AI-supported Code and Documentation Analysis for PL/SQL Systems

Bachelor Thesis

Submitted by Julian Köser

In fulfillment of the requirements for the degree
Bachelor of Science (B.Sc.) in Information Technology
To be awarded by the Fontys University of Applied Sciences

Venlo, June 17, 2025

Information Page Graduation Report

Fontys University of Applied Sciences
Technology and Logistics
Tegelseweg 255, 5912 BG Venlo, Netherlands

Type of report:	Graduation
Student name:	Julian Köser
Student number:	4676661
Study:	Bachelor of Science in Business Informatics
Period:	January 2025 - June 2025
Company name:	Accso - Accelerated Solutions GmbH
Adress:	Im Mediapark 6a
Postal code + City:	50670 Köln
Country:	Germany
Telephone:	0221 6306910
Company supervisor:	Burghof, Axel
Supervising Lecturer:	Schwake, Gregor
Word count:	9,987
Company Confidential:	No

STATEMENT OF AUTHENTICITY

Issued by the FTenL Examination Board, September 2017

I, the undersigned, hereby certify that I have compiled and written this document and the underlying work / pieces of work without assistance from anyone except the specifically assigned academic supervisor. This work is solely my own, and I am solely responsible for the content, organization, and making of this document and the underlying work / pieces of work.

I hereby acknowledge that I have read the instructions for preparation and submission of documents / pieces of work provided by my course / my academic institution, and I understand that this document and the underlying pieces of work will not be accepted for evaluation or for the award of academic credits if it is determined that they have not been prepared in compliance with those instructions and this statement of authenticity.

I further certify that I did not commit plagiarism, did neither take over nor paraphrase (digital or printed, translated or original) material (e.g. ideas, data, pieces of text, figures, diagrams, tables, recordings, videos, code, ...) produced by others without correct and complete citation and correct and complete reference of the source(s). I understand that this document and the underlying work / pieces of work will not be accepted for evaluation or for the award of academic credits if it is determined that they embody plagiarism.

Name: Julian Köser

Student number: 4676661

Place / Date: Cologne, 17.06.2025

Signature:

List of Abbreviations

Abbreviation	Full Form
AI	Artificial Intelligence
API	Application Programming Interface
FR	Functional Requirement
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
LLM	Large Language Model
NFR	Non-Functional Requirement
PL/SQL	Procedural Language/Structured Query Language
RAG	Retrieval-Augmented Generation
SWOT	Strengths, Weaknesses, Opportunities, Threats
UI	User Interface

Abstract

This report will discuss how Large Language Models (LLMs) can help in software maintenance challenges, to be excat understanding and maintaining legacy PL/SQL code. The project presents my graduation project and was done at the company Accso - Accelerated Solutions GmbH. It was a proof of concept system designed and implemented which enables developers to query PL/SQL cod using natural language questions.

This report looks into whether current LLM technology can effectively analyze code relationships when integrated into a Retrieval Augmented Generation (RAG) architecture, especially with a language like PL/SQL. The approach taken incorporates carefully selected components: semantic code chunking methods to preserve code logic, BGE-M3 for embeddings, and Qdrant for vector storage.

The final system demonstrates a AI assisted code analysis which operates in limited hardware resources, achieves interactive responses in under 3 minutes while receiving responses of high quality. It was tested with representative PL/SQL code but there were limitations identified regarding the correct context of chunks.

This project contributes valuable insights to AI assisted code understanding for legacy systems, identifying both the potential and current limitations of applying LLM technology to specialized code analysis. Future work will focus on enhancing context provision, optimizing similarity thresholds, and implementing entity verification mechanisms to address identified limitations.

Contents

List of Figures

List of Tables

1. Background

1.1 Company

Accso - Accelerated Solutions specializes in software development and digital transformation. The company supports clients in modernizing existing IT systems and developing new digital solutions, focusing on technical implementation and close collaboration with customers. Accso's projects cover various areas of software development. A major focus is on the architecture and structuring of complex IT systems. In large, evolving software landscapes, a clear structure ensures maintainability and future development opportunities. Additionally, the company works on process automation to make workflows more efficient and reduce manual tasks.

