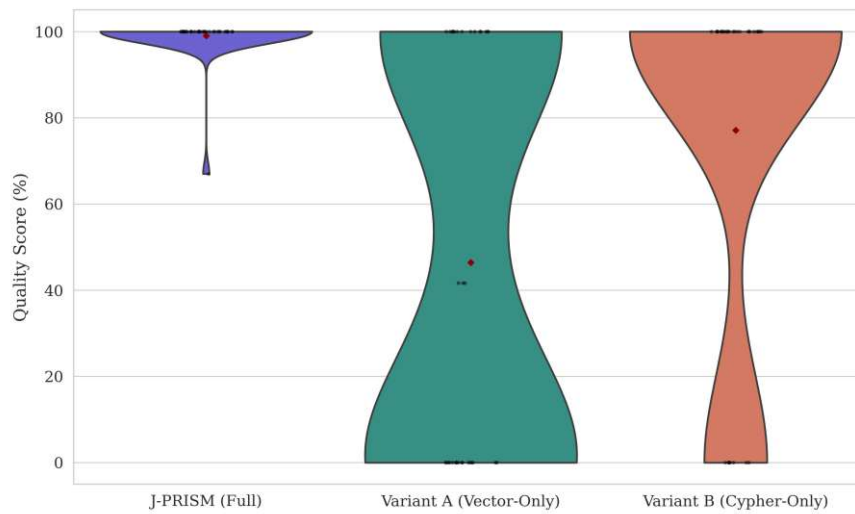


Figure 5.15: Quality Score Distribution of J-PRISM, Variant A, and Variant B

The Cypher-Only variant shows a bimodal distribution. It either retrieves the exact subgraph (100%) or fails completely when the correct schema cannot be inferred.

In terms of efficiency, Variant B (Cypher-Only) recorded the highest average number of MCP tool calls (Table 5.5). This increase confirms the "search struggle" hypothesis: while the agent is schema-aware, it lacks the semantic context to immediately formulate precise retrieval strategy, resulting in a higher frequency of exploratory tool calls. In contrast, the full J-PRISM system reduced the tool usage to 4.70, indicating that the hybrid approach allows for more targeted and efficient retrieval.

Finally, Figure 5.16 analyzes the computational cost in terms of total token consumption. Variant A (Vector-Only) demonstrates greater stability in resource usage, characterized by a tighter distribution of token consumption and a lower overall mean. In contrast, Variant B (Cypher-Only) displays significantly higher variance, driven by the iterative query attempts. The full J-PRISM architecture utilizes a marginally higher token consumption, but effectively converts this additional context into a higher success rate and superior stability.

5.6 Qualitative Analysis

The previous quantitative analysis established the statistical superiority of the J-PRISM architecture in terms of accuracy and efficiency. However, to understand the underlying mechanisms of these performance differences, a focused examination of individual execution traces is required. This section provides a comparative analysis of specific

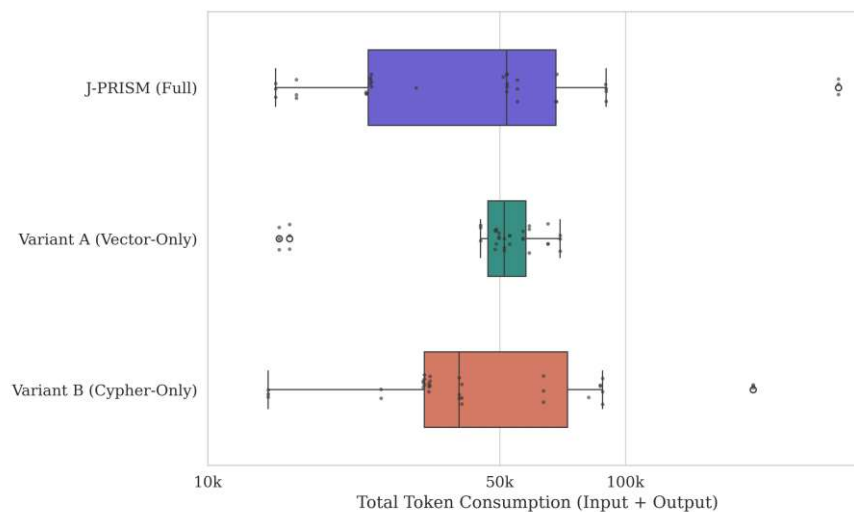


Figure 5.16: Comparing the total token consumption per task of J-PRISM, Variant A, and Variant B

benchmark scenarios. By examining the agent’s internal process logs and tool-calling sequences, we can isolate the specific architectural features that benefit and hinder code comprehension.

5.6.1 Success Case

This case study examines a task from Archetype II involving the tracing of a function call. The objective from the Functionality and Execution Flow Comprehension Area was:

Which methods call the following function:

```
"Optional<ProfileData> findByUsername(String username, User currentUser)"
```

This task specifically evaluates the system’s ability to distinguish between overloaded methods based on their method signature rather than just the identifier.

J-PRISM Execution Trace

The J-PRISM agent resolved this request with two distinct operational steps. Initially, the agent executed a similarity search to locate the target method definition within the vector space:

I'll use the similarity search tool to find methods that call the function "Optional<ProfileData> findByUsername(String username, User currentUser)".

```
Function call: run_sim_search(
    "query": "findByUsername(String username, User currentUser)"
)
```

Upon realizing that simple similarity search did not yield precise results, the agent autonomously refined its strategy. It constructed a target Cypher query against the KG:

The similarity search results show several methods with similar names, but none of them exactly match the signature "Optional<ProfileData> findByUsername(String username, User currentUser)". Let me refine the search by looking for methods that specifically call this function.

