NANYANG TECHNOLOGICAL UNIVERSITY

# NEXT-GEN GENERATIVE SEARCH ENGINE FOR CODE BASE

Timothy Lee Hongyi

College of Computing and Data Science

2026

# NANYANG TECHNOLOGICAL UNIVERSITY

**SC4079**

# NEXT-GEN GENERATIVE SEARCH ENGINE FOR CODE BASE

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

by

Timothy Lee Hongyi

College of Computing and Data Science
2026

# Abstract

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

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

# Acknowledgments

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

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

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. However, for a developer joining a new organisation, the primary hurdle is not writing new code, but understanding the existing codebases tied within complex architectures. Current onboarding processes face three critical issues:

1. **The Hallucination Problem:** LLMs often generate plausible-sounding but incorrect explanations of code structure, leading to confusion and mistrust.

2. **The Context Window Constraint:** Large codebases exceed the input limits of LLMs, resulting in fragmented or incomplete answers.

3. **Cognitive Overload in Navigation:** Static documentation fails to capture the dynamic relationships within code, forcing developers to mentally map text-based

## 1.2 Research Objectives and Goals

The primary goal of this research is to create a citation-grounded, visually adaptive system that reduces the time required for a new developer to familiarise with the codebases. The specific objectives are:

1. **Develop a Retrieval-Augmented Generation (RAG) Framework:** Moving beyond keyword/fuzzy search to a hybrid system that queries semantic embeddings to capture user intent.

2. **Implement a Generative UI for Code Visualisation:** Leveraging libraries like React Flow to dynamically render interactive graphs that adapt to the user's specific natural language intent.

3. **Minimise AI Hallucinations through Citation Grounding:** Ensuring every architectural claim made by the LLM is backed by code snippets or graph nodes from the actual repository.

4. **Evaluate the Impact of Visual Context on Onboarding:** To explore how dynamic graph visualisations affect developer comprehension and cognitive load compared to traditional text-based documentation.

## 1.3 Key Contributions

CodeOrient introduces several innovations that distinguish it from conventional AI coding assistants:

- **Dynamic Graph Synthesis:** Instead of text-based answers, Generative UI is used to create custom visualisations for each query. 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:** By integrating citation mechanisms into the RAG framework, CodeOrient ensures that all LLM outputs are verifiable against actual code segments, significantly reducing misinformation.
- **Agentic LLM Integration:** CodeOrient utilises agentic LLMs capable of autonomously deciding when to query the vector database, generate visualisations, or seek clarifications. This autonomy allows for a more fluid interaction between the developer and the system.

## 1.4 Report Structure

This report is organised to guide the reader through the theoretical foundations and technical implementation of CodeOrient:

- **Chapter 2:** Literature Review explores the intersection of software complexity metrics, semantic code search, and the emerging technologies of Generative UI.
- **Chapter 3:** System Design and Architecture details the technical stack, including the integration of Agentic LLMs with vector databases and React Flow-based frontend.
- **Chapter 4:** Implementation Details discusses the challenges of grounding AI outputs in the source code and the development of the RAG pipeline.
- **Chapter 5 & 6:** Evaluation and Results will analyse the tool’s effectiveness in reducing cognitive load and improving search precision.
- **Chapter 7:** Discussion discusses the implications of the findings, limitations of the current approach, and potential avenues for future research.
- **Chapter 8:** Conclusion summarises the contributions of this research and reflects on the broader impact of AI-driven developer onboarding tools.

# 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 remains 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 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 $P = 1$.

