NANYANG TECHNOLOGICAL UNIVERSITY

NEXT-GEN GENERATIVE SEARCH ENGINE FOR CODE BASE

Timothy Lee Hongyi

College of Computing and Data Science

2026

NANYANG TECHNOLOGICAL UNIVERSITY

SC4079

NEXT-GEN GENERATIVE SEARCH ENGINE FOR CODE BASE

Submitted in Partial Fulfilment of the Requirements
for the Degree of Bachelor of Computing in Data Science and Artificial Intelligence
of the Nanyang Technological University

by

Timothy Lee Hongyi

College of Computing and Data Science

2026

Abstract

This report presents the design, implementation and evaluation of CodeOrient, an Artificial Intelligence (AI) Search tool, designed to accelerate developer onboarding. This application leverages Natural Language Processing (NLP), code graph visualisation and vector-based retrieval to help new developers understand unfamiliar codebases and reduce the time to first commit.

CodeOrient integrates semantic code search using embedding models, structural analysis through dependency graphs and generative user interfaces (UI) to provide context-aware feature cards. Through the use of Retrieval-Augemented Generation (RAG) with source grounding, the application eliminates hallucination, commonly seen in AI code assistants. Preliminary testing suggests that by externalising the mental model of a codebase through a unified graph-and-card interface, the system significantly reduces the cognitive load associated with onboarding and system discovery in large-scale repositories.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Professor Tan Chee Wei, for his unwavering guidance and patience throughout the duration of this Final Year Project. His expertise and insightful suggestions were key in shaping the direction of this project. I am very honoured to have had the opportunity to work under his mentorship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Problem Statement

1.1.1 Developer Onboarding Challenges

Time-to-First-Commit as a Metric

1.2 Research Objectives and Goals

1.3 Contributions

1.4 Report Structure

Chapter 2

Literature Review

Understanding unfamiliar and large codebases poses significant challenges for software developers, especially those new to a project or organisation. Different organisations have various ways to onboard new developers, yet these approaches to codebase exploration remain inefficient. As AI adoption increases, the problem of accurate code comprehension becomes paramount. Often, AI-assisted tools hallucinate information that is not grounded in actual source code. Furthermore, large codebases makes it harder for AI to comprehend due its limited context window.

This literature review examines five interrelated research domains critical to the proposed AI Search Tool - CodeOrient:

1. Software Complexity Metrics and Code Quality Measurements
2. Semantic Code Search and Natural Language Query Processing
3. AI Hallucination and Grounded Code Comprehension
4. Code Visualisation and Dependency Analysis
5. Generative User Interfaces

The combination of these areas provides the theoretical foundation for building an AI application that reduces developer onboarding time while maintaining citation accuracy and reliability.

2.1 Software Complexity Metrics and Code Quality Measurements

2.1.1 Cyclomatic Complexity as Foundation

Thomas J.McCabe introduced cyclomatic complexity, a quantitative measure of program complexity based on control graph flow analysis [1]. His work established that cyclomatic complexity directly correlates with code maintainability and testing. The formula $M = E - N + 2P$ where E represents edges, N represents nodes, and P represents the number of connected components in a control flow graph, represents the complexity as the number of linearly independent paths through a program's source code. As shown in Fig.1, a simple control flow graph of a function below yields a complexity of 2, where $P = 1$.

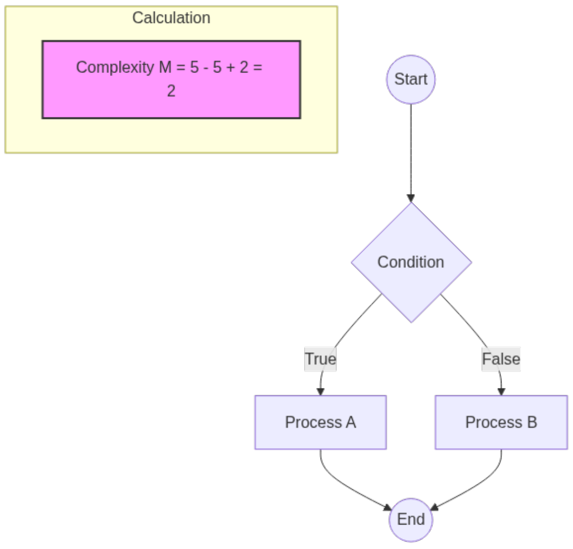

Figure 2.1: *Control flow graph of a simple if-else statement.*

McCabe's complexity measure allows developers to identify highly complex functions and recognise problematic code sections that require refactoring. When implementing the code graph visualisation feature in a code search application, cyclomatic complexity serves as one signal among many to highlight high-risk or critical code sections.

