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 vi

List of Figures vii

List of Listings viii

List of Algorithms ix

1 Introduction 1

1.1 Motivation and Problem Statement 1

1.2 Research Objectives and Goals 2

1.3 Key Contributions 2

1.4 Report Structure 3

2 Literature Review 4

2.1 Software Complexity Metrics and Code Quality Measurements 5

2.1.1 Cyclomatic Complexity as Foundation 5

2.1.2 Modern Complexity in Distributed Systems 6

2.2 Code Search and Natural Language Query Processing 6

2.2.1 Semantic Code Search Through Embeddings 6

2.2.2 Structural Code Search with Domain-Specific Languages 7

2.2.3 Retrieval using Code Embedding Models 7

2.3 AI Hallucination and Grounded Code Comprehension 8
2.3.1 The Hallucination Problem in LLMs 8
2.3.2 Citation-Grounded Code Comprehension 9
2.3.3 Retrieval-Augmented Generation 9
2.4 Code Visualisation and Dependency Analysis 9
2.4.1 Static Analysis and Dependency Graphs 9
2.4.2 Interactive Visualisation Tools 10
2.5 Generative User Interface (UI) 10
2.5.1 Adaptive and Dynamic Interfaces 10
2.6 Research Gaps and Motivation 10
2.6.1 Bridging Semantic and Structural Code Search 11
2.6.2 The "Black Box" of Generative UI in Software Engineering 11
2.7 Conclusion 12

3 System Design and Architecture **13**
3.1 System Overview 13
3.1.1 Core Components 13
3.1.2 Data Flow Architecture 14
3.1.3 Code Indexing Phase 15
3.1.4 Query Response Phase 15
3.1.5 Serverless Service Architecture 16
3.2 Frontend Architecture 17
3.2.1 Technology Stack 17
3.2.2 User Interface Design 18
3.2.3 Interactive Graph Visualisation with React Flow 19
3.2.4 Generative UI Card Components 19
3.3 Backend Architecture 20
3.3.1 API Design 20
3.3.2 Code Search Engine 20
3.3.3 LLM Integration and RAG Pipeline 21
3.4 Data Pipeline 22

3.4.1 Repository Ingestion 22
3.4.2 Chunking Strategy 22
3.4.3 Metadata Extraction 23
3.4.4 Storage of Code Chunks 23
3.5 Technical Stack Summary 24

4 Implementation Details **25**
4.1 Development Methodology 25
4.1.1 Iterative Development Process 25
4.1.2 Version Control and Branching Strategy 26
4.2 Core Implementation Components 27
4.2.1 Search Module 27
4.2.2 Code Graph Analysis 29
4.2.3 Generative UI System 31
4.2.4 Large Language Model Integration 33
4.2.5 RAG Pipeline Implementation 35
4.3 Database Schema 37
4.4 UI/UX Enhancements 40
4.4.1 Account Dashboard 40
4.4.2 Sharing of Conversations 40
4.4.3 Setting Preferences 41
4.5 Challenges and Solutions 42
4.5.1 Handling Large Codebases 42
4.5.2 Real-Time Graph Updates 42

List of Tables

3.1 Summary of Technical Stack and Rationale 24

List of Figures

2.1 *Control flow graph of a simple if-else statement.* 5
2.2 *Semantic code search in a shared embedding space for retrieval [3].* 7
2.3 *Exploiting LLM hallucinations through slopsquatting.* 8
2.4 *Retrieval-Augmented Generation to reduce hallucinations [7].* 9
2.5 *Generative UI for a Room Rug Visualiser [11].* 11

3.1 *Data Flow Architecture of CodeOrient* 14
3.2 *CodeOrient Chat Page Layout* 18

4.1 *Prototype of Dynamic Card Generation for Weather App* 26
4.2 *Git Branching Strategy for CodeOrient Development* 27
4.3 *Example of Repository Card in Generative UI* 32
4.4 *Example of Code Graph Card in Generative UI* 33
4.5 *Example of Multistep Planning by Search Architect Persona* 34
4.6 *Example of Gap Analysis by Gap Analyser Persona* 35
4.7 *Example of sources retrieved from RAG pipeline* 37
4.8 *Example of inline citations in LLM response* 37
4.9 *Example of Account Dashboard* 40
4.10 *Example of Sharing Conversation Link* 41
4.11 *Example of User Preferences Settings* 42

Listings

1 TypeScript interfaces for CodeOrient graph entities.312 Example of Assembled Context with Source Metadata.363 Prisma Database Schema.40

List of Algorithms

1 Language-Aware Recursive Code Splitting Algorithm 30

Chapter 1

Introduction

Software Engineering is undergoing a fundamental shift with the rise in Large Language Models (LLMs), Agentic Artificial Intelligence (AI), and generative user interfaces (GenUI). As the complexity of current software architectures (distributed systems) and codebases grow, the challenge of developers learning a new repository has become a significant bottleneck for engineering productivity. I propose CodeOrient, an autonomous AI-driven developer onboarding platform that leverages Retrieval-Augmented Generation (RAG) with dynamic graph visualisation. Deployed as an interactive onboarding assistant, CodeOrient combines the reasoning capabilities of LLMs with the structural insights of code graphs to transform how developers understand and navigate unfamiliar codebases.

1.1 Motivation and Problem Statement

The rapid advancement of AI-assisted coding tools, such as GitHub Copilot and Cursor, has prioritised code generation over code comprehension. As a new developer onboards, they often struggle with understanding the existing codebases due to their complex architectural nature. Three critical issues affecting effective onboarding are:

1. **LLM Hallucinations:** LLMs are trained on past data and often generate plasusible but factually incorrect information. This is particularly problematic as the generated response has no knowledge of the specific codebase being queried.

2. **Limited Context Window:** Large codebases often exceeds the input context window of LLMs, leading to incomplete or incorrect answers as the model cannot access all relevant information.

3. **Overloading of Information:** Most codebases are accompanied with static documentation which fails to capture the dynamic relationships within the code. Without visual context, developers are often overwhelmed by the volume of code and struggle to identify relevant components in a large codebase.

1.2 Research Objectives and Goals

The primary goal of this research is to design and implement CodeOrient, an AI-driven developer onboarding tool that addresses the challenges of code comprehension in complex codebases. The specific objectives are:

1. **Develop a Retrieval-Augmented Generation (RAG) Framework:** Implement a hybrid search mechanism that combines vector-based semantic search with traditional keyword-based search to retrieve relevant code snippets and documentation.

2. **Implement Code Visualisation with Generative UI:** Utilise React Flow to dynamically render interactive graphs based on user queries.

3. **Minimise AI Hallucinations:** Ensure every response or claim made by the LLM is backed by code citations from the retrieved documents.

4. **Evaluate the Effectiveness of Dynamic Visualisation:** Evaluate how dynamic graph visualisations affect the comprehension ability of developers compared to traditional text-based documentation.

1.3 Key Contributions

CodeOrient introduces three key contributions that separates it from existing developer onboarding tools:

- **Dynamic UIs:** Instead of traditional text-based responses generated by LLMs, Generative UI is utilised to create dynamic visualisations based on user queries. If a developer asks about ”authentication flow,” the UI creates a graph focused strictly on those related modules, rather than a cluttered, static diagram.
- **Citation-Grounded RAG Pipeline:** To reduce hallucination, CodeOrient integrates sources and inline citations mechanisms into the RAG framework, ensuring that all LLM responses can be verified against actual code segments.
- **Integration of Agentic LLMs:** CodeOrient utilises agentic LLMs equipped with tools. They can autonomously decide when to query the vector database, generate visualisations, or seek clarifications based on the conversation context.

1.4 Report Structure

This report is structured for the ideation to implementation journey of CodeOrient:

- **Chapter 2:** Literature Review explores software complexity metrics, semantic code search, and the emerging technologies of Generative UI.
- **Chapter 3:** System Design and Architecture details the technical stack of CodeOrient which features the RAG pipeline, code graph generation, and the interactive chat interface.
- **Chapter 4:** Implementation Details discusses the development methodology, key algorithms, and integration challenges encountered during the build process.
- **Chapter 5 & 6:** Evaluation and Results will analyse the tool's effectiveness to improve the onboarding experience through user studies and case studies on real-world codebases.
- **Chapter 7:** Discussion discusses the implications of the findings, limitations of the current approach, and potential directions for future research.
- **Chapter 8:** Conclusion summarises the contributions of this research and reflects on the transformative potential of AI-driven developer onboarding.

Chapter 2

Literature Review

New developers often struggle with understanding new and unfamiliar codebases, leading to prolonged onboarding times and reduced productivity. Various strategies have been proposed to address this challenge such as improved documentation practices. However, these methods often fall short in providing comprehensive and efficient solutions for code comprehension. With the increasing adoption of AI-assisted tools, new challenges have emerged. Often, these tools hallucinate information that is not grounded in actual source code. Furthermore, large codebases make it harder for AI to comprehend due to 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 [1]. As shown in Fig.1, a simple control flow graph of a function below yields a complexity of 2, where $P = 1$.