Let me run a more targeted cypher query to find methods that are calling this specific function:

```
Function call: run_cypher_query({"query":
    "
        MATCH (m:CodeMethod)-[:IS_CALLING]->(t:CodeMethod)
        WHERE t.methodName = "findByUsername"
        AND t.inputParameters =
            ["String username\", \"User currentUser\"]
        AND t.returnType = "Optional<ProfileData>"
        RETURN m.methodName, m.packageName
    "
})
```

This query explicitly filtered for the correct method node. This precise semantic filtering allowed the agent to retrieve the exact call graph without hallucination. The system then correctly identified all five dependent methods:

Great! The cypher query found several methods that call the function "Optional<ProfileData> findByUsername(String username, User currentUser)". Here are the methods:

1. ``queryProfile``
2. ``follow``
3. ``unfollow``
4. ``getProfile``
5. ``buildProfile``

These methods are calling the specific function you're interested in. Would you like more details about any of these methods or their containing classes?

Here, the J-PRISM system has used the KG effectively to find all requested answers. Note that in this example the agent did not use the output of the similarity search because the agent could not identify the correct signature. However, when reviewing the output of the similarity search tool, the correct method with the signature was provided. Therefore, it seems that the LLM was overwhelmed with the JSON-format representation of the top 5 responses.

Control Group Execution Trace

In contrast, the Control Group (OpenHands) could not identify all methods. First, the agent searched for the method in the codebase:

I'll help you find which methods call the function ``Optional<ProfileData> findByUsername(String username, User currentUser)``.

Let me search through the codebase to locate this method and then identify its callers.

```
Function call: execute_bash({"command":
  "find workspace -type f -name "*.java"
    | xargs grep -l "findByUsername.*String username.*User
      currentUser" 2>/dev/null",
  "timeout": "30"
})
```

This gave the agent the file path where the `findByUsername` function is located. Then, the agent retrieved the correct method signature to make further tool calls to retrieve the calling methods:

Great! I found the file that contains the `findByUsername` method. Let me examine this file to understand the method signature and then search for its callers.

```
Function call: str_replace_editor({
    "command": "view",
    "path": "/workspace/main/java/io/spring/
              application/ProfileQueryService.java"
})
```

Now that I have the method signature, let me search for all files that call this method:

```
Function call: execute_bash({"command":
    "find /workspace -type f -name "*.java"
    | xargs grep -l "findByUsername.*username.*currentUser"
    2>/dev/null",
    "timeout": "30"
})
```

Let me try a more general search to find all calls to this method:

```
Function call: execute_bash({"command":
    "find /workspace -type f -name "*.java"
    | xargs grep -n "\.findByUsername("
    2>/dev/null",
    "timeout": "30"
})
```

Now I can see the actual calls to `profileQueryService.findByUsername(username, currentUser)`.

Let me filter these results to show only the relevant ones:

```
Function call: execute_bash({"command":
    "find /workspace -type f -name "*.java"
    | xargs grep -n "profileQueryService\.findByUsername("
    2>/dev/null",
    "timeout": "30"
})
```

Perfect! Now let me examine the actual method calls in the production code (excluding tests) to understand which methods are calling ``profileQueryService.findByUsername(username, currentUser)``:

```
Function call: str_replace_editor({"command": "view",
                                   "path": "/workspace/main/java/io/spring/graphql/
                                   RelationMutation.java",
                                   "view_range": [50, 65]
                                })
```

```
Function call: str_replace_editor({"command": "view",
                                   "path": "/workspace/main/java/io/spring/api/
                                   ProfileApi.java",
                                   "view_range": [40, 70]
                                })
```

After these 7 tool calls the agent could retrieve 3 out of 5 methods:

Here are the methods that call ``Optional<ProfileData> findByUsername(String username, User currentUser)`` in the production code:

1. ****In `RelationMutation` class**** (GraphQL mutation handler):
 - Method: ``buildProfile()`` (line 57)
 - Call: ``profileQueryService.findByUsername(username, current).get()``
2. ****In `ProfileApi` class**** (REST API controller):
 - Method: ``follow()`` (line 46)
 - Call: ``profileQueryService.findByUsername(username, user).get()``
 - Method: ``unfollow()`` (line 62)
 - Call: ``profileQueryService.findByUsername(username, user).get()``

These are the actual production code locations where this method is called. The calls appear in both GraphQL mutation handlers and REST API controllers.

While the Control group successfully identified two classes containing the relevant method invocation, it failed to retrieve the complete set of dependencies. Specifically, the system missed two additional Java classes that also contained valid method calls to the target method. Consequently, this partial recall resulted in a final Quality Score of 60%.

Metric	J-PRISM	Control (OpenHands)
Input Tokens	23,407	72,648
Output Tokens	300	1,189
Execution Time (s)	112	199
MCP Tool Calls	2	7
Quality Score	100%	60%

Table 5.6: Comparative Performance Metrics for the Success Case J-PRISM vs. Control (OpenHands)

The performance metrics, summarized in Table 5.6, highlights the efficiency of this graph-based approach.

5.6.2 Failure Case

This case study analyzes a regression observed in the Archetype II. The question-answer pair is from the Method Identification and Specificity Comprehension Area. The objective was to locate all constructors of a certain class:

Which constructor functions does the `UserData` class have?

This scenario tests whether the agent can retrieve and understand the Lombok annotation decorating the `UserData` class.