2.1.2 Modern Complexity in Distributed Systems

Kafura's recent reflection on McCabe's work in 2025 acknowledges that cyclomatic complexity has proven durable for the last 50 years. However, modern software architectures such as distributed systems and microservices, require additional metrics beyond control flow analysis [2]. This observation directly motivates the integration of code graph visualisation in modern development tools, as visual representation of information flow across procedures becomes just as important as understanding control flow within individual functions.

2.2 Code Search and Natural Language Query Processing

2.2.1 Semantic Code Search Through Embeddings

With the recent advancements in neural networks and embedding models, the field of semantic code search has evolved significantly from traditional keyword-based search. Cambronero et al. demonstrated that the use of neural embeddings could bridge the semantic gap between natural language queries and code snippets [3]. By transforming both code and queries into a shared vector space, code snippets relevant to the query can be retrieved by calculating the cosine similarity between their embedding vectors, given by the formula:

$$\text{cosine\_similarity}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}||\,||\vec{b}||}$$

where $\vec{a}$ and $\vec{b}$ are the embedding vectors of the query and code snippet, respectively.

In the figure below, developers can express their intent through natural language, which is then mapped into the same embedding space as code snippets. This technique supplements existing search such as fuzzy search or keyword-based approaches that often miss semantically relevant results. This motivates the use of natural language search in modern code repositionaries like GitHub.

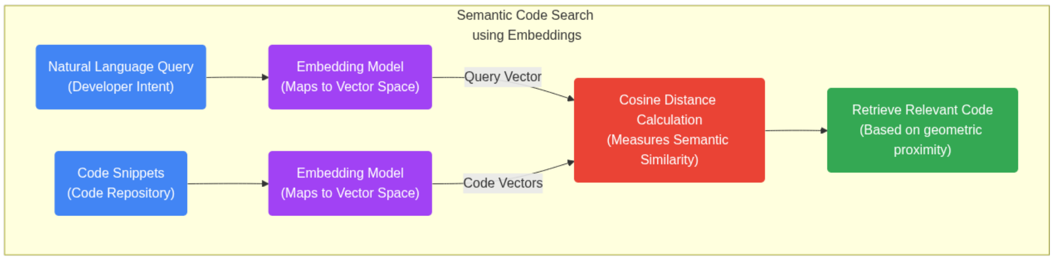
Semantic Code Search using Embeddings
Natural Language Query (Developer Intent)
Embedding Model (Maps to Vector Space)
Query Vector
Code Snippets (Code Repository)
Embedding Model (Maps to Vector Space)
Code Vectors
Cosine Distance Calculation (Measures Semantic Similarity)
Retrieve Relevant Code (Based on geometric proximity)

Figure 2.2: *Semantic code search in a shared embedding space for retrieval [3].*

2.2.2 Structural Code Search with Domain-Specific Languages

Recent research by Limpanukorn et al. (2025) introduces structural code search by translating natural language queries in Doman-Specific Language (DSL) queries for more precise code retrieval [4]. This approach leverages the architectural information embedded in a codebase rather than just its textual meaning. This method achieved a precision score of 55-70% and outperformed semantic search baselines by up to 57% on F1 scores [4].

This research presents a critical missing link in developer tools, such as tracing an ”authentication flow”, where the relationships between modules are more informative than the functions’ names themselves. The findings provide a strong validation for the code graph visualisation tool, providing syntactic and architectural information for developers to understand complex problem spaces.

2.2.3 Code Embedding Models for Retrieval

In 2025, Qodo introduced specialised code embedding models (Qodo Embed-1) that achieved state-of-the-art performance in Codebase Understanding Gartner® 2025, with a product score of 3.72/5. Their approach directly encodes code semantics without an intermediate language description step [5]. By avoiding the overhead of the intermediate step, Qodo Embed-1 has proved to be computationally efficient while maintaining high retrieval accuracy.

2.3 AI Hallucination and Grounded Code Comprehension

2.3.1 The Hallucination Problem in Code-Generating LLMs

With the rise in AI-assisted code generation tools such as GitHub Copilot and ChatGPT, the problem of hallucination becomes increasingly critical. Hallucination in refers to the generation of information that appears plausible but is factually incorrect or fabricated. Spracklen et al.'s (2025) research revealed that package hallucinations are a systemic issue across state-of-the-art code-generating models [6]. They analysed over 576,000 generated code samples across 16 Large Language Models (LLMs) and found that the LLMs consistently hallucinate package names. More critically, they regenerate the same false package name across 43% of repeated prompts [6].