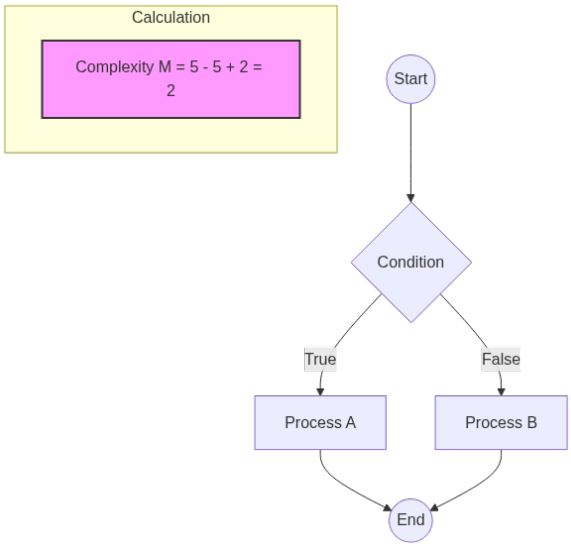

Figure 2.1: *Control flow graph of a simple if-else statement.*

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 showcases the potential of integrating graph visualisation into modern AI assistants to help developers understand not just function-level complexity but also system-level interactions.

2.2 Code Search and Natural Language Query Processing

2.2.1 Semantic Code Search Through Embeddings

Code search has evolved from simple keyword matching to more sophisticated semantic search techniques. Cambronero et al. explored the use of neural embeddings for semantic code search [3]. Their approach involves training models to transform both code snippets and natural language queries into a shared vector space [3]. Similar code snippets can be retrieved by calculating the soine similarity between their embedding vectors. The cosine similarity formula is given by: 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, and 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.

The figure below showcases how a query is transformed into an embedding vector which shares the same embedding space as code snippets. Although this technique does not replace traditional code search, it complements traditional methods that often

miss semantically relevant results.

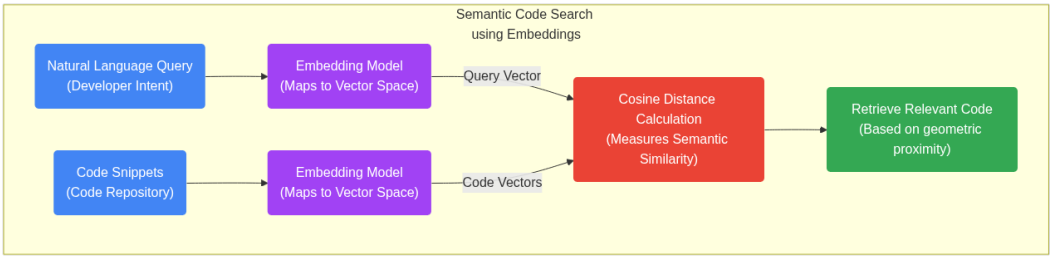

Figure 2.2: *Semantic code search in a shared embedding space for retrieval [3].*

2.2.2 Structural Code Search with Domain-Specific Languages

Recent research conducted in 2025 explored structural code search tecniques using Domain-Specific Languages (DSLs). Limpanukorn et al. proposed translating nautral language queries into DSL queries which capture the structural relationships within a codebase, which goes beyond mere semantic similarity [4]. Their approach 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. Their findings highlight the need for code graph visualisation to represent these structural relationships effectively.

2.2.3 Retrieval using Code Embedding Models

Qodo introduced specialised code embedding models (Qodo Embed-1) that achieved state-of-the-art performance (product score of 3.72/5) in Codebase Understanding Gartner® in 2025 [5]. Their approach bypasses the intermediate langauge description step. Instead, their models directly encode code semantics, resulting the models to be computationally efficient while maintaining high retrieval accuracy for code search tasks [5].

2.3 AI Hallucination and Grounded Code Comprehension

2.3.1 The Hallucination Problem in LLMs

With the incresed adoption of AI-assisted code generation tools like GitHub Copilot and ChatGPT, the risk of hallucination has become a critical concern. Hallucination in code generation refers to the generation of code that appears correct but is actually non-functional. In 2025, Spracklen et al. conducted a comprehensive study on hallucination in code-generating LLMs [6] and found that package hallucinations are a systemic issue across state-of-the-art code-generating models. Their research included analysing over 576,000 code samples generated by 16 different LLMs. Their findings revealed that LLMs consisitently hallucinate package names, and more worryingly, they regenerate the same false package name in 43% of repeated prompts [6].

In an agentic workflow where LLMs autonomously generate and execute code, hallucinations can be exploited by attackers via "slopsquatting", which is the practice of creating malicious packages with names similar to popular ones [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.

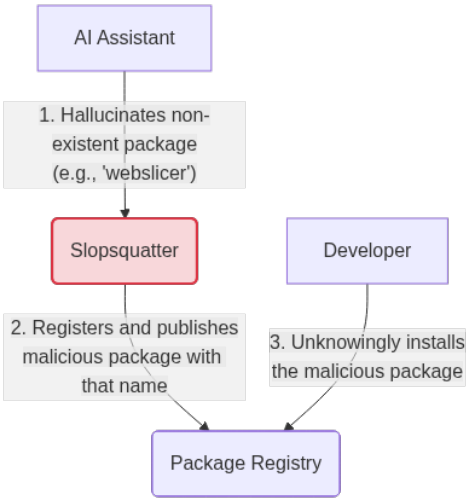

Figure 2.3: *Exploiting LLM hallucinations through slopsquatting.*

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 a context to generate grounded explanations with source citations.

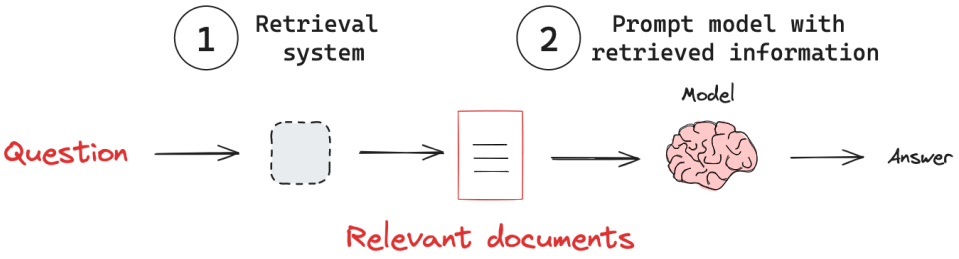

Figure 2.4: *Retrieval-Augmented Generation to reduce hallucinations [7].*

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 other [9]. The lack of visualisation makes it difficult for developers to grasp the complex relationship that are present in modern software architectures. This motivates CodeOrient's code graph visualisation feature to help developers trace data flows across multiple modules, functions, and files.

2.4.2 Interactive Visualisation Tools

Among many code visualisation libraries, React Flow stands out for its interactivity and ease of integration with React applications [10]. React Flow allows developers to explore the relationships between entities. Rather than traditional large static diagrams, developers can pan and hover over nodes to reveal additional information, which is cruicial for understanding complex codebases.

2.5 Generative User Interface (UI)

2.5.1 Adaptive and Dynamic Interfaces

Generative UI is an emerging field research by Google that focuses on using LLMs to generate dynamic UIs [11]. This extends the capability of text-based LLMs to generate UIs using task-driven data models [12]. Different queries will produce different UIs at runtime, allowing for highly personalised and adaptive interfaces. Their research shifts away from generating UI code snippets towards generating UI data models that is more aligned with user intent [12]. This approach directly motivates the use of Generative UI by rendering UI components instead of text-based responses in CodeOrient.

2.6 Research Gaps and Motivation

The reviewed literature highlights significant advancements in respective domains such as semantic search, graph analysis, and Generative UI. However, their integration into a cohesive system for developer onboarding remains underexplored. The following research gaps motivate the development of CodeOrient:

2.6.1 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 relies 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.

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

Research by Leviathan et al. [11] and Cao et al. [12] establishes 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 will serve as an experimental platform to explore whether Generative UI can effectively reduce cognitive load and accelerate code comprehension for developers.

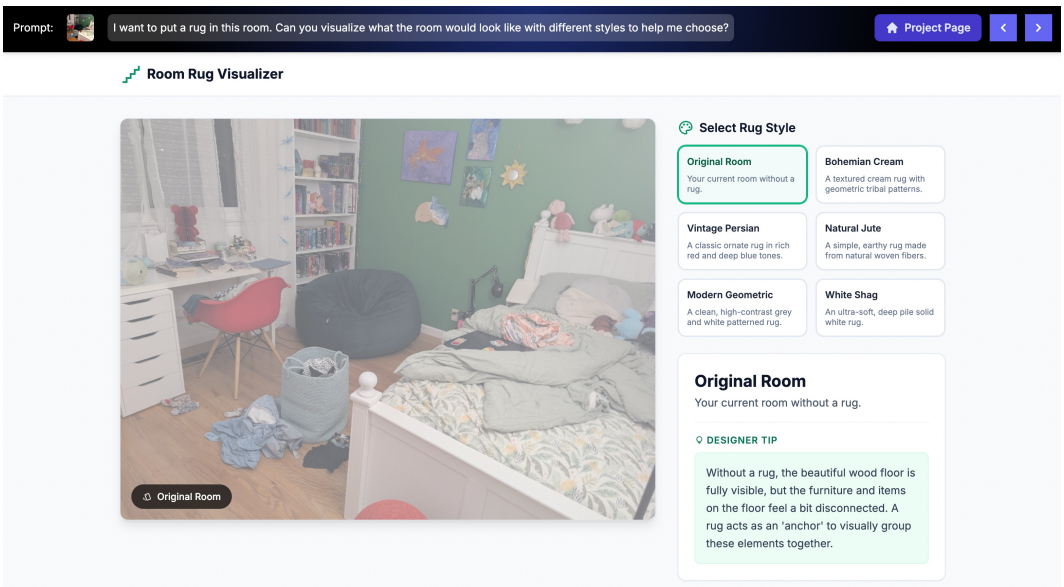

Figure 2.5: *Generative UI for a Room Rug Visualiser [11].*

2.7 Conclusion

To improve developer onboarding, it is essential to address code comprehension at different levels. By integrating insights from software complexity metrics [1], semantic code search [3], hallucination mitigation [7], and Generative UI [11], CodeOrient aims to accelerate developer onboarding while being reliable and grounded in actual source code.