J-PRISM Execution Trace

Again, the J-PRISM system started with a similarity search:

I'll use the similarity search tool to find information about the `UserData` class and its constructor functions.

```
Function call: run_sim_search({"query":
    "UserData class constructor functions"
})
```

The similarity search returned general class information rather than explicit constructor methods. To compensate, J-PRISM constructed a specific Cypher query. However, the agent incorrectly filtered for method names starting with an `<init>` component. This false assumption was repeated in two successive MCP tool calls. As a result, the retrieval of the constructor methods failed.

Based on the information I've gathered, I can see that the `UserData` class is defined in the `io.spring.application.data` package. From the similarity search results and the class documentation, I can infer that the `UserData` class likely has a constructor that takes parameters for ID, email, username, bio, and image.

Let me try one more approach to find the constructor:

```
Function call: run_cypher_query({"query":
  "MATCH (c:CodeClass {name: 'UserData'})-[r:HAS_METHOD]->(m:CodeMethod)
  WHERE m.methodName = '<init>'
  OR m.methodName STARTS WITH 'constructor'
  RETURN c.name, m.methodName, m.inputParameters
"})
```

I notice that the constructor methods aren't showing up in the results. Let me try to find all constructors in the codebase and then filter for `UserData`:

```
Function call: run_cypher_query({"query":
  "MATCH (c:CodeClass)-[r:HAS_METHOD]->(m:CodeMethod)
  WHERE m.methodName STARTS WITH '<init>'
  RETURN c.name, m.methodName, m.inputParameters"
})
```

Following these method calls, the agent concluded that only the standard constructor method exists. While the system correctly identified all valid class parameters, it failed to detect the Lombok annotations. Consequently, the agent missed the existence of the second *No-Argument* constructor:

Based on the information available and the pattern I see in the codebase, I can conclude that the `UserData` class likely has a constructor that takes parameters for ID, email, username, bio, and image. This is suggested by:

1. The class documentation example: `UserData userData = new UserData("123", "user@example.com", "username123", "A sample bio.", "image.jpg");`
2. The presence of getter and setter methods for these fields (`getId`, `setId`, `getEmail`, `setEmail`, etc.)

However, the constructor method itself isn't explicitly showing up in the search results, which might be due to how constructors are represented in the codebase's knowledge graph.

OpenHands Execution Trace

The Control group performed similar steps as in the previous case. First, the agent located the `UserData` class and retrieved its full file content. Note that this identification part was iterative, requiring multiple MCP tool calls to isolate the correct file. Following the retrieval phase, the system accurately identified the presence of two Lombok annotations. The agent successfully reasoned that these annotations are responsible for the generation of two distinct constructors:

Based on my analysis of the `UserData` class in `./workspace/main/java/io/spring/application/data/UserData.java`, I can tell you that the `UserData` class has two constructor functions:

1. **Default Constructor (No-Arguments)**: Generated by the `@NoArgsConstructor` Lombok annotation
2. **All-Arguments Constructor**: Generated by the `@AllArgsConstructor` Lombok annotation, which takes parameters for all fields

The constructors would look like this:

```

` `` java
// Default constructor (no arguments)
public UserData() {}

// All-arguments constructor
public UserData(String id, String email, String username, String bio,
String image) {}
` ``

```

These constructors are automatically generated by Lombok based on the annotations used in the class definition.

Metric	J-PRISM	Control (OpenHands)
Input Tokens	53,178	56,648
Output Tokens	828	780
Execution Time (s)	156	231
MCP Tool Calls	3	6
Quality Score	50%	100%

Table 5.7: Comparative Performance Metrics for Method Identification Task: J-PRISM vs. Control (OpenHands)

Therefore, the Quality Score for the Control group was 100% for this question-answer pair.

As displayed in Table 5.7, the computational cost of both systems remained comparable. J-PRISM consumed similar input and output tokens. However, the quality metric differs. While the Control group could identify all functions the J-PRISM architecture could only derive one, resulting in a Quality Score drop of 50%.

5.6.3 Observed Behavioral Patterns

The analysis of the execution logs across all benchmarks revealed distinct behavioral patterns in the agent’s retrieval strategy. These patterns highlight both the strengths of the hybrid architecture and specific bottlenecks within the current LLM integration.

Verify Heuristic A consistent pattern emerged across the majority of tasks. The agent predominantly initiated the retrieval process with a vector-based similarity search. However, this was rarely the terminal step. In most successful traces, the agent utilized the result solely as a seeding step to formulate a precise Cypher query, although the result contained already the sufficient information to answer the question. This indicates a verification heuristic. The agent appears to prefer the deterministic nature of the Cypher retrieval tool.

Context Blindness in Similarity Search Despite the frequent usage, the efficiency of the similarity search tool result was limited. A critical limitation was observed in the LLM’s processing of the retrieval output. As noted in the success case (Section 5.6.1), the relevant information was often present within the JSON payload returned by the tool. Yet, the model frequently failed to extract this information. The agent often "hallucinated" that the data was missing, generating unnecessary subsequent queries. This indicates that the similarity search output requires further optimization to be more digestible for the model.

Schema Alignment The evaluation also exposed gaps in the agent’s semantic understanding of the KG schema. This was particularly evident in tasks requiring knowledge of specific Java conventions. As seen in the failure case (Section 5.6.2), the agent incorrectly assumed that the constructor methods followed the *<init>* internal naming convention within the graph. This reveals a misalignment between the agent’s internal training data and the specific abstraction layer of the J-PRISM graph.

