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

Abstract i

Acknowledgments ii

Contents iii

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 Motivation and Problem Statement 1

1.1.1 Developer Onboarding Challenges 1

1.2 Research Objectives and Goals 1

1.3 Contributions 1

1.4 Report Structure 1

2 Literature Review 2

2.1 Software Complexity Metrics and Code Quality Measurements 3

2.1.1 Cyclomatic Complexity as Foundation 3

2.1.2 Modern Complexity in Distributed Systems 4

2.2 Code Search and Natural Language Query Processing 4

2.2.1 Semantic Code Search Through Embeddings 4

2.2.2 Structural Code Search with Domain-Specific Languages 5

2.2.3 Code Embedding Models for Retrieval 5

2.3 AI Hallucination and Grounded Code Comprehension 6

2.3.1 The Hallucination Problem in Code-Generating LLMs 6

2.3.2 Citation-Grounded Code Comprehension 7
2.3.3 Retrieval-Augmented Generation 7
2.4 Code Visualisation and Dependency Analysis 7
2.4.1 Static Analysis and Dependency Graphs 7
2.4.2 Interactive Visualisation Tools 8
2.5 Generative User Interface (UI) 8
2.5.1 Generative UI as a Paradigm 8
2.5.2 Task-Driven Data Models for Adaptive Interfaces 8
2.6 Related Systems and Tools 9
2.7 Research Gaps and Motivation 9
2.7.1 Bridging Semantic and Structural Code Search 9
2.7.2 The ”Black Box” of Generative UI in Software Engineering 9
2.8 Conclusion 10

3 System Design and Architecture **11**
3.1 System Overview 12
3.1.1 Core Components 12
3.1.2 Data Flow Architecture 12
3.1.3 Microservices Structure 12
3.2 Frontend Architecture 12
3.2.1 Technology Stack 12
3.2.2 User Interface Design 12
3.2.3 Interactive Graph Visualization with React Flow 12
3.2.4 Generative UI Card Components 12
3.3 Backend Architecture 12
3.3.1 API Design 12
3.3.2 Code Search Engine 12
3.3.3 Dependency Graph Generation 12
3.3.4 LLM Integration and RAG Pipeline 12
3.4 Data Pipeline 12
3.4.1 Repository Ingestion 12

3.4.2 Code Parsing and AST Analysis 12
3.4.3 Embedding Generation 12
3.4.4 Dependency Graph Construction 12
3.4.5 Indexing and Storage 12
3.5 Natural Language Processing Pipeline 12
3.5.1 Query Understanding 12
3.5.2 Semantic Search using Embeddings 12
3.5.3 Structural Search using DSL Translation 12
3.5.4 Hybrid Retrieval Strategy 12
3.6 Citation-Grounded Generation 12
3.6.1 RAG Architecture 12
3.6.2 Source Code Grounding 12
3.6.3 Citation Mechanism 12
3.7 Technical Stack Justification 12
3.7.1 Frontend: Next.js 12
3.7.2 Backend: Next.js API Routes 12
3.7.3 LLM Integration: Vercel AI SDK and OpenAI 12
3.7.4 Code Search: GitHub Search API 12
3.7.5 Graph Visualization: React Flow 12
3.7.6 Database: PostgreSQL 12
3.7.7 Caching: Upstash Redis 12
3.7.8 Deployment: Vercel 12

4 Implementation Details **13**

4.1 Development Methodology 14
4.1.1 Iterative Development Process 14
4.1.2 Version Control and Branching Strategy 14
4.2 Core Implementation Components 14
4.2.1 Search Module 14
4.2.2 Code Graph Analysis 14
4.2.3 Generative UI System 14

4.2.4 RAG Pipeline 14
4.3 API Endpoints and Data Models 14
4.4 Database Schema 14
4.5 Code Examples and Key Algorithms 14
4.5.1 Cyclomatic Complexity Calculation 14
4.5.2 Dependency Graph Construction 14
4.5.3 Embedding-Based Similarity Search 14
4.5.4 RAG Citation Grounding 14
4.6 Challenges and Solutions 14
4.6.1 Handling Large Codebases 14
4.6.2 Real-Time Graph Updates 14
4.6.3 Latency Optimization 14
4.6.4 Hallucination Prevention 14
4.6.5 Language and Framework Diversity 14