1.2 Problem Context: Government funding Application System

Project Background

One of these complex IT modernization projects in which Accso is engaged includes an especially challenging legacy system that Accso helps maintain for a government agency. This system serves as a platform for managing many funding programs that are backed by the state, and it shows the challenges of maintaining business critical applications developed over decades.

Business Domain and System Overview

The core of this system is built on Oracle Forms technology and handles the complete lifecycle of government funding applications.

A lot of sensitive data is managed, ranging from the initial application through approval, disbursement, and continuous oversight. The system also needs to follow state regulations, federal guidelines, and audit requirements. All these processes naturally include a lot of sensitive documentation, which needs to be handled appropriately. The system processes thousands of applications annually across different funding programs, each with distinct eligibility criteria, calculation methods, and approval workflows. It is only used within the customer's company.

System Complexity and Challenges

Through the size of the legacy Oracle Forms application, it presents several maintenance challenges. The system contains over 5,000 PL/SQL packages containing business logic, over 500 Oracle Forms modules, and 25 year-old code. This creates complex interdependencies in the system. Additionally, there are state specific regulations that change over time, different calculation methods for various funding programs, different requirements for the audit trails, and over 10,000 applications being processed annually with different complexity levels.

Limitations of Current Development Tools

While a modern IDE offers a lot of help, navigating the code landscape, there are still many limitations. Oracle IDEs like SQL Developer and Forms Builder have significant limitations when it comes to search capabilities, understanding relationships and tracing function calls. For example a developer only can search for exact text matches and has no assistance in understanding semantic relationships between code components. So "what if" questions about code changes aren't possible.

Additionally, to understand the code manual tracing is needed. Of course it's easy to find the functions and packages but each file needs to be opened individually and there is a risk of human error, like missing dependencies.

Real World Problem Scenario

To illustrate these challenges in practice, consider the following example scenario: A critical issue is discovered in the calculation logic for project funding. The problem affects funding disbursements and may impact compliance with state funding regulations.

Sarah, a developer on the team, needs to navigate herself through over 500 packages to identify in which the faulty calculation is located. She also needs to understand what other funding Forms might be affected across the 500+ Forms modules. After the issue has been resolved, it's also important to know hich modules need testing afterwards.

Current Manual Process:

At the current moment Sarah of course can search for the "calculate_funding_amount" function but this only shows where the function is defined and it's direct references. To get the broader picture and undertand what is affected by the function, she needs to start manually tracing the relationships:

First, she needs to open each package that references the function, then identify which other functions call these references. Next, she must trace these functions to the Forms modules that use them. Finally, she needs to determine which database tables might be affected by any changes. This process can take days or weeks to trace all dependencies manually across such a large system, with a high risk of missing critical connections.

Developer Information Needs

Developers working on this system regularly need to answer questions such as:

- "Which packages call the calculate_funding_amount function?"
- "What tables are modified when processing a small funding application?"
- "Which Forms modules are affected by changes to the applicant validation package?"
- "What funding programs share the same calculation logic?"
- "What dependencies exist between the document verification and approval processes?"

While this project represents a specific case, it reflects a recurring problem across many legacy system projects. Understanding the general pattern helps to better contextualize the approach which is taken in this project.

1.3 Need for a Structured Solution

Thru the complexity of state funding processing, the large amount of data that needs to be taken into account, and the 25+ year old codebase, the idea formed to bring AI assisted help into this system. The main motivation comes from other projects from Accso where AI has been found very useful.

Such a system could help developers quickly understand code relationships, trace dependencies, and ensure that modifications maintain compliance with state lending regulations.

Therefore, a solution is needed that allows quick information access within the codebase, taking context and security into account. The goal is to enable developers to ask natural language questions and receive informative answers that would reduce time spent on manual search.

Now the difficulty in this project is the system has a sensitive nature and source code protection requirements that mandate that only locally operated AI solutions can be considered. Based on this constraint we can't use cloud based AI services and it requires a self contained approach that maintains data sovereignty and security compliance.

This project aims to develop an exploratory prototype system that leverages large language models (LLMs) to process natural language queries about the PL/SQL codebase. As a proof of concept, the prototype will demonstrate whether this approach is feasible and effective for improving code comprehension and maintenance efficiency. The project acknowledges the experimental nature of applying LLMs to specialized legacy code and will document both successes and limitations encountered during development.