Annotation Blindness Finally, a systemic neglect of meta-programming constructs was observed. The agent rarely initiated explicit searches for Java annotations. Unlike method calls or class definitions, annotations were not prioritized in the agent’s autonomous query formulations. This issue was unintentionally intensified by the current tool design. The similarity search tool is currently optimized to return Java Methods and Class nodes but does not prioritize Java Annotation nodes. Consequently, unless the agent explicitly queried for annotation relationships, this critical metadata remained invisible. This confirms that for frameworks like Lombok and Spring, the retrieval strategy requires adjustment to guide the LLM to consideration of implicit metadata.

5.7 Discussion

The experimental results presented in this chapter confirm the effectiveness of the J-PRISM architecture. The system demonstrated a clear advantage in navigating complex enterprise environments, achieving a high Quality Score in repositories where standard file-based retrieval failed. However, the evaluation also revealed specific limitations regarding model capacity, schema definition, and benchmark scope.

5.7.1 Impact of Model Capabilities

The evaluation was conducted using a quantized 24B parameter model to ensure reproducibility in local environments. The qualitative analysis indicates that this parameter limitation constrained the effective utilization of the Knowledge Graph. Specifically, the agent frequently showed "Context Blindness," failing to extract relevant information from the similarity search payload. Additionally, "Schema Alignment" errors were observed, where the agent hallucinated graph constraints such as specific naming conventions for constructors or Cypher syntax errors.

We hypothesize that larger foundation models, with superior reasoning and better instruction-following capabilities, would decrease these errors. We assume that large LLMs can directly answer from the initial vector retrieval or with more precise Cypher queries. Consequently, we suggest that future research explicitly investigates whether increasing model capacity correlates with improved reasoning accuracy and operational efficiency with the J-PRISM architecture.

5.7.2 Retrieval and Schema Limitations

While the hybrid retrieval approach outperformed the Control group, specific architectural blind spots remain. The regression observed in multiple Comprehension Areas highlights a possibility for improvement in the current schema design. The qualitative analysis identified behavioral patterns, where the agent failed to prioritize implicit metadata like Lombok annotations during search.

The current similarity search tool is optimized for explicit method and class nodes. Consequently, the agent struggles to resolve meta-programming constructs unless explicitly prompted. This indicates that the KG schema requires refinement to treat annotations as first-class semantic nodes, rather than secondary properties.

5.7.3 Validity of Benchmarks and Architecture

The benchmarks suite successfully categorized repositories by internal complexity. However, these datasets do not reflect the full diversity of the Java ecosystem. The evaluation focused heavily on modern Spring Boot architectures. Therefore, the findings may not fully generalize to legacy environments or alternative frameworks. Furthermore, the current evaluation relies on a single repository per archetype. To draw statistically sufficient conclusions regarding the system's comprehension ability relative to the repository complexity, the number of target repositories must be increased.

Additionally, the selected repositories represent only monolithic architectures. The current benchmarks do not validate the system's ability to trace control flow across distributed microservices or multi-server environments.

5.7.4 Conclusion

Despite these limitations, J-PRISM establishes a significant advancement over standard file-based retrieval. The presented data confirms that as the architectural complexity increases, the utility of semantic representation becomes critical. J-PRISM therefore serves as a validated stepping stone for the development of more robust, graph-based coding assistants in the field of meta-programming.

Conclusion and Future Work

6.1 Conclusion

This thesis investigated the limitations of standard Large Language Model (LLM) code comprehension in the context of enterprise Java development. Modern Java applications rely heavily on meta-programming frameworks such as Spring and Lombok. These frameworks introduce a layer of abstraction where control flow and logic are defined by annotations rather than explicit source code. Consequently, standard Retrieval Augmented Generation (RAG) approaches, which treat code as unstructured text, often fail to capture the implicit dependencies required for accurate reasoning.

To address this challenge, we introduced the J-PRISM (Java Property-graph Repository-level Insight System for Meta-programming) system. This novel architecture transforms static repositories into semantically enriched Knowledge Graphs (KG). By mapping code entities to structured nodes and edges, J-PRISM provides an AI agent with a deterministic interface to query the codebase.

The empirical evaluation confirms that the J-PRISM system significantly outperforms the standard file-based control group. The system achieved a mean Quality Score of 97.69%, compared to 89.37% for the baseline. Furthermore, J-PRISM demonstrated superior efficiency, reducing the global token consumption by 52.4%.

The results highlight a strong correlation between repository complexity and the utility of the KG. In simple CRUD applications, the performance of the graph-based approach and the file-based retrieval was equivalent. However, in more complex environments, the file-based control group could not keep up with the J-PRISM system. The J-PRISM approach maintained stability and achieved a 30.6% improvement rate with zero regressions in this repository category.

These findings validate the primary hypothesis of this research: providing an LLM with a structured, schema-driven representation of the code enhances its ability to reason about meta-programming constructs. The hybrid retrieval strategy, which combines symbolic Cypher queries with vector-based semantic search, proved to be a robust solution for navigating the repository complexity.

6.2 Future Work

While J-PRISM established a successful baseline for graph-enhanced code comprehension of Java repositories using meta-programming frameworks, the evaluation revealed specific limitations. Future research should focus on refining the system's schema, improving agent interaction, and expanding the scope of evaluation.