Chapter 3

System Design and Architecture

3.1 System Overview

This chapter details the system architecture of CodeOrient. It will provide the high-level overview of the core components, data flow architecture, frontend and backend design, data pipeline, and technical stack summary.

3.1.1 Core Components

The design of CodeOrient is composed of four main components:

- **User Interface:** Built with Next.js and React Flow to provide an interactive experience between the user and AI assistant.
- **LLM Assistant:** Vercel AI SDK and OpenAI are utilised for natural language processing, tool calling, and response generation.
- **Code Search Engine:** A hybrid search engine that combines semantic search with keyword-based retrieval to find relevant code snippets which fallbacks to GitHub Search API when necessary.
- **Storage Solutions:** PostgreSQL is used for storing structured data, Upstash Redis for caching frequent queries, and Upstash Vector for storing vector embeddings of code chunks.

3.1.2 Data Flow Architecture

A multi-stage pipeline optimised for speed and quality of responses is built to handle user interactions. Figure 3.1 illustrates the data flow architecture, split into two phases: *Code Indexing Phase* and *Query Response Phase*.

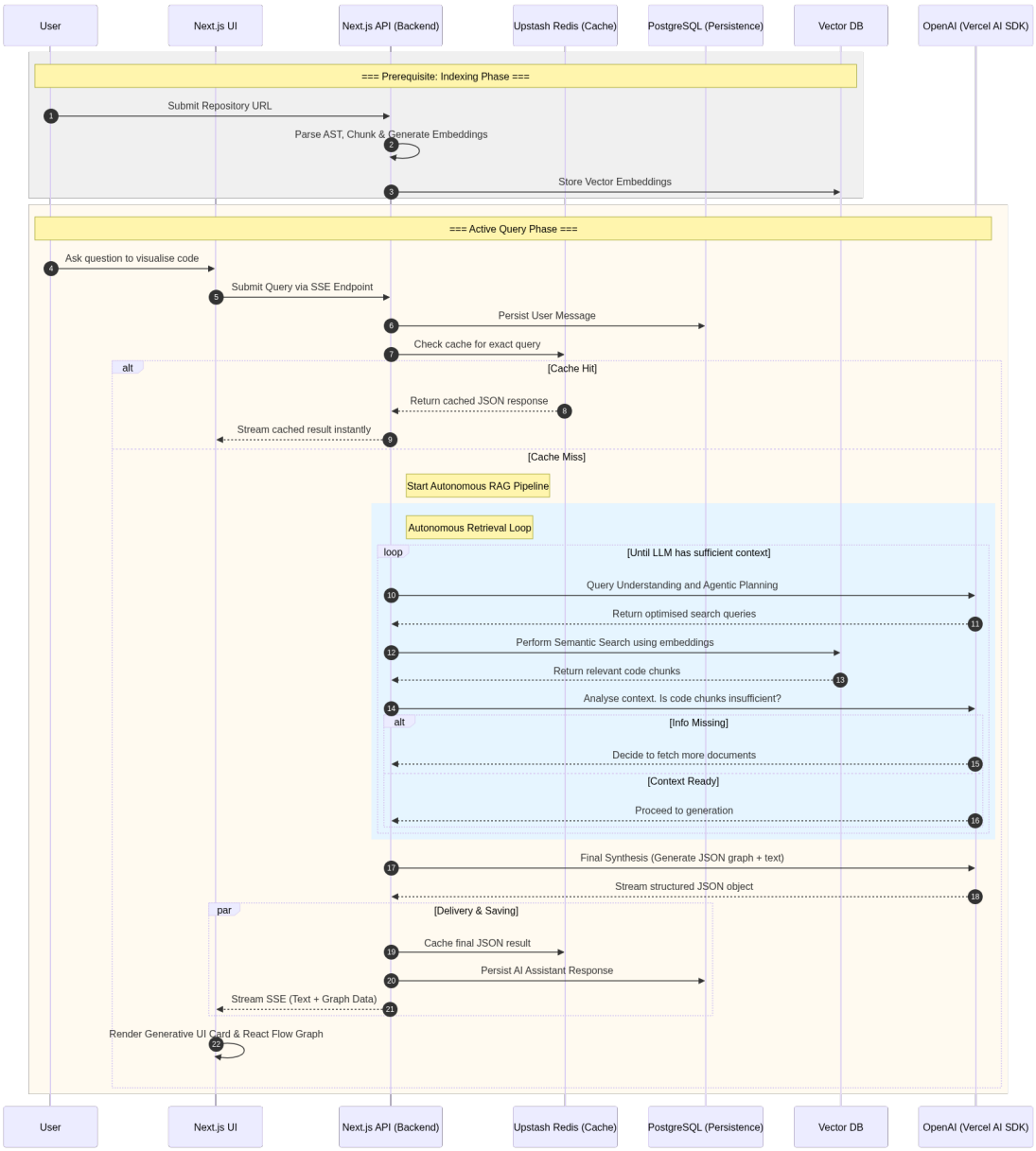

Figure 3.1: *Data Flow Architecture of CodeOrient*

3.1.3 Code Indexing Phase

Code indexing is a prerequisite step that ingests the required code repository for vector-based retrieval. This phase parses the source code files, chunks them, and generates vector embeddings to be stored in a vector database. To improve the recall, docstrings and comments are extracted as well. In cases where indexing is not performed, CodeOrient falls back to searching the codebase directly via GitHub Search API during query time.

3.1.4 Query Response Phase

This phase is triggered when a user submits a query. The steps involved are as follows:

1. **Query Submission:** The user submits a query via the frontend. The message is sent to the backend via API and saved to PostgreSQL to persist chat history.
2. **Cache Layer:** The backend will first check Upstash Redis for an identical query based on the current session ID. Two outcomes are possible:
    - **Cache Hit:** If a cached response exists, the stored JSON response is retrieved and streamed back to the frontend, cutting down latency by over 90%. Additionally, the use of a cache reduces overall LLM token consumption and cost.
    - **Cache Miss:** If no cached response is found, the system initiates the Retrieval-Augmented Generation (RAG) pipeline.
3. **Autonomous RAG Pipeline:** This pipeline represents the core reasoning engine of CodeOrient.
    - **Query Understanding and Planning:** The LLM first understands and rephrases the user's query. This step fixes any potential errors such as typos, ambiguous terms. If there is insufficient context, the LLM will prompt the user for more information instead. Next, the LLM generates a plan to decompose the query into sub-tasks, generating up to 3 sub-queries for retrieval.

- **Hybrid Retrieval and Ranking:** Each sub-query is converted into embeddings before retrieving the top-K relevant code chunks from the vector database. A ranking mechanism based on Distributed-Based Score Fusion (DBSF) is applied to combine results from both semantic and sparse search.
- **LLM Evaluation Loop:** This step utilises an autonomous and specialised LLM for context evaluation. If the context is deemed insufficient, the LLM will trigger additional searches in a loop until it gathers sufficient context to confidently answer the user query.

4. **Citation Grounded Response:** After gathering sufficient context, the LLM generates a structured JSON object that contains the required nodes, edges, code snippets (citations) and explanation required for visualisation. To mitigate hallucinations, the LLM is further prompted to perform inline citations for each code snippet used in the response.
5. **Message Persistence and Caching:** The final chat result is stored in PostgreSQL and also cached in Upstash Redis for future identical queries. Simultaneously, the generated response is streamed via Server-Sent-Events (SSE) back to the frontend for real-time rendering.
6. **Rendering Components:** Finally, the frontend parses the streamed JSON response to render as a Generative UI card, containing an interactive graph visualisation built with React Flow. Citation-backed explanations are streamed in after the graph is rendered.

3.1.5 Serverless Service Architecture

CodeOrient utilises Next.js API Route Handlers to create a modular, serverless backend. This separates logical concerns while maintaining the codebase as a monorepo for quicker development cycles. Additionally, when it is deployed in a serverless environment (Vercel), each Route Handler (e.g., `/api/search` vs `/api/index`) scales independently based on traffic. Even as a monorepo, the backend logic is divided into specialised services that operate independently:

- **Ingestion Service:** Handles the ingesting of code repositories from GitHub API asynchronously through background jobs, allowing the frontend to remain responsive.
- **Vector Orchestration Service:** Manages the communication with the vector database which handles embedding generation and semantic search.
- **RAG Agent:** Specialised agent that has access to various tools such as Code Search, Graph Generation, etc. It maintains the state of the search by deciding if the retrieved content is sufficient to answer the query or if more code snippets are required.
- **Code Search Service:** Interfaces with GitHub Search API to discover relevant code snippets based on user queries.
- **Persistence and Cache Service:** Prisma, a dedicated Object Relational Mapping (ORM) service is used to interact with PostgreSQL for persistence chat history and Redis client to interact with Upstash Redis for caching frequent queries.

3.2 Frontend Architecture

3.2.1 Technology Stack

- **Framework:** The framework of choice is Next.js App Router with Typescript. This framework allows to mix Server Components (RSC) and Client Components seamlessly, optimising for performance and developer experience.
  - **Server Components:** Used for static content that does not require interactivity, such as the landing page and documentation pages. This reduces the amount of JavaScript sent to the client, improving load times.
  - **Client Components:** Used for interactive elements such as the chat interface and graph visualisation. These components can leverage React hooks and state management libraries. To create a Client Component in Next.js, the file must include the directive `"use client"` at the top.