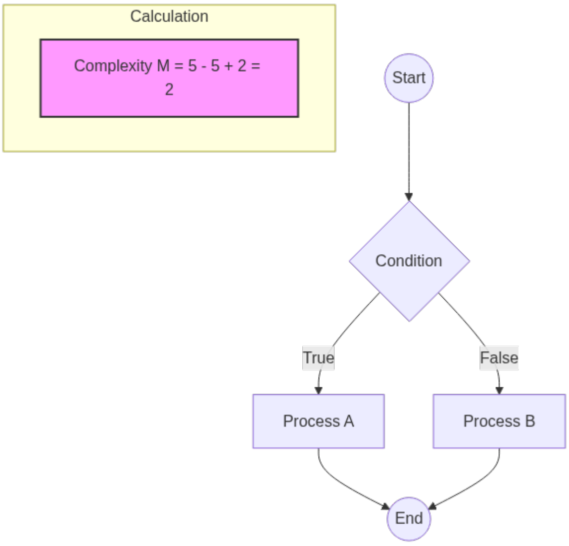


Figure 2.1: *Control flow graph of a simple if-else statement.*

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 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, 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.

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 searches, 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 repositories like GitHub.

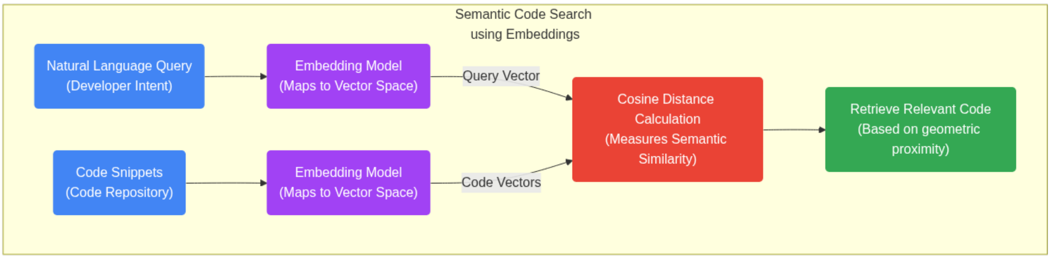

Semantic Code Search using Embeddings
Natural Language Query (Developer Intent)
Embedding Model (Maps to Vector Space)
Query Vector
Code Snippets (Code Repository)
Embedding Model (Maps to Vector Space)
Code Vectors
Cosine Distance Calculation (Measures Semantic Similarity)
Retrieve Relevant Code (Based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

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

### 2.2.2 Structural Code Search with Domain-Specific Languages

Recent research by Limpanukorn et al. (2025) introduces structural code search by translating natural language queries into 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 refers to generated information that seems plausible but is fabricated or factually incorrect. Spracklen et al.'s (2025) research revealed that package hallucinations are a systemic issue across state-of-the-art code-generating models [6]. They analysed over 576,000 generated code samples across 16 Large Language Models (LLMs) and found that the LLMs consistently hallucinate package names. More critically, they regenerate the same false package name across 43% of repeated prompts [6].

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

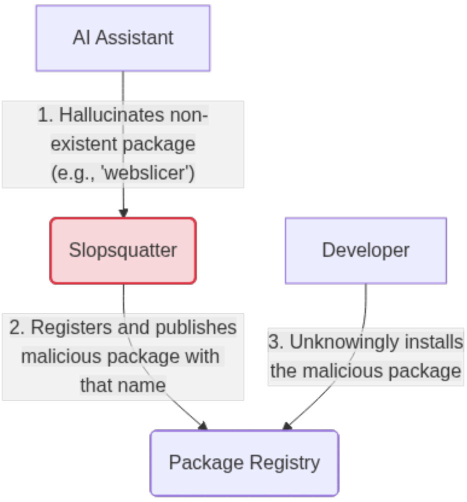

AI Assistant
1. Hallucinates non-existent package (e.g., 'webslicer')
Slopsquatter
Developer
2. Registers and publishes malicious package with that name
3. Unknowingly installs the malicious package
Package Registry