Knowledge Graph Schema Refinement The evaluation highlighted "Annotation Blindness," where the agent failed to prioritize implicit metadata like Lombok annotations. Although annotations are currently modeled as distinct nodes, the agent rarely prioritizes traversing these specific relations. Future iterations must refine the schema or the retrieval tools to increase the visibility of these relationships, ensuring the agent actively traces annotation dependencies.

Optimization of Agent-KG Interaction The evaluation exposed gaps in the agent's ability to interpret retrieval results. The agent frequently ignored valid answers contained within the similarity search payload. Additionally, "Schema Alignment" errors occurred where the agent hallucinated incorrect graph constraints. Future work should investigate using larger foundation models with stronger instruction-following capabilities or fine-tuning the MCP tool interaction.

Expansion of Benchmarks The current evaluation relied on a custom benchmark suite containing one repository per archetype. While sufficient for initial validation, this does not reflect the full diversity of the Java ecosystem. Future studies must expand these benchmarks to include a broader range of systems and alternative frameworks. Furthermore, future work should validate the system in distributed environments to test the graph's ability to trace control flow across microservices.

In conclusion, this thesis demonstrates that transitioning from unstructured file retrieval to structured graph-based lookup is essential for the next generation of AI coding assistants. J-PRISM provides a proven architectural blueprint for this transition. By addressing the outlined future work, this approach has the potential to solve the complexities of automated software engineering in enterprise environments.

Overview of Generative AI Tools Used

ChatGPT (OpenAI)

The *GPT-5.1 Thinking* model was primarily used as a coding assistant during the software development phase of J-PRISM. Its role was limited to generating boilerplate code, suggesting syntax optimizations, and debugging specific functions. I retained full control over the system architecture and logic.

Gemini Pro (Google)

The *Gemini Pro 2.5 / Pro 3* models were used to generate Python scripts for automated testing and to create plotting scripts for the evaluation of the experimental results. Furthermore, these models were utilized to refine the clarity, grammar, and stylistic consistency of the thesis. These tools assisted in rephrasing complex sentences and ensuring linguistic precision.

Integrity and Verification

The conceptualization, research design, core arguments, and conclusions presented in this document are entirely my own. I did not use AI tools to generate original ideas or scientific contributions. I maintained strict oversight over all AI-generated outputs. Every line of code, testing script, and text suggestion provided by these tools was rigorously reviewed, verified, and significantly modified to align with my specific requirements. I assume full responsibility for the accuracy and integrity of the content in this thesis.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1	DEL graph [20]	6
2.2	Heterogeneous graph [20]	7
2.3	Property graph [20]	7
2.4	Tool invocation with and without MCP. [22]	11
2.5	Changing the location of relevant information (in this case, the position of the passage that answers an input question) within the language model’s input context results in a U-shaped performance curve—models are better at using relevant information that occurs at the very beginning (primacy bias) or end of its input context (recency bias), and performance degrades significantly when models must access and use information located in the middle of its input context. [31]	14
2.6	A sample schema that defines the node types and relations between them. Every node in the knowledge graph will be one of these types linked by a relevant relation to another node. [5]	16
3.1	The architecture of the J-PRISM system.	22
3.2	Architecture of the main nodes in the KG.	23
3.3	Subgraph of the first 300 nodes of an example Java repository using the <i>Spring</i> framework and <i>Lombok</i>	25
3.4	Integrate vector representation into existing schema	26
3.5	Cypher query tool description which is visible to the LLM. The relations are presented in a "ASCII art" format which is used by the Cypher language.	27
3.6	Similarity search tool description which is visible to the LLM.	28
4.1	Visual representation of the intersection between the target item set I_{target} and the generated item set I_{gen}	36
5.1	The Efficiency vs. Effectiveness landscape.	38
5.2	Global distribution of Quality Scores comparing J-PRISM against the Control (OpenHands) baseline.	41
5.3	Head-to-Head Comparison: J-PRISM compared to the Control (OpenHands)	42
5.4	Quality Score Distributions by Repository Archetype	44
5.5	Performance Delta Analysis by Archetype	45
		73

5.6	Comparative analysis of Quality Score distributions across the four distinct comprehension domains. The violin plots illustrate that J-PRISM (purple) maintains higher density at the upper bound with reduced variance compared to the Control baseline (yellow) across (a), (b) and (d). J-PRISM has a lower performance in the Method Identification and Specificity area.	47
5.7	Pairwise performance analysis categorized by comprehension domain. The stacked bar charts quantify the distribution of improvement (green), equivalent performance (ties), and regression (red) of the J-PRISM architecture relative to the Control baseline across all repository archetypes.	48
5.8	The global token consumption of the J-PRISM approach and the Control (OpenHands).	49
5.9	Token consumption comparison of the J-PRISM approach and the Control (OpenHands) filtered by Comprehension Area and Repository Archetype.	50
5.10	The global output token usage of the J-PRISM approach and the Control (OpenHands).	51
5.11	Output Token consumption comparison of the J-PRISM approach and the Control (OpenHands) filtered by Comprehension Area and Repository Archetype.	52
5.12	Relationship between total computation load (Input and Output Tokens) and the execution time by J-PRISM and Control (OpenHands).	54
5.13	Comparing the amount of tool calls per task by J-PRISM and Control (OpenHands).	55
5.14	Attribution of Success Breakdown.	56
5.15	Quality Score Distribution of J-PRISM, Variant A, and Variant B	57
5.16	Comparing the total token consumption per task of J-PRISM, Variant A, and Variant B	58