5 Evaluation Methodology **15**
5.1 Evaluation Framework 16
5.1.1 Research Questions 16
5.1.2 Hypotheses 16
5.1.3 Evaluation Metrics 16
5.2 User Study Design 16
5.2.1 Participant Selection 16
5.2.2 Study Protocol 16
5.2.3 Task Design 16
5.2.4 Baseline Comparisons 16
5.3 Quantitative Metrics 16
5.3.1 Time-to-First-Commit 16
5.3.2 Time to Complete Onboarding Tasks 16
5.3.3 Code Search Precision and Recall 16
5.3.4 Graph Visualization Quality 16
5.3.5 Citation Accuracy and Hallucination Rate 16

5.4 Qualitative Metrics 16
5.4.1 User Satisfaction Survey 16
5.4.2 Usability Assessment 16
5.4.3 Cognitive Load Measurement 16
5.4.4 User Feedback and Interviews 16
5.5 Baseline Comparisons 16
5.5.1 Traditional Code Search (GitHub Search) 16
5.5.2 Existing AI Code Assistants 16
5.5.3 Manual Code Navigation 16
5.6 Statistical Analysis 16
5.6.1 Hypothesis Testing 16
5.6.2 Effect Size Calculation 16
5.6.3 Confidence Intervals 16

6 Results and Analysis **17**
6.1 Quantitative Results 18
6.1.1 Time-to-First-Commit Analysis 18
6.1.2 Search Performance Evaluation 18
6.1.3 Citation Accuracy Results 18
6.1.4 System Performance Metrics 18
6.2 Qualitative Results 18
6.2.1 User Satisfaction Findings 18
6.2.2 Usability Observations 18
6.2.3 Thematic Analysis of Feedback 18
6.3 Comparative Analysis with Baselines 18
6.3.1 Performance vs. GitHub Search 18
6.3.2 Performance vs. AI Code Assistants 18
6.3.3 Performance vs. Manual Navigation 18
6.4 Case Studies 18
6.4.1 Case Study 1: Understanding the Authentication Flow 18
6.4.2 Case Study 2: API Endpoint Discovery 18

6.4.3 Case Study 3: Error Handling Patterns 18
6.5 Data Visualisation and Interpretation 18

7 Discussion **19**
7.1 Key Findings 20
7.1.1 Effectiveness of Citation Grounding 20
7.1.2 Impact of Code Graph Visualization 20
7.1.3 Generative UI Effectiveness 20
7.1.4 Hybrid Retrieval Strategy Benefits 20
7.2 Implications for Developer Onboarding 20
7.3 Limitations 20
7.3.1 Scale Limitations 20
7.3.2 Language and Framework Coverage 20
7.3.3 User Study Scope 20
7.3.4 Computational Resource Requirements 20
7.4 Comparison with Related Work 20
7.4.1 How We Advance Beyond Existing Solutions 20
7.4.2 Unique Contributions 20
7.5 Threats to Validity 20
7.5.1 Internal Validity 20
7.5.2 External Validity 20
7.5.3 Construct Validity 20
7.6 Practical Implications for Industry 20

8 Conclusion **21**
8.1 Summary of Contributions 21
8.2 Key Takeaways 21
8.3 Future Work 21
8.3.1 Scalability Improvements 21
8.3.2 Multi-Language Support 21
8.3.3 Advanced Generative UI 21
8.3.4 Integration with Developer Tools 21

8.3.5 Real-Time Collaborative Features 21
8.4 Recommendations for Implementation 21
8.5 Final Remarks 21

A User Study Materials **22**
A.1 Participant Consent Form 22
A.2 Study Instructions 22
A.3 Task Descriptions 22
A.4 Survey Questionnaire 22
A.5 Interview Questions 22