- **Styling:** Tailwind CSS & Shadcn UI are used for the application's design system, providing a consistent and responsive user interface.

3.2.2 User Interface Design

This section discusses the key design principles and layout of the chat page of CodeOrient application. Figure 3.2 showcases the chat page layout which features a split-pane design that prioritise the chat interface with the AI assistant and the source code.

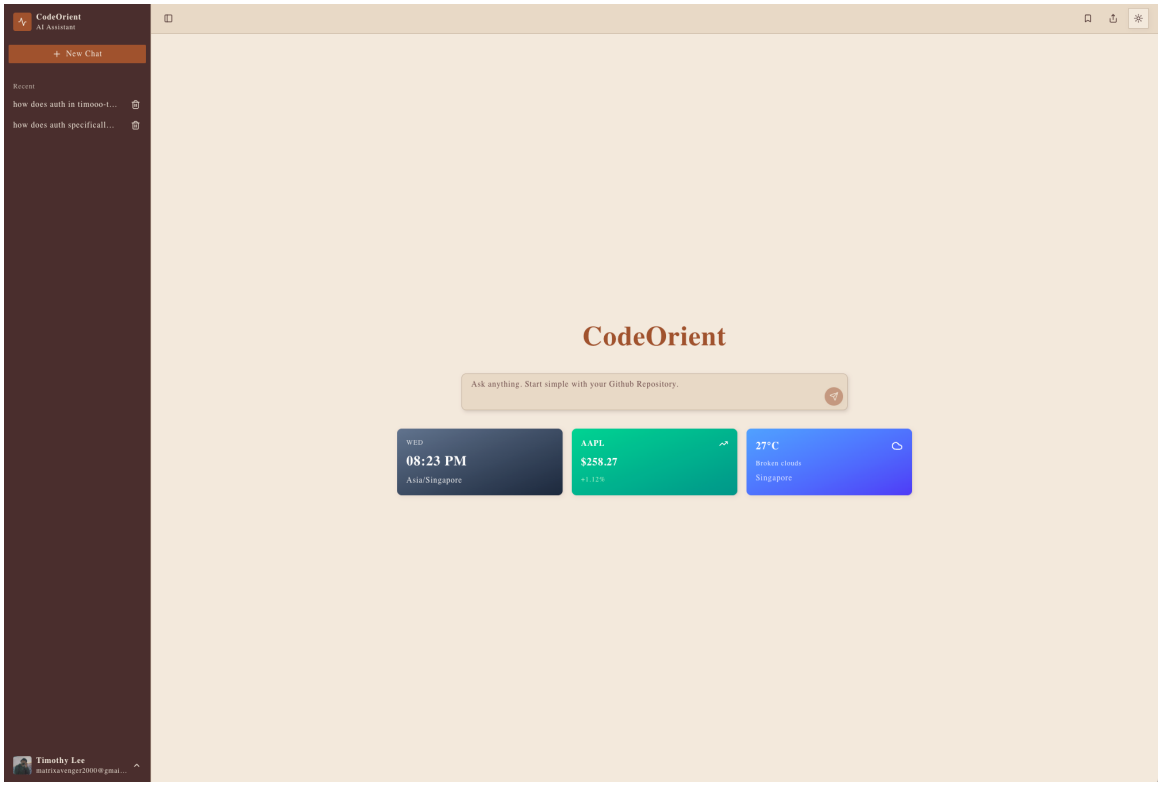

Figure 3.2: *CodeOrient Chat Page Layout*

The main components of the chat page layout include:

- **Persistent Sidebar:** A collapsible left sidebar is introduced to manage user context and provide quick access to conversation history in a chronological order. It also includes a user profile section at the bottom for account settings and logout.
- **Navigation Bar:** Sharing options, bookmarking and toggling of light/dark mode is included in the top navigation bar.

- **Main Chat Area:** This area is dedicated to the chat interface with the AI assistant.
- **Responsive Design:** The layout adapts to different screen sizes, ensuring usability across devices from mobile to desktop.

3.2.3 Interactive Graph Visualisation with React Flow

Static architecture diagrams are often high-level and extremely abstracted, making it difficult for users to scope in on a specific module. To address this, CodeOrient integrates LLMs and React Flow to visualise a subset of the code repository as an interactive graph. The core components of a graph visualisation include:

- **Nodes as Entities:** Nodes represent high-level code entities such as files, functions, classes, and components. Each node includes metadata about the entity type, file path, code snippet and description.
- **Edges as Relationships:** Dependencies are presented as directed edges between nodes, illustrating how data and logic flow between different parts of the codebase. Common relationship types include imports, calls, extends, and uses. For example, Component X importing Function Y would be represented as an edge from Node X to Node Y.
- **Interactive Features:** To improve code understanding and navigation within the codebase, users can interact with the graph by panning, zooming, or hovering over nodes to reveal additional information such as code snippets, code language, file path and description.

3.2.4 Generative UI Card Components

Standard LLM responses are limited to text or markdown outputs. Often, this is insufficient to convey complex code structures or relationships. To address this, CodeOrient adopted a Generative UI architecture where the LLM is able to generate structured JSON objects that renders as functional React components.

Based on the user's query, the LLM automonously decides which component to gen-

erate. For example, if the user asks about their list of repositories, the LLM would generate a "Repository Card" component. To simulate a smoother user experience, the component data is streamed into the UI, allowing the card to populate incrementally, thereby reducing perceived latency.

3.3 Backend Architecture

3.3.1 API Design

A RESTful API design pattern is adopted to create a clear separation of concerns between different backend services. Each endpoint is implemented as a serverless function, allowing each service to scale independently based on demand. The API is divided into four primary services:

- **Authentication API:** Manages user login, registration, and session management.
- **Ingestion API:** Manages the lifecycle of ingesting and indexing code repositories from GitHub API.
- **Search & Retrieval API:** Interfaces with the vector and structured databases to perform semantic search, retrieval of code snippets and chat persistence.
- **Streaming Chat API:** Orchestrates the Vercel AI SDK to handle LLM generation via SSE to the client.

3.3.2 Code Search Engine

Traditional keyword-based or fuzzy search methods fail to capture the intent behind a user's query, especially in a large and complex code repository. To overcome this problem, CodeOrient utilises a semantic Code Search Engine for vector-based retrieval of relevant code snippets.

1. **GitHub Octokit:** The engine uses GitHub's Octokit library to fetch private repository contents from the user's account. This allows the system to access up-to-date codebases for indexing.

2. **Hybrid Retrieval Strategy:** The engine utilises `bge-large-en-v1.5`, an embedding model to perform semantic search in Upstash Vector. Furthermore, it conducts a sparse search to find documents based on keyword frequency and term importance.

3. **Filter Mechanism:** To prevent cross-project contamination, the vector database is partitioned by repository and user ID. To further refine search results, the engine supports metadata filtering such as file type or entity type.

4. **Ranking Mechanism:** To balance between spase and dense retrievals, retrieved code snippets are ranked using Distributed-Based Score Fusion (DBSF).

3.3.3 LLM Integration and RAG Pipeline

Creating an autonomous RAG pipeline allows the LLM to move beyond simple search and retrieval patterns. Instead, the LLM follows a more complex reasoning process:

1. **Query Rephrasing:** The LLM first rephrases the user's query into multiple optimised sub-queries to improve recall during retrieval.

2. **Autonomous Reasoning Loop:** The pipeline implements a loop where the LLM will evaluate the initial search results. If a certain context is missing, the LLM will autonomously generate secondary searches before finalising the response. For example, if the LLM finds a function call but is missing the definition, the agent identifies the missing piece and triggers another search specifically for that function definition.

3. **Hallucination Mitigation:** To eliminate hallucinations, the LLM is prompted to only cite the relevant code snippets that are used to formulate the response. Each citation includes the file path, URL and code snippet for user verification. This ensures higher precision and trustworthiness of the generated content.

3.4 Data Pipeline

This section details the four stage data pipline that transforms code repositories into a structured knowledge base.

3.4.1 Repository Ingestion

The system uses an efficient streaming ingestion strategy to bypass GitHub API rate limits. The key optimisations include:

1. **Tarball Archive:** The system fetches the entire repository as a compressed `.tar.gz` archive in a single request using Octokit.
2. **In-memory Extraction:** The system utilises LangChain to extract code chunks from the archive without writing to disk, speeding up the ingestion process.
3. **Filtered Indexing:** A whitelist containing indexable file types (e.g., `.ts`, `.py`, `.go`) is used to filter out non-essential files (e.g., DockerFile) during ingestion.

3.4.2 Chunking Strategy

Instead of indexing one file as a single document, the engine splits each code file into smaller "logical" chunks while ensuring that each chunk maintains coherence during retrieval.

1. **Language Parsers:** The system uses `RecursiveCharacterTextSplitter` with its `.fromLanguage()` method to parse code files based on their programming langauge. The advantage of this approach is that it prioritises splitting at language specific boundaries (e.g., functions, classes) rather than arbitrary character limits.
2. **Optimised Chunk Size:** Each chunk is limited to a maximum of 1,500 tokens with a chunk overlap of 200 tokens. This overlap is key to maintain context across chunk boundaries, ensuring that related code segments are not lost during retrieval.