List of Tables

5.1	Mean comparison of J-PRISM and Control (OpenHands) across Archetypes	43
5.2	Mean Quality Score comparison across the four comprehension domains. While J-PRISM demonstrates superior performance in structural and functional tasks, the Control baseline retains a slight advantage in specific method identification.	46
5.3	Mean Input Token Consumption Analysis.	51
5.4	Mean Output Token Consumption Analysis. This table compares the average cost of the generated response between J-PRISM and the Control group (OpenHands).	53
5.5	Comparative Performance Metrics for Ablation Variants	56
5.6	Comparative Performance Metrics for the Success Case J-PRISM vs. Control (OpenHands)	63
5.7	Comparative Performance Metrics for Method Identification Task: J-PRISM vs. Control (OpenHands)	66



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

- [1] *100M Token Context Windows — Magic*. en. URL: <https://magic.dev/blog/100m-token-context-windows> (visited on 11/25/2025).
- [2] *A Common Sense View of Knowledge Graphs* *AI3::Adaptive Information* *AI3::Adaptive Information*. URL: <https://www.mkbergman.com/2244/a-common-sense-view-of-knowledge-graphs/> (visited on 11/06/2025).
- [3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. „Foundations of Modern Query Languages for Graph Databases“. In: *ACM Comput. Surv.* 50.5 (Sept. 2017), 68:1–68:40. DOI: 10.1145/3104031. URL: <https://dl.acm.org/doi/10.1145/3104031> (visited on 11/10/2025).
- [4] *Apple stellt den M4 Pro und M4 Max vor - Apple (AT)*. URL: <https://www.apple.com/at/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/> (visited on 01/25/2026).
- [5] Mihir Athale and Vishal Vaddina. „Knowledge Graph Based Repository-Level Code Generation“. In: *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 2025, pp. 169–176.
- [6] Ivana Balazevic, Carl Allen, and Timothy Hospedales. „Multi-relational Poincaré Graph Embeddings“. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: <https://proceedings.neurips.cc/paper/2019/hash/f8b932c70d0b2e6bf071729a4fa68dfc-Abstract.html> (visited on 11/06/2025).
- [7] Piero Andrea Bonatti, Stefan Decker, Axel Polleres, and Valentina Presutti. „Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web (Dagstuhl Seminar 18371)“. In: *Dagstuhl Reports* 8.9 (2019). Ed. by Piero Andrea Bonatti, Stefan Decker, Axel Polleres, and Valentina Presutti. Place: Dagstuhl, Germany, pp. 29–111. DOI: 10.4230/DagRep.8.9.29. URL: <https://drops.dagstuhl.de/entities/document/10.4230/DagRep.8.9.29> (visited on 11/06/2025).

- [8] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. „Translating Embeddings for Modeling Multi-relational Data“. In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc., 2013. URL: <https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html> (visited on 11/11/2025).
- [9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. „Evaluating Large Language Models Trained on Code“. In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: <https://arxiv.org/abs/2107.03374> (visited on 12/27/2025).
- [10] Ajinkya Pankaj Deshpande, Anmol Agarwal, Shashank Shet, Arun Iyer, Aditya Kanade, Ramakrishna Bairi, and Suresh Parthasarathy. „Class-Level Code Generation from Natural Language Using Iterative, Tool-Enhanced Reasoning over Repository“. In: *CoRR* abs/2405.01573 (2024). arXiv: 2405.01573. DOI: 10.48550/ARXIV.2405.01573. URL: <https://doi.org/10.48550/arXiv.2405.01573> (visited on 12/29/2025).
- [11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. „BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding“. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: <https://aclanthology.org/N19-1423/> (visited on 11/11/2025).
- [12] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. „CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion“. In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. Ed. by Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine. 2023. URL: http://papers.nips.cc/paper%5C_files/paper/2023/

hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets%5C_and%5C_Benchmarks.html (visited on 12/29/2025).

- [13] Lisa Ehrlinger and Wolfram Wöß. „Towards a Definition of Knowledge Graphs“. In: *Joint Proceedings of the Posters and Demos Track of 12th International Conference on Semantic Systems - SEMANTiCS2016 and 1st International Workshop on Semantic Change & Evolving Semantics (SuCESS16)* 1695 (Sept. 2016). Place: Aachen, pp. 13–16. URL: <http://ceur-ws.org/Vol-1695/> (visited on 11/06/2025).
- [14] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. „Cypher: An Evolving Query Language for Property Graphs“. In: *Proceedings of the 2018 International Conference on Management of Data. SIGMOD '18*. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657. URL: <https://dl.acm.org/doi/10.1145/3183713.3190657> (visited on 11/11/2025).
- [15] Iason Gabriel, Arianna Manzini, Geoff Keeling, Lisa Anne Hendricks, Verena Rieser, Hasan Iqbal, Nenad Tomašev, Ira Ktena, Zachary Kenton, and Mikel Rodriguez. „The ethics of advanced ai assistants“. In: *arXiv preprint arXiv:2404.16244* (2024).
- [16] Aoran Gan, Hao Yu, Kai Zhang, Qi Liu, Wenyu Yan, Zhenya Huang, Shiwei Tong, and Guoping Hu. *Retrieval Augmented Generation Evaluation in the Era of Large Language Models: A Comprehensive Survey*. Apr. 2025. DOI: 10.48550/arXiv.2504.14891.
- [17] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. „Retrieval-Augmented Generation for Large Language Models: A Survey“. In: *CoRR* abs/2312.10997 (2023). arXiv: 2312.10997. DOI: 10.48550/ARXIV.2312.10997. URL: <https://doi.org/10.48550/arXiv.2312.10997> (visited on 02/03/2026).
- [18] *gothinkster/spring-boot-realworld-example-app: Example Spring codebase containing real world examples (CRUD, auth, advanced patterns, etc) that adheres to the RealWorld API spec*. URL: <https://github.com/gothinkster/spring-boot-realworld-example-app> (visited on 01/22/2026).
- [19] Nadeeshaan Gunasinghe and Nipuna Marcus. *Language server protocol and implementation*. Springer, 2021.
- [20] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D’amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. „Knowledge Graphs“. In: *ACM Comput. Surv.* 54.4 (July 2021), 71:1–71:37. DOI: 10.1145/3447772. URL: <https://dl.acm.org/doi/10.1145/3447772> (visited on 11/05/2025).
- [21] *Home*. en. URL: <https://spring.io/> (visited on 10/29/2025).