This poses a critical issue in agentic workflow as the hallucination is exploitable via ”slopsquatting”, creating risks in the software supply chain [6]. The figure below showcases how an attacker can exploit hallucinations from LLMs. This research highlights the urgent need for citation-grounded code comprehension systems that can verify all claims against actual source code.

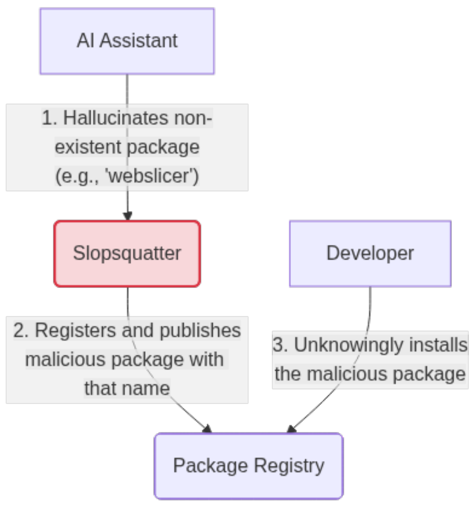
AI Assistant
1. Hallucinates non-existent package (e.g., 'webslicer')
Slopsquatter
Developer
2. Registers and publishes malicious package with that name
3. Unknowingly installs the malicious package
Package Registry

Figure 2.3: *Exploiting LLM hallucinations through slopsquatting.*

2.3.2 Citation-Grounded Code Comprehension

Arafat et al.'s recent work on citation-grounded code comprehension directly addresses the hallucination problem [7]. They conclude that code comprehension systems must ground all claims in verifiable source code citations. Their proposed hybrid retrieval system with Neo4j graph database to provide import relationships, achieved a 92% citation accuracy with zero hallucinations. Moreover, the graph component discovered richer cross-file relationships that purely text-based retrieval missed, 62% of architectural queries [7].

2.3.3 Retrieval-Augmented Generation

The broader principle emerging from hallucination research is Retrieval-Augmented Generation (RAG). As it is not possible to train new information into LLMs at scale, RAG provides a mechanism to ground LLM outputs with real-time data via a retriever [8]. The use of a retriever reduced hallucination rates across all categories from a baseline high of 21% to below 7.5% [8]. In the context of code comprehension, RAG refers to a retriever that fetches relevant code snippets, which are then passed to an LLM as context to generate grounded explanations with source citations.