B Detailed Results Tables **23**
B.1 Quantitative Results Tables 23
B.2 User Feedback Summary 23
B.3 Performance Metrics by Repository 23

C Code Snippets and Implementation Details **24**
C.1 Key Algorithm Implementations 24
C.2 API Endpoint Specifications 24
C.3 Database Schema SQL 24
C.4 Frontend Component Code 24
C.5 RAG Pipeline Code 24

D System Architecture Diagrams **25**
D.1 High-Level System Architecture 25
D.2 Data Flow Diagrams 25
D.3 Component Interaction Diagram 25
D.4 Deployment Architecture 25

E Additional Evaluation Data **26**
E.1 Raw User Study Data 26
E.2 Search Performance Analysis 26
E.3 Citation Accuracy Detailed Results 26

E.4 Latency Distributions 26

F Related Work Comparison Table **27**

F.1 Feature Comparison Matrix 27

F.2 Performance Comparison 27

List of Tables

List of Figures

2.1 *Control flow graph of a simple if-else statement.* 3
2.2 *Semantic code search in a shared embedding space for retrieval [3].* 5
2.3 *Exploiting LLM hallucinations through slopsquatting.* 6
2.4 *Retrieval-Augmented Generation to reduce hallucinations [7].* 7
2.5 *Generative UI for a Room Rug Visualiser [11].* 10

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 $P = 1$.

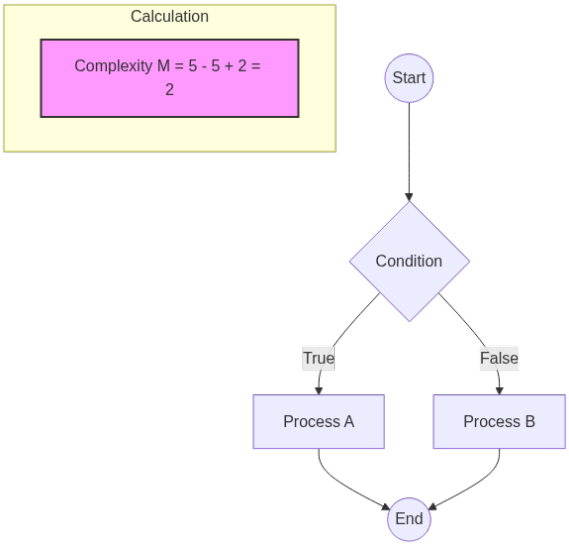

Figure 2.1: *Control flow graph of a simple if-else statement.*

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.

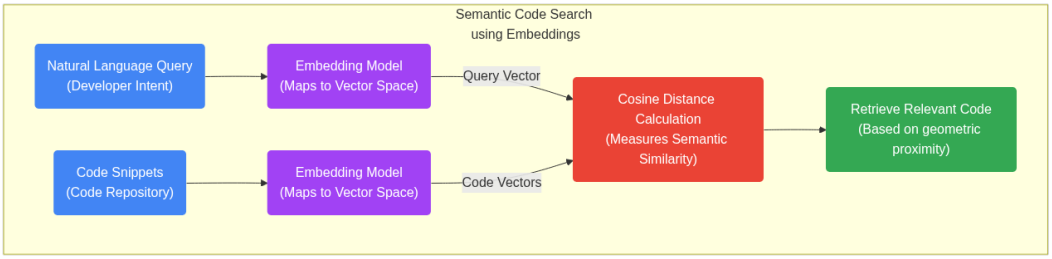

Figure 2.2: *Semantic code search in a shared embedding space for retrieval [3].*

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.

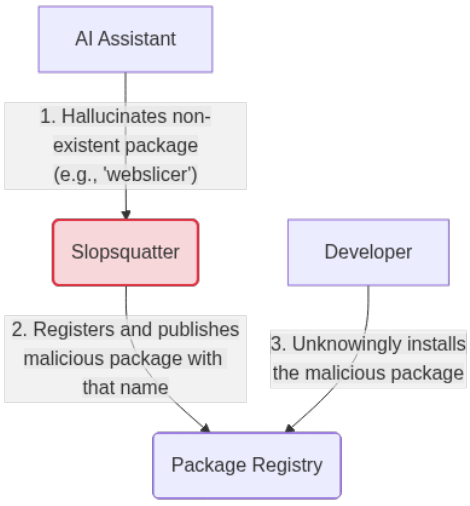

Figure 2.3: *Exploiting LLM hallucinations through slopsquatting.*

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.