2. Project Management

This chapter outlines the project management approach used for developing the AI supported code analysis system. It covers the project objectives, risk assessment, scope definition, stakeholder analysis, development approach, methodology, and quality assurance measures, all tailored to the exploratory and proof of concept nature of this work.

2.1 Objective

The primary objective of this project is to develop an experimental prototype system that enables developers to access and understand critical information from an existing Oracle-based PL/SQL codebase through natural language queries. This proof of concept aims to demonstrate the feasibility of using Large Language Models (LLMs) to assist with code comprehension and navigation, while acknowledging the limitations and challenges of such an approach.

Additionally, the project seeks to evaluate the limits of current LLMs in understanding PL/SQL code and establish a foundation for a potential future exploration of AI applications in legacy code maintenance.

Success for this project is defined as either a working prototype that meets the later defined requirements and shows potential for this approach, or a clear understanding of the specific limitations that prevent such a system from being fully realized at this time.

In either case, the project will produce valuable insights about the current state of LLM technology as applied to specialized code comprehension tasks. The focus is strictly on code comprehension and information retrieval, not code generation or modification.

2.2 Scope

After defining the project objective, the next step was to define clear boundaries, which help to prevent scope creep later and maintain focus on the core objective. To define the project scope, the funding system described in Chapter 1 needed to be analyzed. This helped to identify which aspects of the AI assisted code analysis challenge could realistically be addressed within the constraints of a graduation thesis project, and what would be meaningful evidence for the proof of concept.

By setting the boundaries, a foundation for decisions about methodology, technology selection, and success criteria is laid.

In Scope

The following elements are within the scope of this exploratory project, based on the analysis of the government funding system's complexity:

- **Experimental Prototype Development**: Creating an initial version of the system as a proof of concept, tested in a controlled environment using representative PL/SQL code samples.
- **LLM Integration**: Exploring how existing LLMs can be applied to process queries and retrieve relevant sections of PL/SQL code.
- **Natural Language Query Support**: Evaluating the feasibility of enabling users to ask technical questions in plain language about the PL/SQL codebase.
- **Feasibility Assessment**: Systematically documenting both the capabilities and limitations of the approach to provide a clear picture of its potential use.

Out of Scope

The following elements are explicitly excluded from the scope of this proof of concept project:

- **Code Modification or Generation**: The system will not modify, rewrite, refactor, or generate PL/SQL code. It is strictly designed for code understanding and information retrieval.
- **Production Ready Implementation**: As a proof of concept, the system is not expected to meet all the requirements of a production ready tool in terms of performance, security, or user interface.
- **Adaptation to Other Projects**: While insights may be applicable elsewhere, this phase focuses solely on evaluating feasibility for the current Oracle based system.
- **Training Custom LLMs**: The project will utilize existing LLMs rather than training new models from scratch, focusing on evaluating current capabilities rather than developing new ones.
- **Comprehensive Solution**: The prototype is not expected to address all aspects of code comprehension or to handle all possible queries with perfect accuracy.

2.3 Stakeholders

With the project scope clearly defined, the next step was to identify and analyze all stakeholders who would be affected by or could influence the success of this AI assisted code analysis project. Understanding stakeholder perspectives became crucial for shaping both the technical approach and the evaluation criteria for the proof of concept.

Project Team

As the primary researcher and developer for this graduation thesis project, I am responsible for all aspects of the project's execution, including:

- Designing and implementing the experimental system architecture
- Researching and selecting appropriate LLMs for evaluation
- Structuring test data for efficient retrieval
- Developing and testing the natural language query system
- Documenting both capabilities and limitations encountered

I am supported by my university supervisor, who provides academic guidance and ensures the project meets the requirements for my degree program. Additionally, I have access to experienced professionals at Accso who offer domain expertise in LLMs and the Oracle based system. These mentors provide valuable insights and feedback throughout the development process, particularly in evaluating the practical utility of the approach.