Figure 2.3: *Exploiting LLM hallucinations through slopsquatting.*

### 2.3.2 Citation-Grounded Code Comprehension

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

### 2.3.3 Retrieval-Augmented Generation

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

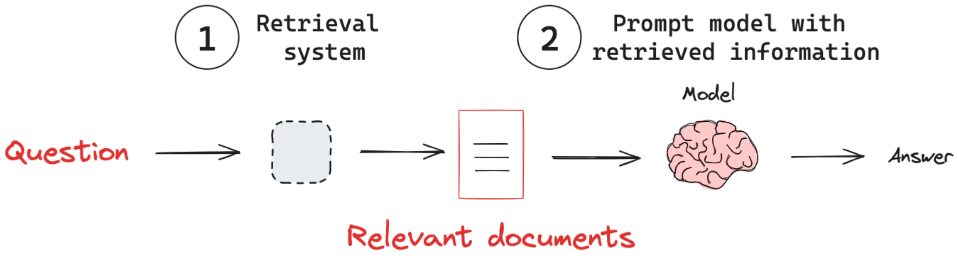


Figure 2.4: *Retrieval-Augmented Generation to reduce hallucinations [7].*

## 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]. Without this visualisation, it is inherently difficult for developers to grasp the complex relationships present 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 their 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. showcases 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 than 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:

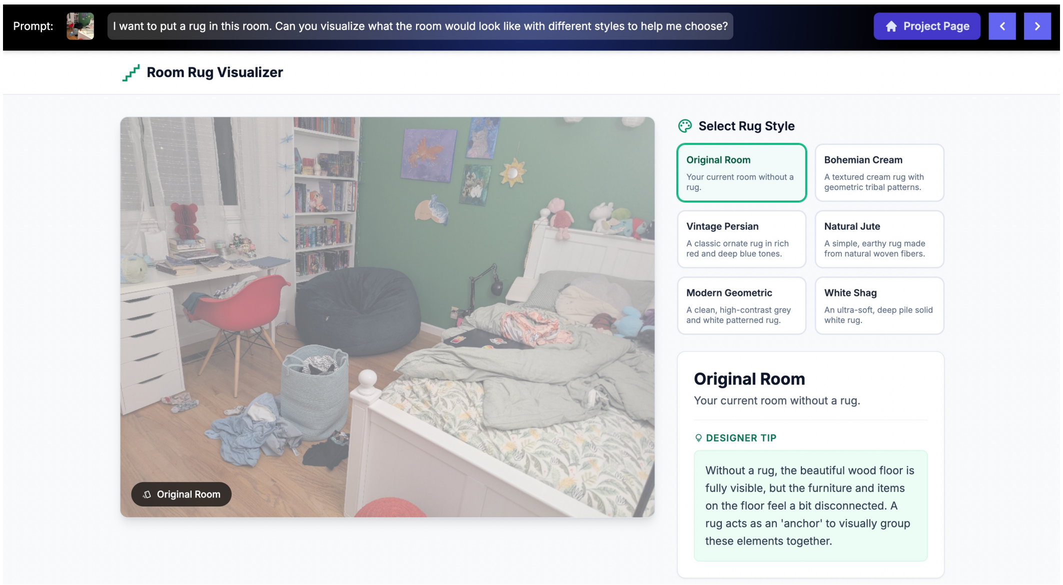
### 2.7.1 Bridging Semantic and Structural Code Search

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 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.

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



I want to put a rug in this room. Can you visualize what the room would look like with different styles to help me choose?

Select Rug Style

Original Room — Your current room without a rug.

Bohemian Cream — A textured cream rug with geometric tribal patterns.

Vintage Persian — A classic ornate rug in rich red and deep blue tones.

Natural Jute — A simple, earthy rug made from natural woven fibers.

Modern Geometric — A clean, high-contrast grey and white patterned rug.

White Shag — An ultra-soft, deep pile solid white rug.

Original Room

Your current room without a rug.

DESIGNER TIP

Without a rug, the beautiful wood floor is fully visible, but the furniture and items on the floor feel a bit disconnected. A rug acts as an 'anchor' to visually group these elements together.

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 reliable and accurate.

# Chapter 3

# System Design and Architecture

## 3.1 System Overview

This section provides a high-level overview of CodeOrient's system architecture on how it transforms a user's query into a citation-backed technical response with relevant code snippets and visualisations.