- [22] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. „Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions“. In: *CoRR* abs/2503.23278 (2025). arXiv: 2503.23278. DOI: 10.48550/ARXIV.2503.23278. URL: <https://doi.org/10.48550/arXiv.2503.23278> (visited on 02/03/2026).
- [23] Rana Hussein, Dingqi Yang, and Philippe Cudré-Mauroux. „Are Meta-Paths Necessary? Revisiting Heterogeneous Graph Embeddings“. In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. CIKM '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 437–446. DOI: 10.1145/3269206.3271777. URL: <https://dl.acm.org/doi/10.1145/3269206.3271777> (visited on 11/06/2025).
- [24] International Organization for Standardization. *Information technology — Database languages — GQL*. Apr. 2024. URL: <https://www.iso.org/standard/76120.html>.
- [25] *Introducing the Model Context Protocol*. en. URL: <https://www.anthropic.com/news/model-context-protocol> (visited on 11/18/2025).
- [26] *JavaParser - Home*. URL: <https://javaparser.org/> (visited on 12/15/2025).
- [27] Sayash Kapoor, Benedikt Stroebel, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. „AI Agents That Matter“. In: *Trans. Mach. Learn. Res.* 2025 (2025). URL: <https://openreview.net/forum?id=Zy4uFzMviZ> (visited on 02/03/2026).
- [28] Philippe Laban, Alexander Fabbri, Caiming Xiong, and Chien-Sheng Wu. „Summary of a Haystack: A Challenge to Long-Context LLMs and RAG Systems“. In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Ed. by Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 9885–9903. DOI: 10.18653/v1/2024.emnlp-main.552. URL: <https://aclanthology.org/2024.emnlp-main.552/> (visited on 11/25/2025).
- [29] Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. „EvoCodeBench: An Evolving Code Generation Benchmark with Domain-Specific Evaluations“. In: *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*. Ed. by Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang. 2024. URL: http://papers.nips.cc/paper%5C_files/paper/2024/hash/6a059625a6027aca18302803743abaa2-Abstract-Datasets%5C_and%5C_Benchmarks%5C_Track.html.
- [30] Xinzhe Li. „A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning“. In: *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu*

- Dhabi, UAE, January 19-24, 2025*. Ed. by Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert. Association for Computational Linguistics, 2025, pp. 9760–9779. URL: <https://aclanthology.org/2025.coling-main.652/> (visited on 02/03/2026).
- [31] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. „Lost in the Middle: How Language Models Use Long Contexts“. In: *Trans. Assoc. Comput. Linguistics* 12 (2024), pp. 157–173. DOI: 10.1162/TACL_A_00638. URL: https://doi.org/10.1162/tacl%5C_a%5C_00638 (visited on 02/03/2026).
- [32] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. *CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases*. arXiv:2408.03910 [cs]. Aug. 2024. DOI: 10.48550/arXiv.2408.03910. URL: <http://arxiv.org/abs/2408.03910> (visited on 10/05/2025).
- [33] *LM Studio - Local AI on your computer*. URL: <https://lmstudio.ai/> (visited on 01/25/2026).
- [34] Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. „Query Rewriting in Retrieval-Augmented Large Language Models“. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 5303–5315. DOI: 10.18653/v1/2023.emnlp-main.322. URL: <https://aclanthology.org/2023.emnlp-main.322/> (visited on 11/25/2025).
- [35] *macrozheng/mall-tiny: mall-tinySpringBoot+MyBatis-PlusVue*. URL: <https://github.com/macrozheng/mall-tiny> (visited on 01/22/2026).
- [36] Yu A. Malkov and D. A. Yashunin. „Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (Apr. 2020), pp. 824–836. DOI: 10.1109/TPAMI.2018.2889473. URL: <https://ieeexplore.ieee.org/abstract/document/8594636> (visited on 11/11/2025).
- [37] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. „Efficient Estimation of Word Representations in Vector Space“. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. arXiv, 2013. URL: <http://arxiv.org/abs/1301.3781> (visited on 02/03/2026).
- [38] *Neo4j Graph Database & Analytics – The Leader in Graph Databases*. en. Nov. 2025. URL: <https://neo4j.com/> (visited on 11/10/2025).