End Users

The system's feasibility will be evaluated against the needs of three distinct groups of potential users:

- **Developers**: Maintain the existing PL/SQL codebase and implement new features. They would use the system to quickly retrieve relevant code, dependencies, and references.
- **Support Engineers**: Answer technical questions and troubleshoot issues. They require a deep understanding of the codebase and could use the LLM to assist in identifying relevant code sections.
- **Testers**: Verify new functionality added to the Oracle based project. They would use the system to understand the logic behind implemented changes and validate expected behavior.

Input from representatives of these groups will be valuable in assessing whether the prototype demonstrates sufficient potential to warrant further development beyond this proof of concept stage.

Secondary Stakeholders

- **Senior Management**: Interested in the potential of AI assisted approaches to improve maintenance efficiency for legacy systems.
- **Security and IT Compliance**: Concerned with ensuring that any AI based approach to code analysis adheres to security policies when processing sensitive internal code.

2.4 Approach

After clear goals and scope are set, the project is built on three structured phases designed to effectively explore the feasibility of applying LLMs to PL/SQL code analysis

Research and Foundation Phase

The project starts with a 6-week phase which focuses on gathering an information foundation by exploring the problem and the current technical landscape. Key activities include stakeholder analysis, technology evaluation of LLMs and embedding models, best practices research, PL/SQL code structure analysis, and system architecture design. This phase helps to have an informed and structured starting point for the development.

Component Development Phase

Afterwards, the development phase starts, which is set for 8 weeks in total. In this the RAG system is implemented with its core components. Each component is developed and tested independently before integration, allowing for focused problem solving and earlier identification of any limitations.

Evaluation and Optimization Phase

Lastly, there is an 8-week phase in which the system's capabilities are evaluated, documented, and a final feasibility assessment and recommendations are created.

2.5 Methodology

This project utilizes a hybrid approach combining waterfall structure and agile development practices. On the higher level the project is structured based on the previously mentioned three phases which need to be completed sequentially to ensure a proper foundation. However, within individual phases, agile practices are used to maintain flexibility and separate concerns of each component. This includes 2-week sprints, weekly Monday meetings, and regular delivery of components.

2.5.1 Milestone Structure

Phase	**Start Date**	**End Date**	**Milestor**	**Key Deliverables**
Research and Foundation	17.02.2025	30.03.2025	M1	Stakeholder requirements, technology evaluation, system architecture design, risk assessment, proof-of-concept validation
Component Development	01.04.2025	31.05.2025	M2	Code processing pipeline, BGE-M3 embedding integration, Qdrant vector database, query processor, LLM integration, basic UI
Evaluation and Optimization	01.06.2025	31.07.2025	M3	Performance optimization, similarity threshold tuning, comprehensive testing, accuracy assessment, final feasibility report

Table 2.1: Project Phases and Milestones

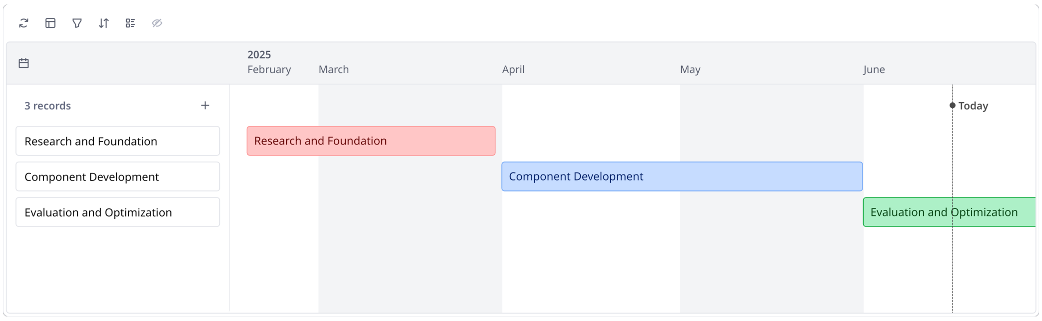

Figure 2.1: Project Gantt Chart showing phase timeline and component development schedule

2.5.2 Project Tracking

Project progress is tracked using GitLab's integrated project management tools. All work is organized into structured issues with detailed descriptions, task breakdowns, and milestone associations, supporting both the systematic documentation requirements of the waterfall structure and the iterative refinement needs of agile development.