### 3.1.1 Core Components

The main components of CodeOrient include:

- **User Interface:** Built with Next.js and React Flow. It provides an end to end experience to input queries, visualise code dependencies, and view generated responses.
- **LLM Integration:** Utilises Vercel AI SDK and OpenAI for natural language processing, autonomous tool calling, and response generation.
- **Code Discovery Engine:** Leverages GitHub Search API to find relevant code snippets and index them for semantic retrieval.
- **Storage Solutions:** PostgreSQL for structured data, Upstash Redis for caching and Upstash Vector for embedding storage.

### 3.1.2 Data Flow Architecture

CodeOrient handles user interaction througha multi-stage pipeline that is optimised for speed and quality of responses. The lifecycle is split into two phases: *Code Indexing Cycle* and *Query-Response Cycle*.

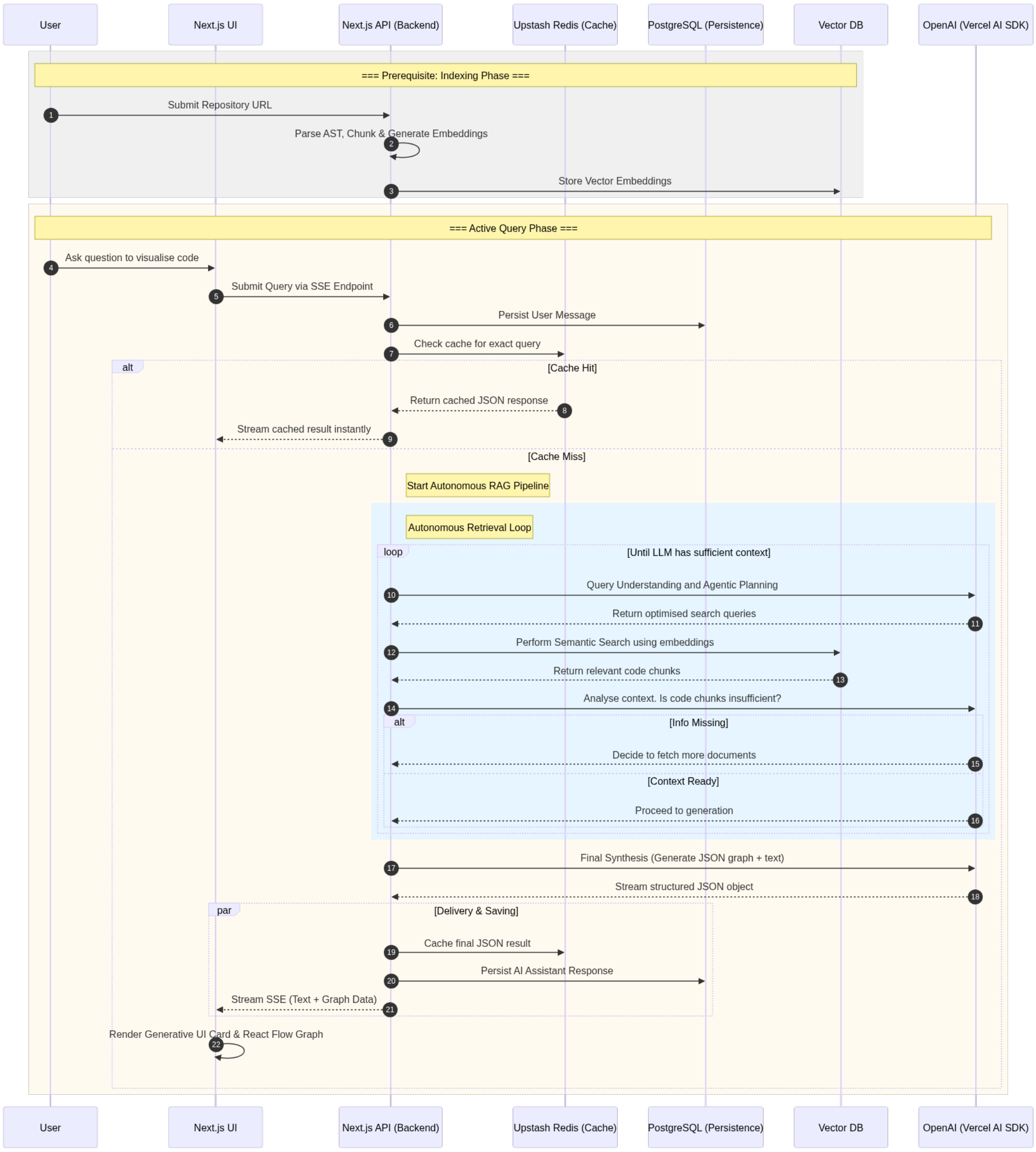

User
Next.js UI
Next.js API (Backend)
Upstash Redis (Cache)
PostgreSQL (Persistence)
Vector DB
OpenAI (Vercel AI SDK)
=== Prerequisite: Indexing Phase ===
1 Submit Repository URL
2 Parse AST, Chunk & Generate Embeddings
3 Store Vector Embeddings
=== Active Query Phase ===
4 Ask question to visualise code
5 Submit Query via SSE Endpoint
6 Persist User Message
7 Check cache for exact query
alt
[Cache Hit]
8 Return cached JSON response
9 Stream cached result instantly
[Cache Miss]
Start Autonomous RAG Pipeline
Autonomous Retrieval Loop
loop
[Until LLM has sufficient context]
10 Query Understanding and Agentic Planning
11 Return optimised search queries
12 Perform Semantic Search using embeddings
13 Return relevant code chunks
14 Analyse context. Is code chunks insufficient?
alt
[Info Missing]
15 Decide to fetch more documents
[Context Ready]
16 Proceed to generation
17 Final Synthesis (Generate JSON graph + text)
18 Stream structured JSON object
par
[Delivery & Saving]
19 Cache final JSON result
20 Persist AI Assistant Response
21 Stream SSE (Text + Graph Data)
22 Render Generative UI Card & React Flow Graph

Figure 3.1: *Data Flow Architecture of CodeOrient*