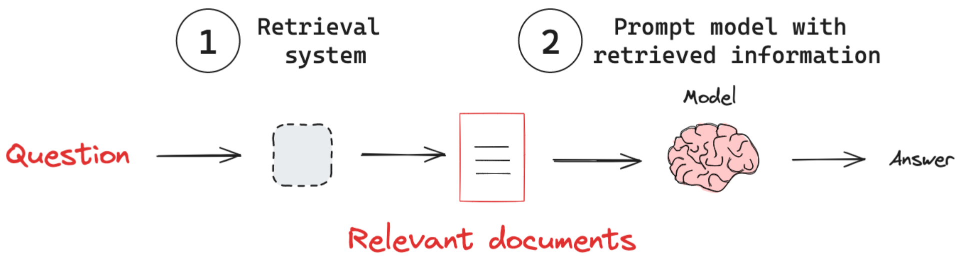

Figure 2.4: *Retrieval-Augmented Generation to reduce hallucinations [7].*

2.4 Code Visualisation and Dependency Analysis

Chapter 3

System Design and Architecture

3.1 System Overview

3.1.1 Core Components

3.1.2 Data Flow Architecture

3.1.3 Microservices Structure

3.2 Frontend Architecture

3.2.1 Technology Stack

3.2.2 User Interface Design

3.2.3 Interactive Graph Visualization with React Flow

3.2.4 Generative UI Card Components

3.3 Backend Architecture

3.3.1 API Design

3.3.2 Code Search Engine

3.3.3 Dependency Graph Generation

3.3.4 LLM Integration and RAG Pipeline

Chapter 4

Implementation Details

4.1 Development Methodology

4.1.1 Iterative Development Process

4.1.2 Version Control and Branching Strategy

4.2 Core Implementation Components

4.2.1 Search Module

BM25 Sparse Retrieval

Dense Embedding Retrieval

Hybrid Search Integration

4.2.2 Code Graph Analysis

Dependency Extraction

Graph Construction Algorithm

Graph Traversal and Ranking

4.2.3 Generative UI System

Query-to-Task Model Translation

Task Model to UI Specification

Chapter 5

Evaluation Methodology

5.1 Evaluation Framework

5.1.1 Research Questions

5.1.2 Hypotheses

5.1.3 Evaluation Metrics

5.2 User Study Design

5.2.1 Participant Selection

5.2.2 Study Protocol

5.2.3 Task Design

5.2.4 Baseline Comparisons

5.3 Quantitative Metrics

5.3.1 Time-to-First-Commit

5.3.2 Time to Complete Onboarding Tasks

5.3.3 Code Search Precision and Recall

5.3.4 Graph Visualization Quality

Chapter 6

Results and Analysis

6.1 Quantitative Results

6.1.1 Time-to-First-Commit Analysis

6.1.2 Search Performance Evaluation

6.1.3 Citation Accuracy Results

6.1.4 System Performance Metrics

Query Latency

Throughput

Resource Utilisation

6.2 Qualitative Results

6.2.1 User Satisfaction Findings

6.2.2 Usability Observations

6.2.3 Thematic Analysis of Feedback

6.3 Comparative Analysis with Baselines

6.3.1 Performance vs. GitHub Search

Chapter 7

Discussion

7.1 Key Findings

7.1.1 Effectiveness of Citation Grounding

7.1.2 Impact of Code Graph Visualization

7.1.3 Generative UI Effectiveness

7.1.4 Hybrid Retrieval Strategy Benefits

7.2 Implications for Developer Onboarding

7.3 Limitations

7.3.1 Scale Limitations

7.3.2 Language and Framework Coverage

7.3.3 User Study Scope

7.3.4 Computational Resource Requirements

7.4 Comparison with Related Work

7.4.1 How We Advance Beyond Existing Solutions

7.4.2 Unique Contributions

Chapter 8

Conclusion

8.1 Summary of Contributions

8.2 Key Takeaways

8.3 Future Work

8.3.1 Scalability Improvements

8.3.2 Multi-Language Support

8.3.3 Advanced Generative UI

8.3.4 Integration with Developer Tools

8.3.5 Real-Time Collaborative Features

8.4 Recommendations for Implementation

8.5 Final Remarks

Appendix A

User Study Materials

A.1 Participant Consent Form

A.2 Study Instructions

A.3 Task Descriptions

A.4 Survey Questionnaire

A.5 Interview Questions

Appendix B

Detailed Results Tables

B.1 Quantitative Results Tables

B.2 User Feedback Summary

B.3 Performance Metrics by Repository

Appendix C

Code Snippets and Implementation Details

C.1 Key Algorithm Implementations

C.2 API Endpoint Specifications

C.3 Database Schema SQL

C.4 Frontend Component Code

C.5 RAG Pipeline Code

Appendix D

System Architecture Diagrams

Appendix E

Additional Evaluation Data

E.1 Raw User Study Data

E.2 Search Performance Analysis

E.3 Citation Accuracy Detailed Results

E.4 Latency Distributions

Appendix F

Related Work Comparison Table

F.1 Feature Comparison Matrix

F.2 Performance Comparison

Bibliography

[1] T.J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[2] Dennis Kafura. “Reflections on McCabe’s Cyclomatic Complexity”. In: *IEEE Transactions on Software Engineering* 51.3 (2025), pp. 700–705. DOI: 10.1109/TSE.2025.3534580.

[3] José Cambronero et al. “When Deep Learning Met Code Search”. In: *CoRR* abs/1905.03813 (2019). arXiv: 1905.03813. URL: http://arxiv.org/abs/1905.03813.

[4] Ben Limpanukorn et al. *Structural Code Search using Natural Language Queries*. 2025. arXiv: 2507.02107 [cs.SE]. URL: https://arxiv.org/abs/2507.02107.

[5] Sheffer Tai. *State-of-the-Art Code Retrieval with Efficient Embeddings*. 2025. URL: https://www.qodo.ai/blog/qodo-embed-1-code-embedding-code-retrieval/.

[6] Joseph Spracklen et al. *We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs*. 2025. arXiv: 2406.10279 [cs.SE]. URL: https://arxiv.org/abs/2406.10279.

[7] Jahidul Arafat. *Citation-Grounded Code Comprehension: Preventing LLM Hallucination Through Hybrid Retrieval and Graph-Augmented Context*. 2025. arXiv: 2512.12117 [cs.SE]. URL: https://arxiv.org/abs/2512.12117.

[8] Orlando Ayala and Patrice Bechard. “Reducing hallucination in structured outputs via Retrieval-Augmented Generation”. In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics:*

Human Language Technologies (Volume 6: Industry Track). Association for Computational Linguistics, 2024, pp. 228–238. DOI: `10.18653/v1/2024.naacl-industry.19`. URL: `http://dx.doi.org/10.18653/v1/2024.naacl-industry.19`.