3.4.3 Metadata Extraction

To provide richer context during retrieval, each code chunk contains additional metadata fields:

1. **Docstring Extraction:** The parser scans for docstrings or comments preceding code entities. These high-level summaries provide semantic context during retrieval, improving the relevance of search results.
2. **Source Attribution:** To provide accurate citations during response generation, each chunk is tagged with its `filePath`, `repoFullName`, specific `startLine` and `endLine`.

3.4.4 Storage of Code Chunks

The final stage involves storing the processed code chunks for low-latency retrieval during query time.

1. **Indexing Status:** The system uses Prisma ORM and PostgereSQL to track the real-time progress of the indexing lifecycle of each repository to the user (e.g., `CLONING` → `PARSING` → `INDEXING` → `COMPLETED`). In the case of a network interruption, this serves as a checkpoint to resume indexing.
2. **Batch Upsert:** To optimise throughput, code chunks in batches of 100 are converted into embeddings using `bge-large-en-v1.5` model and upserted into Upstash Vector.

3.5 Technical Stack Summary

Category	Technology	Engineering Justification
Framework	Next.js 16	Enables a monorep setup for seamless integration of server and client components.
AI Framework	Vercel AI SDK	Enables real-time Generative UI rendering via SSE.
Database & ORM	PostgreSQL & Prisma	Manages the user, repository, and chat-session relational data while providing a type-safe query interface.
Vector Store	Upstash Vector	Provides a serverless vector database with metadata filtering to avoid cross-repository contamination.
Caching	Upstash Redis	Reduces LLM token consumption and decreases latency by over 90% by caching the same queries in the same chat session.
Visualisation	React Flow	Provides an interactive graph visualisation library to assist with code understanding.
Ingestion	Octokit (GitHub)	A library for secure fetching of code repositories from GitHub.
Analytics	Sentry	Provides real-time error and log monitoring, and performance tracking in different environments.

Table 3.1: Summary of Technical Stack and Rationale

Chapter 4

Implementation Details

4.1 Development Methodology

4.1.1 Iterative Development Process

CodeOrient was developed using an iterative approach to continuously refine and integrate feedback. The key stages of the development process are outlined below:

1. **Requirement Analysis:** The pain points of new developers navigating unfamiliar codebases were identified to gather initial requirements.
2. **Prototyping:** To validate core concepts, early prototypes of the search and Generative UI systems were built. For example, a simple weather card UI was created to test the Generative UI's capabilities is shown in Figure 4.1.

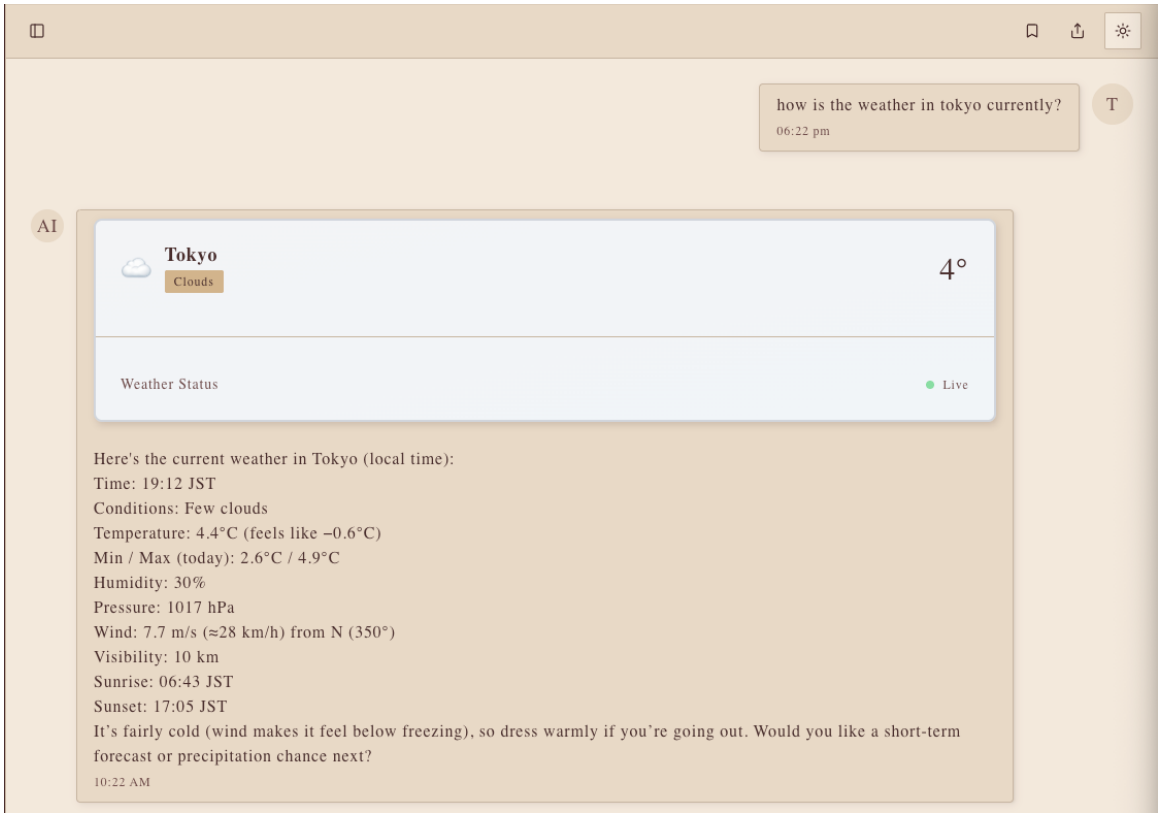

Figure 4.1: *Prototype of Dynamic Card Generation for Weather App*

3. **Incremental Development:** The search module, code graph analysis, and RAG pipeline were identified as priority features. Developement for these features was done incrementally to gather feedback early.

4. **Integrating User Feedback:** Feedback from user testing was regularly incorporated to improve the user experience of CodeOrient.

5. **Final Testing and Optimisation:** The system underwent rigorous testing and automated deployment via CI/CD pipelines to ensure performance and reliability of production release.

4.1.2 Version Control and Branching Strategy

This project utilised Git and GitHub for version control. Feature-branching was used to isolate the development of core features. This ensured that the main branch remained stable and protected for user testing. As for deployment, it was automated through CI/CD pipelines and was deployed to Vercel. Figure 4.2 illustrates the branching

strategy which squashed feature branches into the main branch after code reviews and testing.

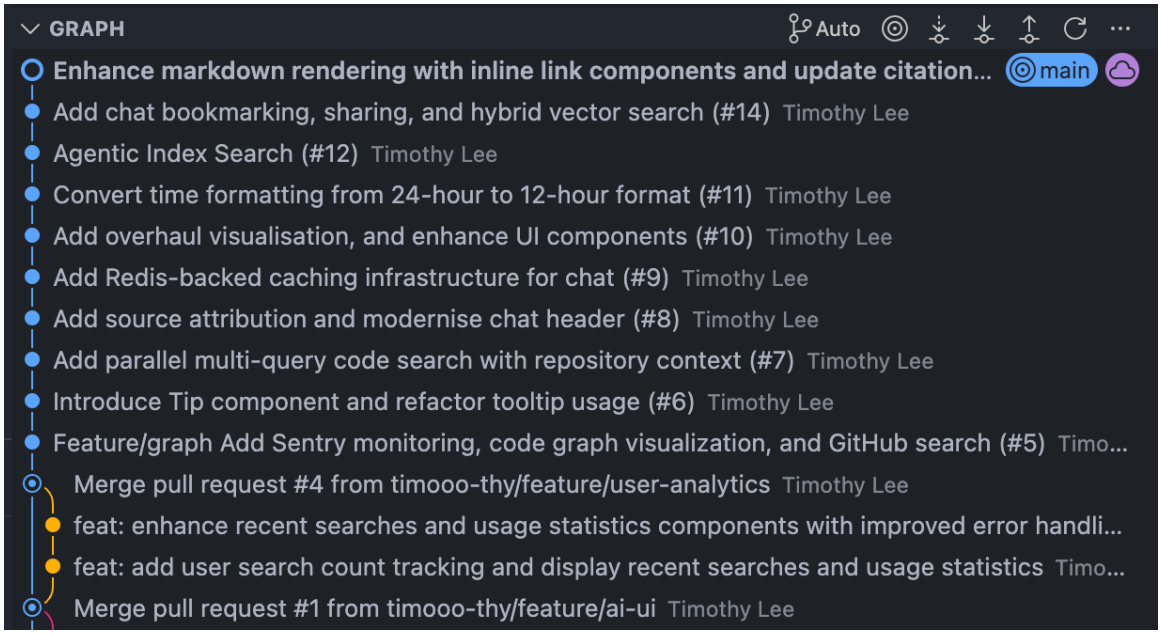

Figure 4.2: *Git Branching Strategy for CodeOrient Development*

4.2 Core Implementation Components

4.2.1 Search Module

BM25 Sparse Retrieval

To handle keyword-based searches, the system implements the Best Matching 25 (BM25) algorithm. The BM25 score for a document D given a query Q is computed as:

$$\text{BM25}(D,Q) = \sum_{q_i \in Q} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})}$$

where:

- $f(q_i, D)$ is the term frequency of query term q_i in document D.
- $|D|$ is the length of document D.
- $avgdl$ is the average document length in the corpus.

- k_1 and b are hyperparameters, typically set to $k_1 = 1.5$ and $b = 0.75$ for general text.
- $IDF(q_i)$ is the inverse document frequency of term q_i, which is calculated as:

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

  where N is the total number of documents and $n(q_i)$ is the number of documents containing term q_i.