### 3.1.3 Code Indexing Cycle

This phase is responsible for ingesting code repositories via GitHub API. This is a prerequisite step which parses the source code files, chunks them, generates semantic embeddings, before storing them into a vector database for efficient retrieval.

### 3.1.4 Query-Response Cycle

When a user asks a question, the system goes through the following steps:

1. **Input with Persistence** The user submits query via the frontend built with Next.js. The message is saved to PostgereSQL for chat history persistence.

2. **Cache Layer** The backend will first check Upstash Redis for an identical query in the same session:

   - **Cache Hit:** If a cached response exists, the stored JSON response is retrieved and streamed back to the UI, which cuts down latency over 90%.

   - **Cache Miss:** If no cached response is found, the system initiates the Retrieval-Augmented Generation (RAG) pipeline.

   All user-system interactions are persisted in a relational database, while the Redis layer acts as a high-speed ”buffer” for repeated technical queries, significantly reducing LLM token costs and latency.

3. **Autonomous RAG Pipeline** This pipeline serves as the core reasoning engine:

   - **Query Understanding & Planning:** The LLM first understands and rephrases the user's natural language query. This step is crucial to fix potential errors such as typos, ambiguous terms, or lack of context. Next, the LLM generates a plan to decompose the query into sub-tasks, generating up to 3 sub-queries for retrieval.

   - **Semantic Retrieval:** Each sub-query is converted into embeddings and the system retrieves the top-K relevant code chunks from the vector database.

   - **LLM Evaluation Loop:** The LLM evaluates the retrieved content. If the context is deemed insufficient, it automonously triggers further searches in a loop until it gathers sufficient context to confidently answer the original query.

4. **Citation-Grounded Response:** Once sufficient context is gathered, the LLM generates a structured JSON object that contains the required nodes, edges, code snippets (citations) and explanation required for visualisation and response generation.