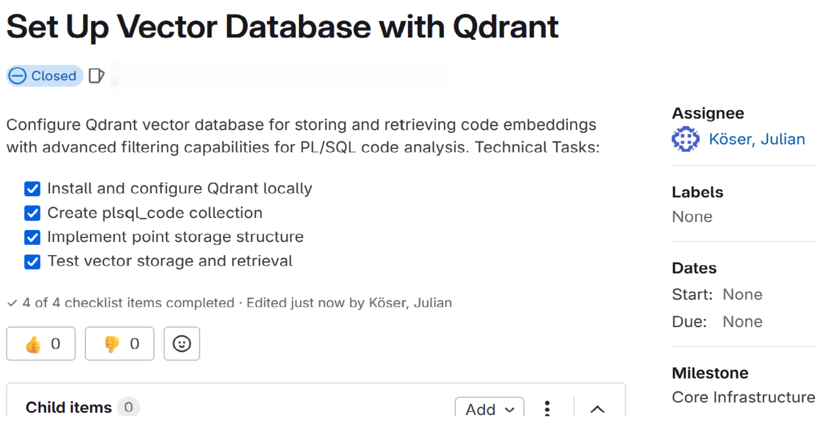

Figure 2.2: Example Issue from GitLab showing structured task management

This systematic approach ensures comprehensive documentation of both technical implementations and experimental findings, supporting the proof of concept's objective of either demonstrating capability or clearly documenting current technological limitations.

2.6 Quality

For this project, quality measures focus on evaluating feasibility rather than production readiness. The approach acknowledges the experimental nature of applying LLMs to specialized legacy code and aims to capture insights about both capabilities and limitations.

Quality Criteria

The prototype's quality will be assessed across dimensions aligned with the project's exploratory nature:

- **Capability Assessment**: The system should demonstrate what's currently possible with LLM technology applied to PL/SQL code, with traceable responses to relevant codebase parts. Perfect accuracy isn't expected, but achieved accuracy will indicate feasibility.
- **Limitation Documentation**: The project should identify and document areas where current LLM technology struggles with PL/SQL comprehension, providing insights for future work regardless of overall success.
- **Practical Utility**: Even as a proof of concept, the system should demonstrate sufficient utility to indicate whether the approach has potential value in real development. This will be assessed through user feedback rather than strict metrics.
- **Technical Feasibility**: The implementation should provide clear evidence about whether the approach is technically viable with current technology, including any fundamental limitations.

These criteria focus on learning and insight generation rather than traditional software quality metrics.

Testing Approach

The testing strategy focuses on experimental evaluation rather than validation against fixed requirements. Testing activities systematically explore possibilities and identify specific limitations:

- **Capability Testing**: Representative queries will assess what the system can successfully handle, providing a baseline understanding of current capabilities.
- **Limitation Probing**: Deliberate exploration of edge cases and challenging queries will identify specific limitations, which are as

important to document as successes.

This approach treats the entire project as an experiment, with each test contributing to overall feasibility understanding.

Acceptance Criteria

The acceptance criteria acknowledge that success can take multiple forms:

- **SC-001: Demonstrated Potential**: The prototype shows sufficient capability (correctly answering approximately 60-70% of predefined test queries) indicating promise for further development.
- **SC-002: Clear Limitation Identification**: The project identifies specific limitations in applying current LLM technology to PL/SQL code, providing guidance for future approaches.
- **SC-003: Practical Insight Generation**: The project produces documented insights about applying LLMs to legacy code comprehension that advance understanding in this area.

The prototype should process queries within a reasonable timeframe for interactive testing, but optimization isn't a primary focus.

The most important outcome is understanding whether and how LLM technology can be applied to PL/SQL code comprehension, whether through successful implementation or well documented limitations. This balanced view acknowledges the project's exploratory nature and ensures valuable insights even if the technology proves to have significant limitations for this application.

3. Project Analysis

This chapter presents the analysis conducted to understand the problem domain, user needs, and system requirements for the AI supported code analysis proof of concept. The analysis forms the foundation for the experimental design and implementation decisions that follow.