This retrieval method is used by CodeOrient to retrieve code snippets that match the keywords in the user query. For example, if a user searchs for an exact function name, BM25 will prioritise documents containing that exact term.

Dense Embedding Retrieval

`bge-large-en-v1.5` embedding model is used for dense retrieval. Each chunk of code is converted to a 1024-dimensional vector and stored in Upstash Vector. This allows the engine to retrieve code snippets based on semantic similarity to the user query. The similarity between the query vector Q and document vector D is computed using cosine similarity:

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

Hybrid Search Integration

The final ranking is acheived through Distributed-Based Score Fusion. The normalised score is computed as:

$$Score = \frac{s - (\mu - 3\sigma)}{(\mu + 3\sigma) - (\mu - 3\sigma)}$$

where:

- s is the score.
- μ is the mean of the scores.

- σ is the standard deviation.
- $(\mu - 3\sigma)$ represents the minimum value (lower tail of the distribution).
- $(\mu + 3\sigma)$ represents the maximum value (upper tail of the distribution).

This approach takes into account the distribution of scores which is more sensitive to variation in score ranges from the different retrieval methods.

4.2.2 Code Graph Analysis

This section details the transformation of the code chunks into an interactive graph.

Dependency Extraction

Rather than using AST for traversal, the system utilise a recursive language aware splitting strategy. This approach splits the code into smaller chunks while maintaining the programming language's nuances, syntax and structure. This strategy is outlined in Algorithm 1.

Algorithm 1: Language-Aware Recursive Code Splitting Algorithm

Input: File Content C, File Extension E, ChunkSize S, Overlap O

Output: List of Semantic Chunks K

$K \leftarrow \emptyset$

$Language \leftarrow$ MapExtensionToLanguage(E)

if *Language is supported* **then**

    $Splitter \leftarrow$ InitialiseLangChainSplitter($Language, S, O$)

end

else

    $Separators \leftarrow$ {"\nclass ", "\ndef ", "\n\n", "\n", " "}

    $Splitter \leftarrow$ InitialiseRecursiveSplitter($Separators, S, O$)

end

$Documents \leftarrow Splitter$.createDocuments($C$)

foreach Doc *in* $Documents$ **do**

    $Chunk \leftarrow$ ExtractContentAndMetadata(Doc)

    Add $Chunk$ to K

end

return K

Graph Construction Algorithm

As React Flow requires a structured object to render the graph, the extracted entities and their relationships are mapped to a JSON object. In Listing 1, the code entities are represented as nodes and their relationships as edges.

```typescript
/**
 * Schema for Code Graph Nodes
 */
export type CodeGraphNode = {
  id: string; // Unique identifier: userId::repo::path::type::name
  label: string; // The display name of the entity
  type?: "file" | "function" | "class" | "component";
  filePath?: string; // Original source file path
  codeSnippet?: string; // The raw source code associated with the entity
  description?: string; // Extracted docstring or JSDoc comment
};

/**
 * Schema for Code Graph Edges
 */
export type CodeGraphEdge = {
  id: string; // Composite ID: sourceID->targetID
  source: string; // ID of the originating node
  target: string; // ID of the destination node
  label?: string; // Relationship type
  type?: "imports" | "calls" | "extends" | "uses";
  animated?: boolean; // Visual indicator for logic flow
};
```

Listing 1: TypeScript interfaces for CodeOrient graph entities.

4.2.3 Generative UI System

The Generative UI system dynamically creates interactive UI cards based on user queries. The implementation involves several key components:

Toolkit Selection

External tools are integrated to enhance the LLM's capabilities. The available tools for selection are:

- **Code Graph Tool:** Retrieves and visualises code entities and their relationships.

- **Repository Search Tool:** Fetches all repositories associated with the user.
- **GitHub Search Tool:** Fetches specific files or code snippets from GitHub directly.
- **Vector Search Tool:** Interacts with the hybrid search module to fetch relevant code snippets.

Dynamic Card Generation

Based on the user query, the LLM intelligently selects the appropriate tool(s) to fulfill the request. It then generates a structured JSON object based on the tool(s) chosen which results in different card visualisations. It is dynamically streamed to the frontend for real-time rendering. The different visualisations supported are:

- **Repository Card**

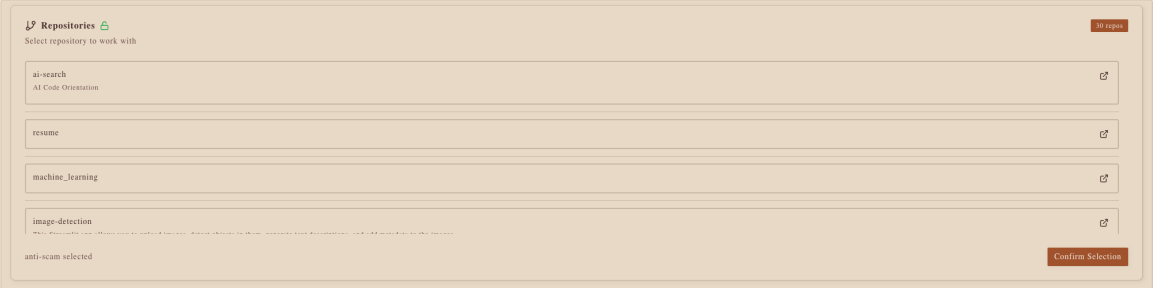

Figure 4.3: *Example of Repository Card in Generative UI*

- **Code Graph Card**

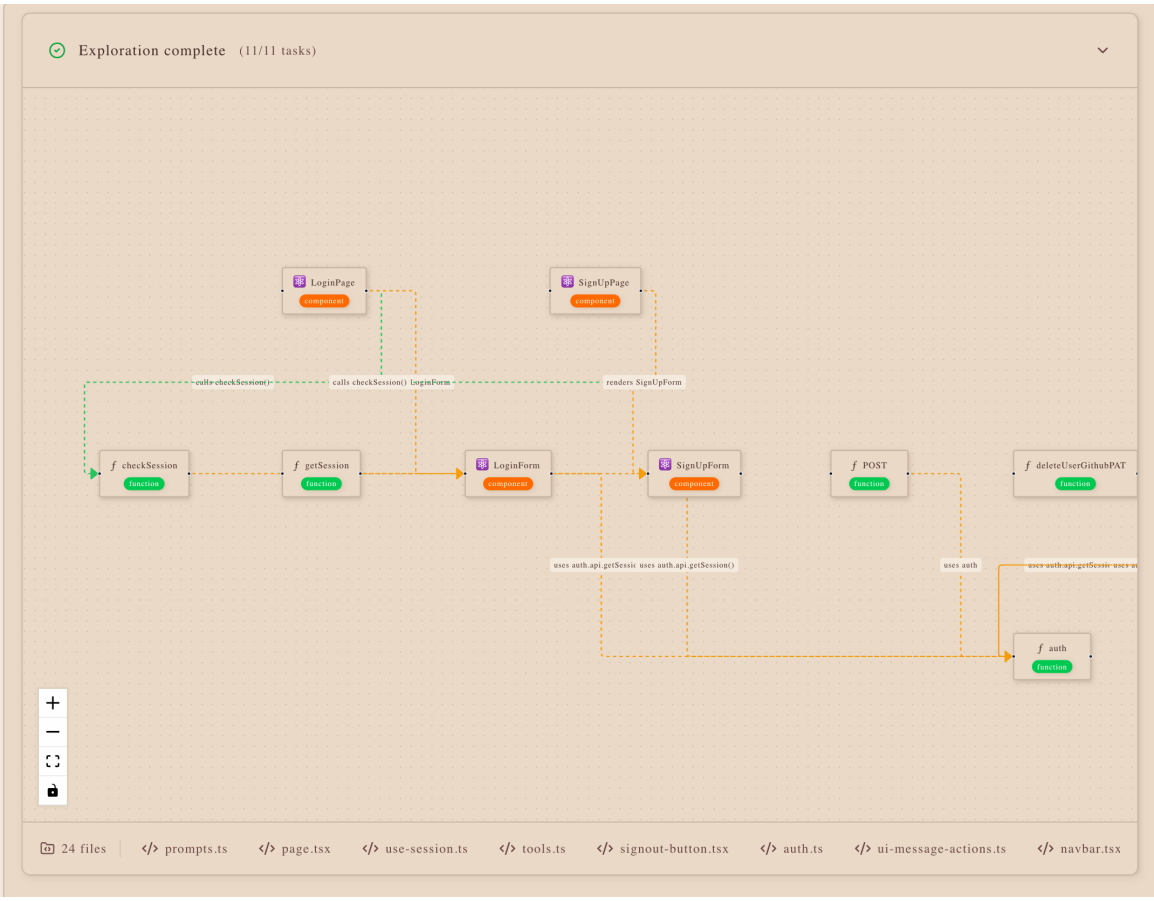

Figure 4.4: *Example of Code Graph Card in Generative UI*

4.2.4 Large Language Model Integration

CodeOrient is model agnostic and can integrate with any LLM providers that support tool-calling.

Model Selection and Justification

In this implementation, CodeOrient utilises OpenAI's GPT-4.1-mini model due to its advanced reasoning capabilities and large context window, which prevents "lost in the middle" degradation. Furthermore, the latency and cost-effectiveness of the mini variant make it suitable for real-time applications compared to State-Of-The-Art reasoning models.