5. **Message Persistence & Caching:** The final result is stored in PostgreSQL for chat history and also cached in Upstash Redis for future identical queries. Simultaneously, the response is streamed via Server-Sent-Events (SSE) back to the frontend for real-time rendering.

6. **Interactive Rendering:** Next.js parses the streamed JSON response to render as a Generative UI card that contains an interactive graph visualisation built with React Flow, along with citation-backed explanations and code snippets.

### 3.1.5 Serverless Service Architecture

CodeOrient utilises Next.js API Route Handlers to create a modular, serverless backend. This separates logical concerns whle 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 & 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. The page features a split-pane design that prioritise the chat interface with the AI assistant and the source code.

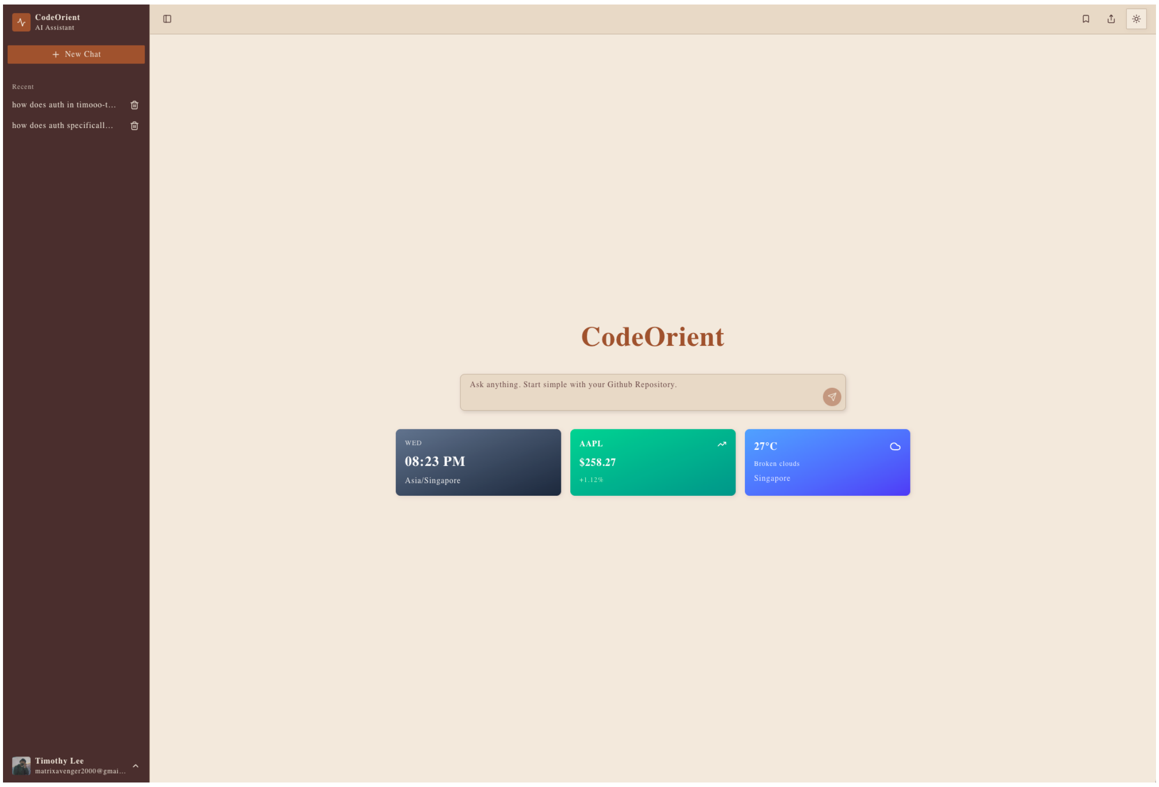

CodeOrient
AI Assistant
+ New Chat
Recent
how does auth in timooo-t...
how does auth specificall...
Timothy Lee
matrixavenger2000@gmai...
CodeOrient
Ask anything. Start simple with your Github Repository.
WED
08:23 PM
Asia/Singapore
AAPL
$258.27
+1.12%
27°C
Broken clouds
Singapore