3.1 Analysis Approach

The analysis stage explored the Oracle PL/SQL ecosystem, user requirements by role, and prototype requirements. The analysis looked at PL/SQL code structure in Oracle 19c to determine LLM understanding difficulties, held developer interviews to determine pain points, and reviewed current workflows to determine LLM use cases. The assessment took into account both promising advantages and intrinsic limitations of using LLM technology for niche legacy code.

3.2 Risk

The experimental nature of applying LLMs to specialized legacy code introduces unique uncertainties that require careful monitoring. The most critical risk is LLMs' potential limitation in understanding PL/SQL syntax, which directly affects the core feasibility of the proof of concept. To mitigate this, a controlled test environment with sample code will be created to evaluate LLM comprehension before proceeding with full implementation.

Table 3.1: Key Risk Assessment

Risk	Prob.	Impact	Level	Mitigation Strategy
LLM limitations in understanding PL/SQL syntax	High	High	Critical	Create a controlled test environment with sample PL/SQL code first; evaluate LLM comprehension before full implementation; select models with proven code understanding capabilities
Inaccurate or misleading responses	Med	High	High	Implement validation mechanisms; provide source references; continuous testing
Complex code interpretation challenges	High	Med	High	Use structured parsing; refine query-matching logic; create specialized embeddings
Security concerns	Med	High	High	Design for local execution; implement access controls; ensure security compliance
Timeline constraints	High	Med	High	Use iterative approach; clear milestones; prioritize core functionality

The approach allows for incremental feasibility assessment and adaptation based on early findings. For each identified risk, specific mitigation strategies have been developed that acknowledge the project's experimental nature. The risk matrix will be reviewed regularly throughout development, ensuring the project delivers valuable insights even if current LLM technology proves to have significant limitations for this application.

3.3 Domain Model

The domain model captures the key entities and relationships in the proposed AI supported code analysis system. Figure 3.1 illustrates how these components would interact in an experimental implementation.

The model identifies several key components that would need to work together in the proof of concept system. The AI Analysis System serves as the core component that would process queries, retrieve information, and generate responses. Various types of system users would interact with the prototype through queries and responses. The large language model would

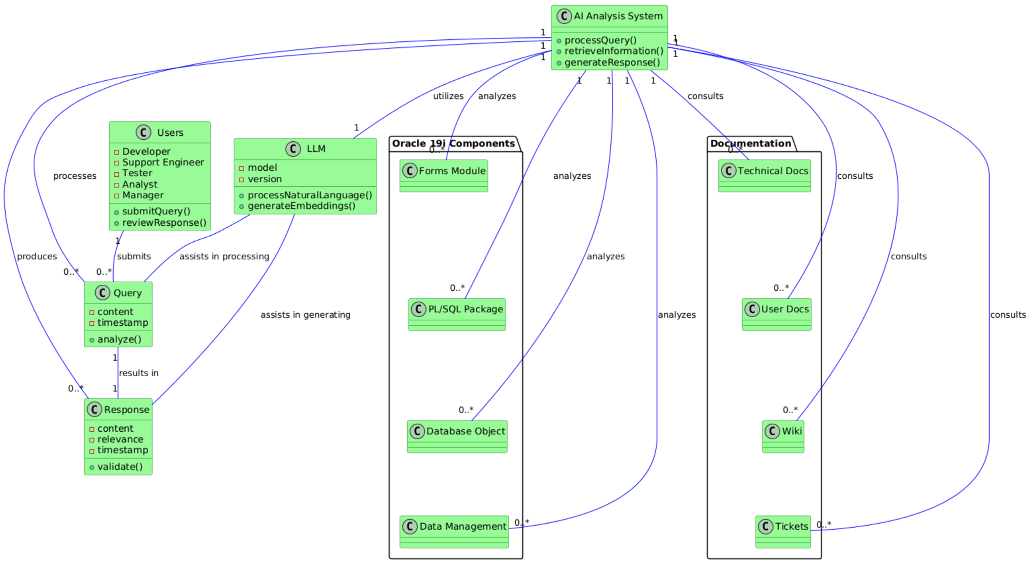

Figure 3.1: Domain Model of the AI-Supported Code Analysis System

assist in processing these queries and generating appropriate responses. The system would need to understand Oracle 19i Components including Forms Modules, PL/SQL Packages, Database Objects, and Data Management elements. Finally, various forms of documentation would be consulted to augment code understanding.