Prompt Engineering Strategies

To optimise the performance of the LLM and reduce hallucinations during code exploration, a Multi-layered Prompting Strategy is employed with specialised personas for different stages of the RAG pipeline:

- **Search Architect Persona:** This persona's responsibility is to decompose a user's query into a structured search plan. The plan entails the steps it will take before generating a graph. The most important responsibility is to break down complex queries into non-overlapping sub-queries that target different parts of the codebase. Additionally, the repository's tree is provided as context to guide the planning process. This reduces the hallucination of non-existent files or functions. In Figure 4.5, the step by step breakdown of the planning process is illustrated to the user.

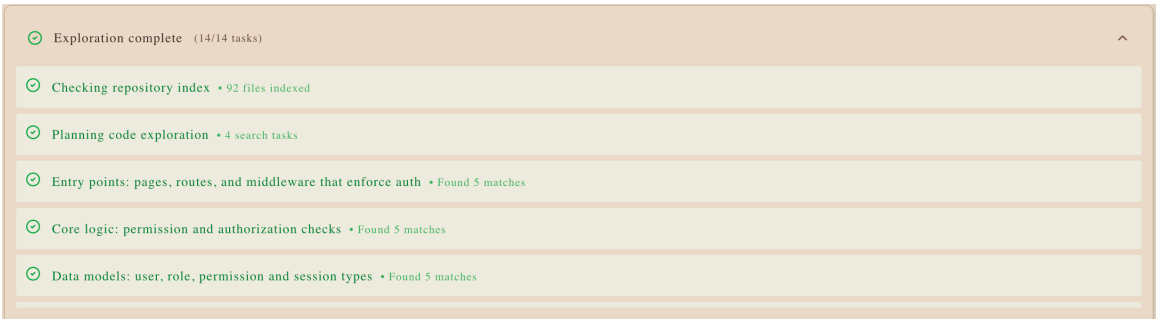

Figure 4.5: *Example of Multistep Planning by Search Architect Persona*

- **Gap Analyser Persona:** To give the LLM autonomy in exploring the codebase, this persona acts as a quality control layer to identify gaps in the retreived context. An additional search iteration with refined queries will be triggered if the context is deemed insufficient. This iterative process continues until the LLM can answer the user's query accurately. Figure 4.6 illustrates an example of the LLM identifying a need for additional context.

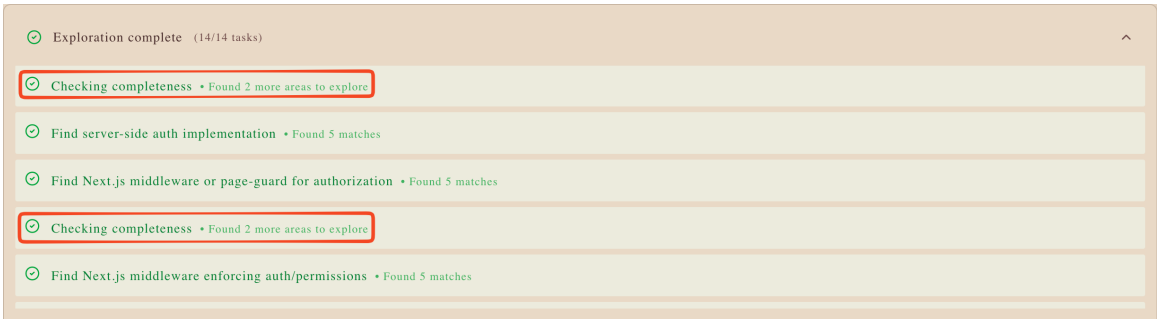

Figure 4.6: *Example of Gap Analysis by Gap Analyser Persona*

- **Graph Architect Persona:** As React Flow requires a structured object to render the graph, this persona translates the retrieved context into a graph object. This is achieved by identifying the relevant entities and their relationships, before formatting them into nodes and edges.

4.2.5 RAG Pipeline Implementation

This pipeline iteratively retrieves relevant code snippets to ground the LLM's responses. The key components are outlined below:

Query Processing

User queries are usually ambiguous and may include typos. To address this, the pipeline first rephrases the query into technical search queries that align with the codebase's terminology and structure. This is achieved using the Search Architect persona described earlier.

Multistep Planning & Exploration

Mentioned previously, the Search Architect persona decomposes complex queries and conducts a breadth-first exploration.

Retrieval & Ranking

The refined queries are used to fetch relevant code chunks from the hybrid search module, with `userId` and `repoFullName` used as filters to ensure strict multi-tenancy.

K-Nearest Neighbour with $K = 10$ is used to retrieve the most similar chunks based on cosine similarity. The retrieved chunks are then ranked using the Distributed-Based Score Fusion method to ensure the most relevant snippets are prioritised.

Context Assembly

The top-ranked code chunks are assembled and wrapped in XML-style tags containing metadata for the LLM to accurately reference the sources during the response phase. An example of the assembled context is shown in Listing 2.

```
<chunk file="lib/auth.ts" lines="12-45" url="...">
  [Code Snippet]
</chunk>
```

Listing 2: Example of Assembled Context with Source Metadata.

Citation Extraction and Grounding

To increase the credibility of the generated answer, all sources retrieved from the RAG pipeline are shown to the user in Figure 4.7. To further reduce hallucination in the response, the LLM is prompted to provide inline citations using markdown format `[file_path](link_to_source_code)`. The frontend parses these citations to create clickable links that direct users to the exact source code locations. An example of inline citations is shown in Figure 4.8.

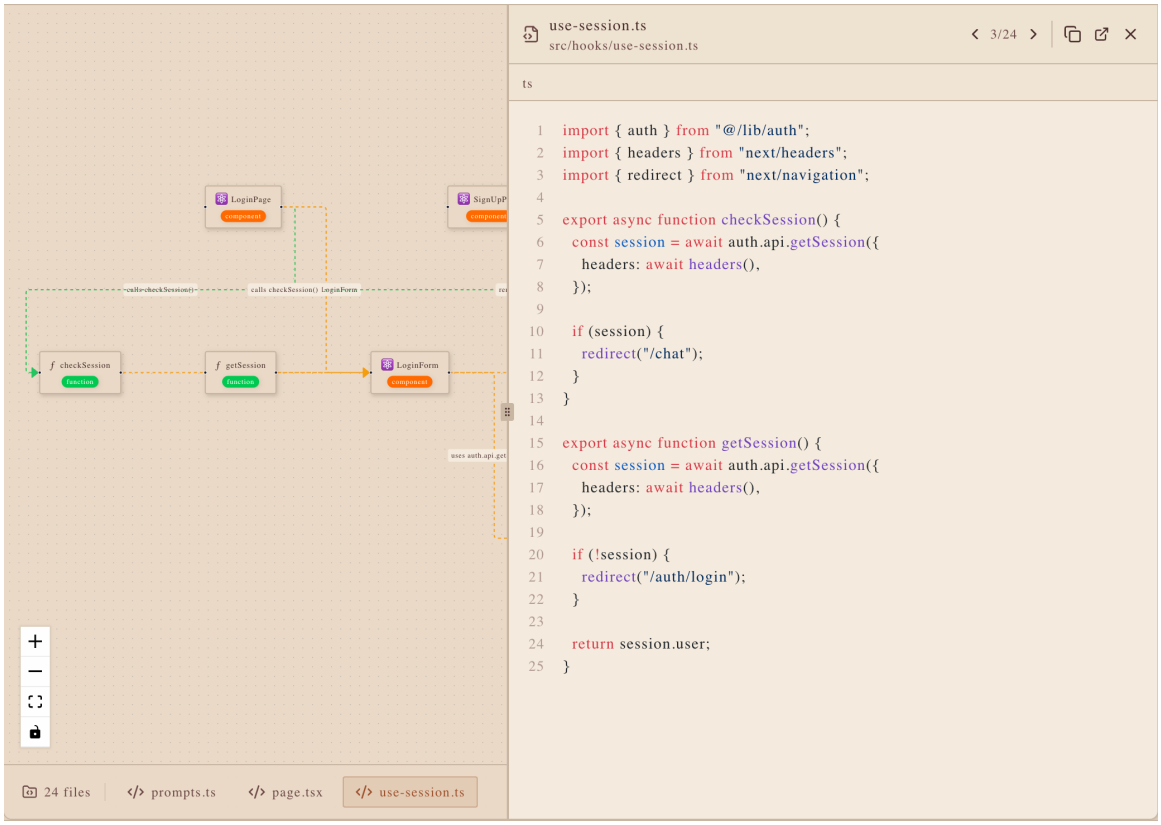

Figure 4.7: *Example of sources retrieved from RAG pipeline*

In the "timooo-thy/ai-search" repo, authentication is implemented primarily using the better-auth library with Prisma as the database adapter.
Core Auth Setup:
In src/lib/auth.ts, an auth instance is created using betterAuth(). This configures authentication to use Prisma as the adapter, tying auth data to the Prisma-managed database.
Client-side Auth:
The client-side uses an authClient created with createAuthClient() from better-auth's React integration in src/lib/auth-client.ts. This authClient handles functions like signing in and signing out from the UI.
Session Management:
The checkSession() and getSession() functions in src/hooks/use-session.ts interact with the auth.api.getSession() to verify if a user session is active. checkSession() redirects to login if no session exists, while getSession() redirects to the chat if a session is found.
Authentication Pages:
The login page (src/app/auth/login/page.tsx) checks the session and then renders a login form component.
Similarly, the signup page (src/app/auth/signup/page.tsx) checks the session and renders a signup form.
API Route:
Authentication requests (like sign-in, sign-out, session retrieval) are handled by API route handlers defined in src/app/api/auth/[...all]/route.ts, which delegates to the better-auth instance.
Sign Out:
There is a sign-out button component (src/app/chat/components/signout-button.tsx) that signs the user out via the authClient and redirects to login.