Figure 3.2: *CodeOrient Chat Page Layout*

- **Persistent Sidebar:** Contains a collapsible left sidebar that manages user context, providing quick access to their conversation history in chronological order. It also includes a user profile section at the bottom for account settings and logout.
- **Navigation Bar:** The top navigation bar includes sharing options, bookmarking and toggling of light/dark mode for user convenience.
- **Main Chat Area:** The central area occupies the main viewport, where the AI assistant interacts with the user via Generative UI or text-based responses.
- **Responsive Design:** The layout adapts to different screen sizes, ensuring usability across laptops, tablets, and mobile devices.

### 3.2.3 Interactive Graph Visualisation with React Flow

Static architecture diagrams often disconnect users from the actual source code. This makes it hard to navigate and update dynamically as a user explores various parts of the repository. To combat this pain point, CodeOrient integrates React Flow to create a dynamic canvas that visualises a subset of the code repository. Hence, this allows for a real-time and interactive exploration of the codebase.

1. **Nodes as Entities:** High-level code entities such as files, functions, classes, and components are represented by individual nodes. Each node carries metadata including the entity type, file path, code snippet and description.
2. **Edges as Relationships:** Dependencies are presented as directed edges between nodes. These illustrate how data and logic flow between different parts of the codebase. Common relationship types include imports, calls, extends, and uses. For example, Component A imports Function B would be represented as an edge from Node A to Node B.
3. **Interactive Features:** Users can pan, zoom, and click or hover over nodes to reveal additional information such as code snippets, code language, file path and description. This interactivity enhances understanding and navigation within the codebase.

### 3.2.4 Generative UI Card Components

Standard LLM responses are limited to text or markdown formats. 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. This allows for richer and more interactive responses.

With the structured JSON object, the frontend acts as an orchestrator, parsing the JSON and mapping it to a React component. For example, a JSON block describing a graph visualisation would be rendered as a ”Graph Card” component rather than a bulleted list.

To achieve this, CodeOrient leverages Vercel AI SDK and its `streamObject` function. Based on the user's query, the LLM automonously decides which component to generate. For example, if the user asks about their list of repositories, the LLM would generate a ”Repository Card” component. The component data is streamed into the UI, which the card to populate incrementally. This creates a smooth user experience by reducing perceived latency.

## 3.3 Backend Architecture

### 3.3.1 API Design

The backend of CodeOrient is built on a RESTful Serverless architecture using Next.js API Route Handlers.

1. **Stateless Functions:** Each API endpoint is implemented as a stateless function. This allows the system to scale horizontally, as each function can be invoked independently based on demand.

2. **Endpoint Categories:** The API is divided into four primary domains:

   - **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. This hybrid approach allows the engine to find relevant code even if the user's query does not explicitly match the code's keywords or function names. Apart from the code itself, the engine leverages the docstrings and comments extracted during the indexing phase to improve context matching.

3. **Scoping and Filtering:** The engine partitions searches by repository and user ID to prevent cross-project contamination. Additionally, the engine supports metadata filtering, such as file type or entity type, to refine search results.

4. **Ranking Mechanism:** Retrieved code snippets are ranked using Distributed-Based Score Fusion (DBSF), which combines results from dense and sparse indexes. The normalised score is computed as:

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

   where:

   - $s$ is the score.
   - $\mu$ is the mean of the scores.
   - $\sigma$ 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).

   For CodeOrient, DBSF was chosen because it balances semantic meaning with exact keyword match effectively. This ranking ensures that the most contextually relevant code snippets are prioritised for retrieval.

### 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 pipeline transforms a raw GitHub code repository into a structured and searchable knowledge base through a four-stage process:

### 3.4.1 Repository Ingestion

To bypass GitHub API rate limits and ensure efficient ingestion, the system uses a streaming ingestion strategy.