This model serves as a conceptual framework for exploring how an AI assisted system might interact with the various elements of the legacy codebase and its users.

3.4 User Stories

Stakeholder interviews and workflow analysis revealed the following user stories across roles interacting with the legacy PL/SQL codebase. These represent potential areas for AI assistance, with feasibility to be determined through our proof of concept.

ID	Role	User Story
US-001	Developer	As a developer, I want to quickly identify all references to specific functions or procedures, in order to understand their usage across the system.
US-002	Developer	As a developer, I want to understand dependencies between modules, in order to avoid regression issues when making changes.
US-003	Developer	As a developer, I want to access historical context and documentation, in order to maintain code effectively.
US-004	Developer	As a developer, I want visibility into database interactions, in order to know which tables are read or modified by specific code.
US-005	Tester	As a tester, I want to identify all modules affected by a specific change, in order to focus testing efforts efficiently.
US-006	Tester	As a tester, I want to access information about modified code blocks, in order to understand what has changed.
US-007	Tester	As a tester, I want to understand code dependencies, in order to ensure comprehensive test coverage.
US-008	Support Engineer	As a support engineer, I want to retrieve known issues related to specific error codes, in order to quickly diagnose problems.
US-009	Support Engineer	As a support engineer, I want to access code sections related to specific business processes, in order to troubleshoot effectively.
US-010	Support Engineer	As a support engineer, I want to understand module dependencies, in order to check for related issues when solving problems.

Table 3.2: User Stories by Role

3.5 Use Cases and Focus Selection

Overview of Identified Use Cases

Our analysis identified four key use cases that address the primary needs of different user roles. These represent potential applications of the technology that could be explored through the proof of concept:

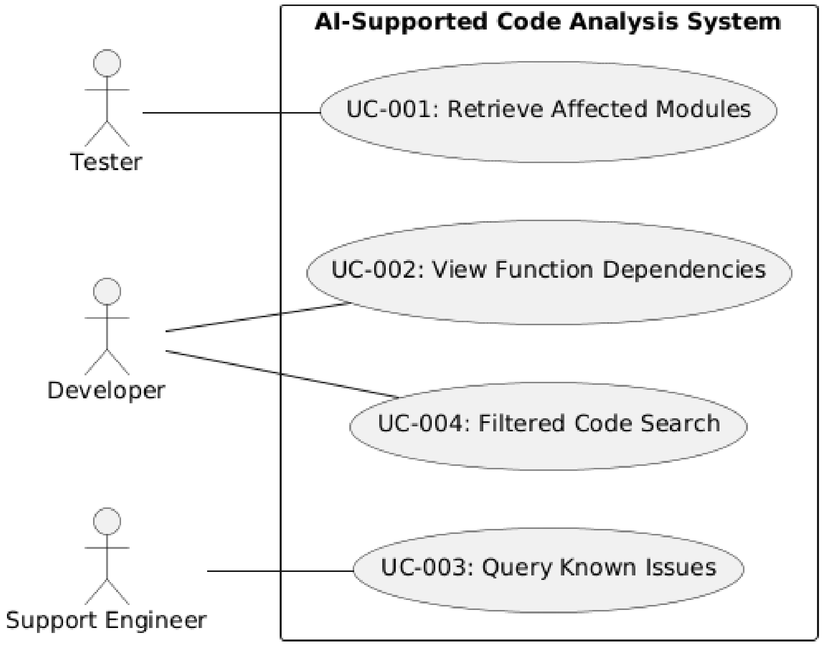

Figure 3.2: Use Case Diagram for the AI Supported Code Analysis System

The identified use cases include UC-001: Retrieve Affected Modules, which enables testers to identify all modules impacted by a specific feature or change; UC-002: View Function Dependencies, which allows developers to understand the dependencies and relationships of specific functions; UC-003: Query Known Issues, which helps support engineers find historical information about error codes and issues; and UC-004: Filtered Code Search, which provides all users with the ability to narrow search results by specific criteria.