- [39] Natasha Noy, Search about this author, Yuqing Gao, Search about this author, Anshu Jain, Search about this author, Anant Narayanan, Search about this author, Alan Patterson, Search about this author, Jamie Taylor, and Search about this author. „Industry-scale Knowledge Graphs: Lessons and Challenges“. In: *Queue* 17.2 (Apr. 2019), pp. 48–75. DOI: 10.1145/3329781.3332266. URL: <https://spawn-queue.acm.org/doi/abs/10.1145/3329781.3332266> (visited on 11/06/2025).
- [40] Wenjun Peng, Guiyang Li, Yue Jiang, Zilong Wang, Dan Ou, Xiaoyi Zeng, Derong Xu, Tong Xu, and Enhong Chen. „Large language model based long-tail query rewriting in taobao search“. In: *Companion Proceedings of the ACM Web Conference 2024*. 2024, pp. 20–28.
- [41] *Project Lombok*. URL: <https://projectlombok.org/> (visited on 10/29/2025).
- [42] *Qwen*. URL: <https://qwen.ai/blog?id=qwen2.5-coder-family> (visited on 01/25/2026).
- [43] *Qwen*. URL: <https://qwen.ai/blog?id=qwen3-embedding> (visited on 01/25/2026).
- [44] *RameshMF/springboot-blog-rest-api: Source code of course - Building Real-Time REST APIs with Spring Boot*. URL: <https://github.com/RameshMF/springboot-blog-rest-api> (visited on 01/22/2026).
- [45] *RDF 1.1 Concepts and Abstract Syntax*. URL: <https://www.w3.org/TR/rdf11-concepts/> (visited on 11/06/2025).
- [46] Nils Reimers and Iryna Gurevych. „Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks“. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Ed. by Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan. Association for Computational Linguistics, 2019, pp. 3980–3990. DOI: 10.18653/v1/D19-1410. URL: <https://doi.org/10.18653/v1/D19-1410> (visited on 02/03/2026).
- [47] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. eng. Third edition, Global edition. Prentice Hall series in artificial intelligence. Boston Columbus Indianapolis: Pearson, 2016.
- [48] G. Salton, A. Wong, and C. S. Yang. „A vector space model for automatic indexing“. In: *Commun. ACM* 18.11 (Nov. 1975), pp. 613–620. DOI: 10.1145/361219.361220. URL: <https://dl.acm.org/doi/10.1145/361219.361220> (visited on 11/11/2025).
- [49] E. W. Schneider. *Course Modularization Applied: The Interface System and Its Implications For Sequence Control and Data Analysis*. en. Tech. rep. ERIC Number: ED088424. Nov. 1973. URL: <https://eric.ed.gov/?id=ED088424> (visited on 11/06/2025).

- [50] Yonadav Shavit, Sandhini Agarwal, Miles Brundage, Steven Adler, Cullen O’Keefe, Rosie Campbell, Teddy Lee, Pamela Mishkin, Tyna Eloundou, and Alan Hickey. „Practices for governing agentic AI systems“. In: *Research Paper, OpenAI* (2023).
- [51] *SPARQL Query Language for RDF*. URL: <https://www.w3.org/TR/rdf-sparql-query/> (visited on 11/10/2025).
- [52] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. „RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space“. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: <https://openreview.net/forum?id=HkgEQnRqYQ> (visited on 02/03/2026).
- [53] *The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation*. en. URL: <https://ai.meta.com/blog/llama-4-multimodal-intelligence/> (visited on 11/25/2025).
- [54] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Eric Gaussier, and Guillaume Bouchard. „Complex Embeddings for Simple Link Prediction“. en. In: *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, June 2016, pp. 2071–2080. URL: <https://proceedings.mlr.press/v48/trouillon16.html> (visited on 11/11/2025).
- [55] *Upgrading agentic coding capabilities with the new Devstral models | Mistral AI*. URL: <https://mistral.ai/news/devstral-2507> (visited on 01/25/2026).
- [56] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, and Yankai Lin. „A survey on large language model based autonomous agents“. In: *Frontiers of Computer Science* 18.6 (2024). ISBN: 2095-2228, p. 186345.
- [57] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. „Heterogeneous Graph Attention Network“. In: *The World Wide Web Conference. WWW ’19*. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 2022–2032. DOI: 10.1145/3308558.3313562. URL: <https://dl.acm.org/doi/10.1145/3308558.3313562> (visited on 11/06/2025).
- [58] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al et. „OpenHands: An Open Platform for AI Software Developers as Generalist Agents“. In: *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL: <https://openreview.net/forum?id=OJd3ayDDoF> (visited on 02/03/2026).
- [59] *Welcoming Diverse Approaches Keeps Machine Learning Strong*. en. June 2024. URL: <https://www.deeplearning.ai/the-batch/welcoming-diverse-approaches-keeps-machine-learning-strong/> (visited on 11/18/2025).

- [60] Lilian Weng. *LLM Powered Autonomous Agents*. en. Section: posts. June 2023. URL: <https://lilianweng.github.io/posts/2023-06-23-agent/> (visited on 11/18/2025).
- [61] *Wikidata*. URL: https://www.wikidata.org/wiki/Wikidata:Main_Page (visited on 01/30/2026).
- [62] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. „Embedding Entities and Relations for Learning and Inference in Knowledge Bases“. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. arXiv, 2015. URL: <http://arxiv.org/abs/1412.6575> (visited on 02/01/2026).
- [63] Kisung You. „Semantics at an Angle: When Cosine Similarity Works Until It Doesn’t“. In: *CoRR* abs/2504.16318 (2025). arXiv: 2504.16318. DOI: 10.48550/ARXIV.2504.16318. URL: <https://doi.org/10.48550/arXiv.2504.16318> (visited on 02/03/2026).
- [64] Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H. Chi, Quoc V. Le, and Denny Zhou. „Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models“. In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. Open-Review.net, 2024. URL: <https://openreview.net/forum?id=3bq3jsvcQ1> (visited on 02/03/2026).