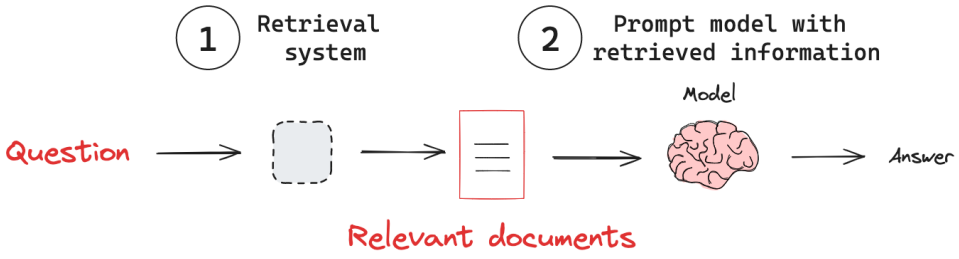

Figure 2.4: *Retrieval-Augmented Generation to reduce hallucinations [7].*

2.4 Code Visualisation and Dependency Analysis

2.4.1 Static Analysis and Dependency Graphs

To understand the real structure of a codebase, dependency graphs are essential. Using entities as nodes and relationships as edges, code graphs provide a visual representation

of how different components interact with each another [9]. Without this visualisation, it is inherently difficult for developers to grasp the complex relationships, preent in modern software architectures like microservices. In particular for CodeOrient, visualising dependencies helps developers trace data flows and control flows across modules which helps them understand unfamiliar codebases faster.

2.4.2 Interactive Visualisation Tools

React Flow is a popular open source library for building interactive node-based User Interfaces (UIs) in React applications [10]. This interactivity is crucial for developers to understand the linkage between nodes and edges and how they interact. Rather than traditional static diagrams, developers can explore relationships dynamically by hovering over nodes and its related code snippets.

2.5 Generative User Interface (UI)

2.5.1 Generative UI as a Paradigm

Google's recent research on Generative UI introduces a paradigm shift in interface design [11]. Traditionally, LLMs can only generate text-based outputs, but Generative UI extends this capability by generating dynamic user interfaces at runtime based on user prompts. This strengthens the capability of LLMs by giving them the tools to generate dynamic code graphs for visualisation, with their structure and content optimised for each specific query.

2.5.2 Task-Driven Data Models for Adaptive Interfaces

Recent research by Cao et al. showcase the use of task-driven data models as the foundation for Generative UI [12]. By allowing LLMs to generate a data model representing the core task, then mapping it to UI specifications, the resulting interfaces were more adaptive and aligned with user intent, compared to direct UI code generation. This also ensures that the generated UI is grounded in actual data structures rather than arbitrary code snippets.

2.6 Related Systems and Tools

Based on current semantic search tools like RAG [8] and interactive graph visualisation tools [10], there is a clear gap in integrating these capabilities into a unified developer onboarding experience. This addresses the onboarding challenges faced by developers when exploring unfamiliar codebases. The integration of LLMs, semantic search, code graph visualisation, and generative UI into a single application allows developers to search for relevant code snippets, understand their relationships via dynamically generated graphs, while ensuring trust through citation grounding [7].

2.7 Research Gaps and Motivation

While the individual domains of semantic search, graph analysis, and Generative UI have matured significantly, their integration into a developer tool reveals two critical research gaps that CodeOrient aims to address:

2.7.1 Bridging Semantic and Structural Code Search

Current tools typically favour either semantic search (finding code that looks right) or structural analysis (finding code that is connected). As noted by Limpanukorn et al. (2025), structural search outperforms semantic baselines, yet most AI tools like GitHub Copilot still rely primarily on text-based RAG. There is a lack of research into how Generative UI can bridge this gap by dynamically synthesising a visual graph that represents both the user's natural language intent and the codebase's physical architecture.