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:** Using the `tar` and `zlib` libraries, the system decompressed and parses files in-memory without writing to disk. This reduces I/O

overhead and speeds up processing.

3. **Indexing with Filtering:** The pipeline whitelists indexable file types (e.g., `.ts`, `.py`, `.go`) and ignores binary or large files (e.g., `node_modules`) to optimise storage and retrieval efficiency.

### 3.4.2 Chunking Strategy

Instead of indexing entire files, the engine breaks code files into smaller ”logical” chunks to ensure the LLM receives the most relevant context.

1. **Entity-Based Parsing**: The parser isolates code blocks such as standalone functions, classes, or React components as an individual entity type. Each type is assigned a chunk size limit to optimise retrieval:

| **Entity Type** | **Max Character Limit** | **Rationale** |
|---|---|---|
| File Header | 2,000 chars | Captures imports and high-level file intent. |
| Classes | 2,500 chars | Provides structural context of data and methods. |
| Functions/Components | 1,500 chars | Focuses on a single unit of execution logic. |
| Methods | 1,500 chars | Enables pinpoint retrieval of class behaviors. |
| Notebook Cells | 2,000 chars | Balances code logic with markdown context. |

Table 3.1: Entity-Based Chunk Size Limits and Rationale

2. **Truncation Strategy:** If an entity exceeds its designated length limit, the engine uses a `truncateAtBoundary` function to prevent a hard character cut. Instead, it searches for the last complete statement by looking for delimiters. This ensures that code chunks remain syntactically valid and semantically coherent.

### 3.4.3 Metadata Extraction

To assist LLM with context understanding, each code chunk is enriched with relational metadata:

1. **Docstring Extraction:** The parser scans for docstrings or comments immediately preceding code entities. These high-level summaries provide a natural language description of what the code does.

2. **Dependency Mapping:** The system extracts `imports` and `calledFunctions`. This enables the LLM to understand the relationships between different code entities during reasoning, which increases the accuracy of generated responses.

3. **Source Attribution:** Each chunk is tagged with its `filePath`, `repoFullName`, and specific `startLine/endLine`. This creates the ground truth for the citation mechanism during response generation.

### 3.4.4 Indexing and Storage

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

1. **State Persistence:** The system uses Prisma ORM and PostgereSQL to track the indexing lifecycle of each repository (e.g., `CLONING` → `PARSING` → `INDEXING` → `‘CO’MPLETED`). This allows for resumable indexing in case of interruptions. Additionally, it allows the UI to show real-time progress updates to the user.

2. **Batch Vector Upsert:** Code chunks are converted into embeddings using `bge-large-en-v1.5` model and upserted into Upstash Vector in batches of 100. This strategy optimises throughput and ensures the indexing process is resilient to transient API issues.

## 3.5 Technical Stack Summary

| Category | Technology | Engineering Justification |
|---|---|---|
| **Framework** | Next.js 16 | Enables a unified architecture of Server Components for data safety and Client Components for complex React Flow interactivity. |
| **AI Framework** | Vercel AI SDK | Provides native primitives for `streamObject`, enabling real-time Generative UI rendering and SSE-based communication. |
| **LLM Provider** | OpenAI | Offers state-of-the-art models with robust capabilities in code understanding and multi-turn reasoning. |
| **Database & ORM** | PostgreSQL & Prisma | Ensures robust relational integrity for managing complex user, repository, and chat-session hierarchies via Prisma ORM. |
| **Vector Store** | Upstash Vector | A serverless vector database that allows for high-dimensional semantic indexing with native metadata filtering for multi-repo isolation. |
| **Caching** | Upstash Redis | Acts as a high-speed buffer to intercept repeated semantic queries, reducing LLM token consumption and decreasing latency by over 90%. |
| **Visualisation** | React Flow | Offers the capabability of rendering dynamic nodes and edges derived from code analysis. |
| **Ingestion** | Octokit (GitHub) | Facilitates secure fetching of code repositories from GitHub. |

Table 3.2: Summary of Technical Stack and Rationale

# 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.