Figure 4.8: *Example of inline citations in LLM response*

4.3 Database Schema

CodeOrient utilises PostgreSQL for relational data storage, managed via Prisma ORM. The schema is organised into three primary clusters:

- **User Identity & Session:** Tables to manage user authentication and GitHub Personal Access Tokens (PATs).
- **Conversation State:** Tables storing `Chat`, `Message`, and `Part` models to support Generative UI and tool-calling outputs.
- **Indexing Lifecycle:** Table to track the asynchronous indexing jobs for user repositories, used in the RAG pipeline.

The complete Prisma schema is provided in Listing 3.

```
// Core User and Repository Models
model User {
  id            String    @id
  name          String
  email         String    @unique
  githubPAT     String?   // Encrypted token for repository access
  chats         Chat[]
  createdAt     DateTime  @default(now())
  @@map("user")
}

model IndexedRepository {
  id            String         @id @default(cuid())
  userId        String
  repoFullName  String         // Format: "owner/repo"
  status        IndexingStatus @default(PENDING)
  progress      Int            @default(0)
  totalFiles    Int            @default(0)
  indexedFiles  Int            @default(0)
  lastIndexedAt DateTime?

  @@unique([userId, repoFullName])
  @@map("indexed_repository")
}

// Conversational State with Generative UI Support
model Chat {
```

```
28    id        String    @id @default(cuid())
29    title     String
30    messages  Message[]
31    userId    String?
32    User      User?     @relation(fields: [userId], references: [id])
33    @@map("chat")
34  }
35
36  model Message {
37    id        String      @id @default(cuid())
38    chatId    String
39    chat      Chat        @relation(fields: [chatId], references: [id],
      ↪  onDelete: Cascade)
40    role      MessageRole
41    parts     Part[]      // Supports multi-modal and tool-call outputs
42    @@map("message")
43  }
44
45  model Part {
46    id        String          @id @default(cuid())
47    messageId String
48    message   Message         @relation(fields: [messageId], references: [id],
      ↪  onDelete: Cascade)
49    type      MessagePartType
50
51    // Generative UI and Tool Metadata
52    tool_toolCallId           String?
53    tool_visualiseCodeGraph_output Json? // Stores React Flow graph data
54    data_codeGraph            Json?
55
56    @@map("part")
57  }
58
59  enum IndexingStatus {
60    PENDING
61    CLONING
62    PARSING
63    INDEXING
```

```
64    COMPLETED
65    FAILED
66  }
```

Listing 3: Prisma Database Schema.

4.4 UI/UX Enhancements

This section highlights the various UI/UX features implemented to enhance user experience in CodeOrient.

4.4.1 Account Dashboard

The account dashboard allows users to monitor their interaction with the platform. As shown in Figure 4.9, users can view statistics such as:

- Recent Activities
- Usage statistics (e.g., lifetime searches, total repositories analysed)

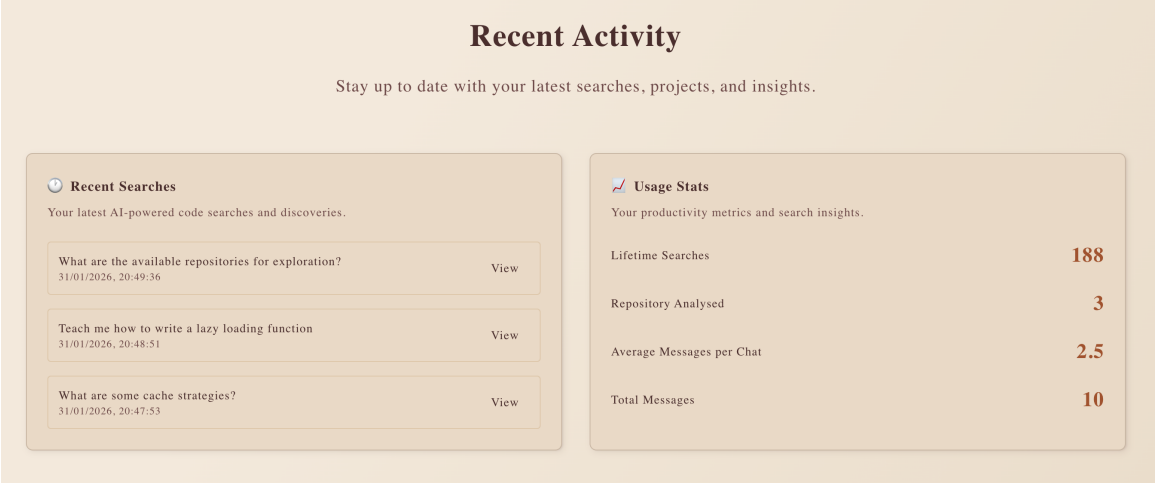

Figure 4.9: *Example of Account Dashboard*

4.4.2 Sharing of Conversations

To facilitate collaboration, CodeOrient supports the sharing of conversations via unique read-only links. Figure 4.10 illustrates the shared chat view where authorised users can

view the conversation history in a read-only format.

Figure 4.10: *Example of Sharing Conversation Link*

4.4.3 Setting Preferences

This interface allows users to customise their experience. As illustrated in Figure 4.11, this tab of the settings page focuses on data integration. Users can securely connect their GitHub accounts via Personal Access Tokens (PATs) to enable repository indexing and analysis. Furthermore, users can index any private repositories they have access to. The entire indexing process is asynchronous, and real-time progress updates are provided. By enabling indexing, the LLM will prioritise searching the vector database over GitHub to reduce latency and cost.

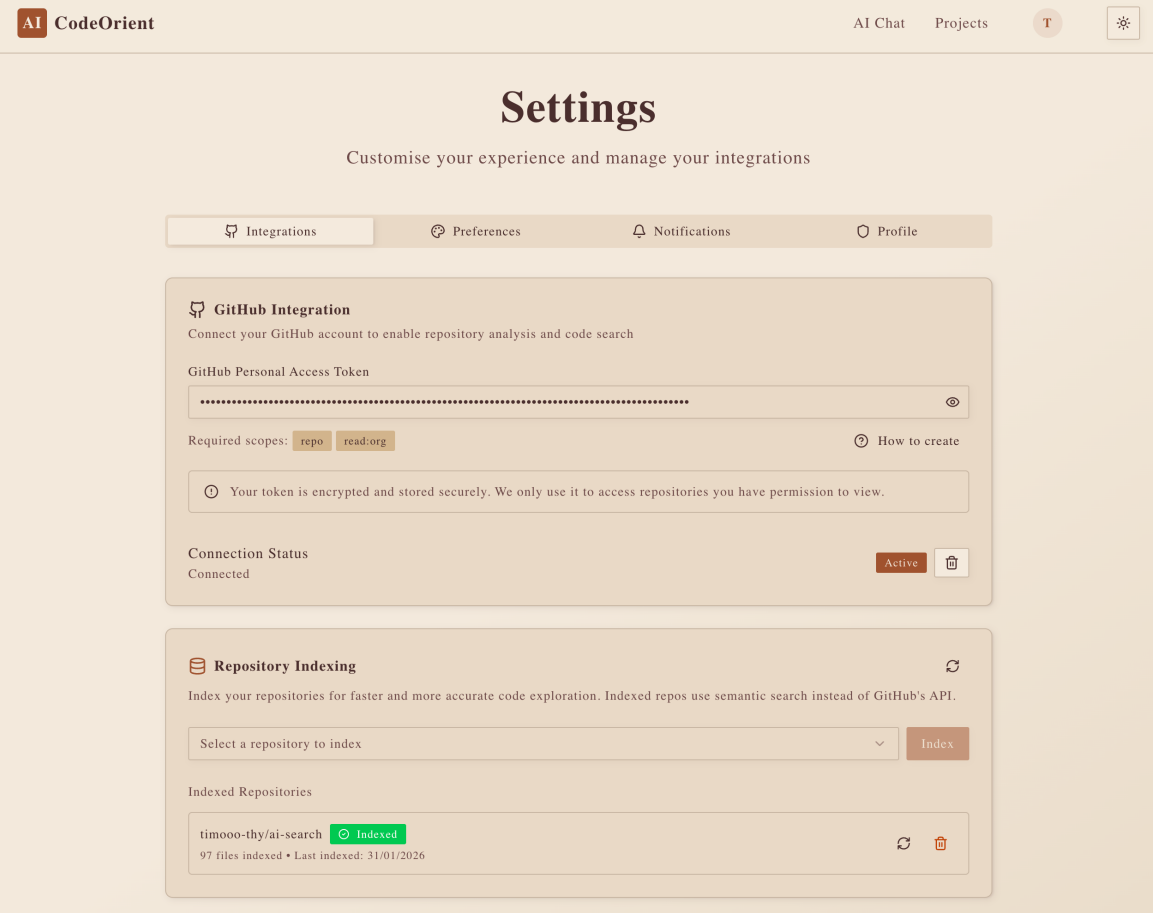

Figure 4.11: *Example of User Preferences Settings*

4.5 Challenges and Solutions

4.5.1 Handling Large Codebases

To overcome GitHub API rate limits, the system implements a **Streaming Tarball Strategy**. By fetching a single compressed `.tar.gz` archive, the system reduces network calls by over 90% and processes repository data entirely in-memory.

4.5.2 Real-Time Graph Updates

The system utilises Server-Sent Events (SSE) to stream the graph data for a smoother user experience. This allows React Flow to render an empty canvas initially and progressively add nodes and edges after the search has completed.

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.