2.7.2 The "Black Box" of Generative UI in Software Engineering

Research by Leviathan et al. [11] and Cao et al. [12] establish the framework for task-driven UIs. However, these studies focus on general tasks such as education or shopping as shown below. In the high-stakes domain of software engineering, it is unknown how a constantly changing, generative interface affects a developer's productivity.

CodeOrient provides an experimental platform to observe whether generative graphs reduce cognitive load during onboarding or if the lack of visual consistency hinders comprehension.

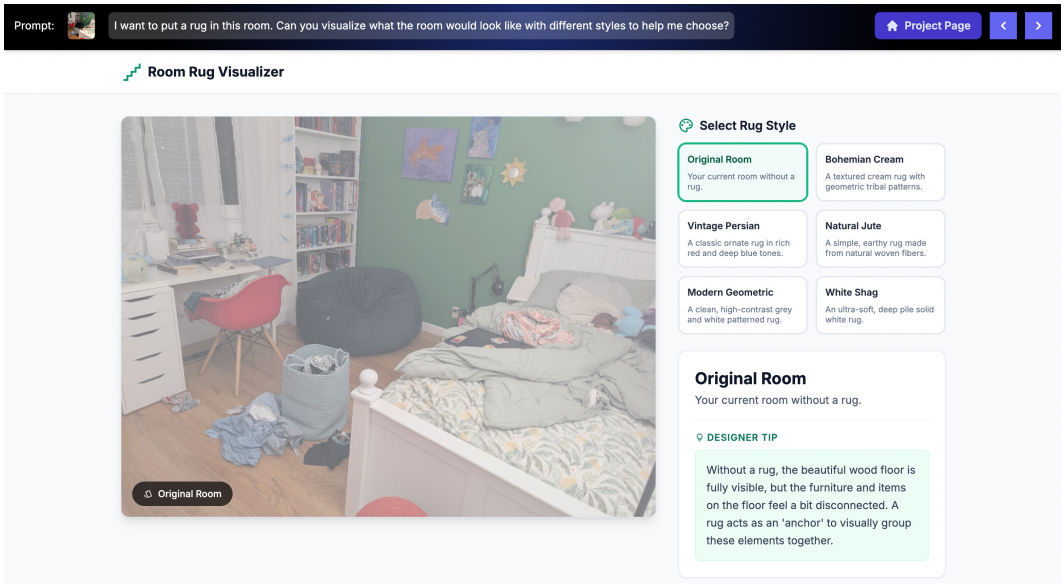

Figure 2.5: *Generative UI for a Room Rug Visualiser [11].*

2.8 Conclusion

Improving developer onboarding requires addressing code comprehension at multiple levels. Recent research in cyclomatic complexity [1], semantic code search [3], citation-grounded AI [7], and Generative UI [12] provides a comprehensive foundation. By integrating these approaches together, CodeOrient has the potential to accelerate developer onboarding while being reliabile and accurate.

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

D.1 High-Level System Architecture

D.2 Data Flow Diagrams

D.3 Component Interaction Diagram

D.4 Deployment Architecture

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.

[9] Jit. *How to Use a Dependency Graph to Analyze Dependencies*. 2025. URL: https://www.jit.io/resources/app-security/how-to-use-a-dependency-graph-to-analyze-dependencies.

[10] Nikola Jovanov et al. "A visual approach to project management using react flow". In: Jan. 2025, pp. 306–312. DOI: 10.5937/IIZS25306J.

[11] Y. Leviathan et al. *Generative UI: LLMs are Effective UI Generators*. 2025. URL: https://research.google/blog/generative-ui-a-rich-custom-visual-interactive-user-experience-for-any-prompt/.

[12] Yining Cao, Peiling Jiang, and Haijun Xia. *Generative and Malleable User Interfaces with Generative and Evolving Task-Driven Data Model*. 2025. arXiv: 2503.04084 [cs.HC]. URL: https://arxiv.org/abs/2503.04084.