Focus Use Case Selection For clarity and focus in this report, we selected **UC-002: View Function Dependencies** as our primary use case for detailed exploration. This use case demonstrates the system's ability to analyze code structure and relationships, addressing a time consuming aspect of PL/SQL maintenance.

UC-002: View Function Dependencies

Use Case ID	UC-002
Use Case	View Function Dependencies
Primary Actor	Developer
Brief Description	A Developer wants to retrieve a dependency map for a specific PL/SQL function to understand potential side effects before modifying it.
Preconditions	• The Developer is authenticated
	• The system's knowledge base includes the PL/SQL codebase and its dependencies
Main Flow	1. Developer opens the AI interface in their IDE
	2. Developer enters a query: "Which tables does updateCustomerAddress modify, and which packages call it?"
	3. System retrieves relevant code references and documentation
	4. System displays a dependency map showing packages that call the function and tables it modifies
	5. Developer uses this information to plan changes accordingly
Postconditions	Success: The Developer understands all relevant dependencies
	Failure: The system can't provide a useful dependency map
Alternative	**A1: Partial Information Only**
	• If the system can only find partial references, it displays a note
	• Developer may refine or expand the query

Table 3.3: UC-002: View Function Dependencies

3.6 SWOT Analysis

We undertook a SWOT analysis for the evaluation of the strategic position of the project, with the identification of potential strengths and limitations.

Strengths

It provides a less complex natural language interface than other software. It integrates code, documentation, and ticket data in a fashion other software can't. In all cases possible, it can decrease the amount of manual work for code search and retain institutional memory by making implicit relationships explicit.

Weaknesses

Current LLM technology may struggle with specialized PL/SQL syntax and complex relationships. Effectiveness will likely depend on the quality of existing documentation and comments. The proof of concept focuses on a specific Oracle based project with limited generalizability. As a prototype, it will have inherent limitations compared to mature tools.

Opportunities

If this is successful, it can then be used with other legacy code and can shorten code understanding time considerably. Even the software can be integrated with the dev platforms and the IDEs. The research would be going towards the realization of the potential of the LLM for real coding tasks.

Threats

Security, though locally controlled, remains a concern with AI processing sensitive code. Some members of the team will resist changing standard procedures. Bad AI output could propagate errors unless most carefully validated. Most importantly, there is a question about whether existing LLM technology can handle sophisticated PL/SQL insight under our local deployment scenario effectively.

This analysis provides a balanced view of the project's potential, acknowledging that valuable insights will be gained regardless of ultimate feasibility.

3.7 Key Requirements

Based on the user needs and selected use case, we identified key requirements for the AI supported code analysis prototype. These requirements will guide the experimental implementation and provide criteria against which to evaluate its feasibility.

3.7.1 Functional Requirements

ID	Requirement	Description
FR-001	Natural Language Query Processing	The system shall accept natural language queries about PL/SQL code structure and functionality.
FR-002	Code Relationship Analysis	The system shall attempt to identify relationships between different code modules, including function calls and dependencies.
FR-003	Database Impact Analysis	The system shall provide information about database tables accessed or modified by specific code, to the extent that current LLM technology can determine these relationships.
FR-004	Code Explanation Generation	The system shall generate human readable explanations of code functionality based on its analysis.
FR-005	Source Reference Provision	The system shall provide source references for all information included in responses to enable verification.

Table 3.4: Functional Requirements

3.7.2 Non-Functional Requirements

These requirements acknowledge the experimental nature of the project while providing clear criteria for evaluating the feasibility of the approach.

ID	Requirement	Description
NFR-001	Response Time Performance	The prototype shall process queries and generate responses within a reasonable timeframe to allow for interactive testing (target: under 3 minutes).
NFR-002	Security and Local Execution	The system shall execute entirely within the company's secure environment to address security concerns and maintain data sovereignty.
NFR-003	User Interface Simplicity	The system shall provide a simple interface for query submission to facilitate evaluation and testing.
NFR-004	Modular Architecture	The system shall be designed with modular components to facilitate experimentation with different approaches and technologies.
NFR-005	Confidence Indication	The system shall indicate confidence levels or uncertainty in its responses to help users appropriately interpret the results.

Table 3.5: Non-Functional Requirements

3.8 Analysis Findings and Implications

The analysis phase revealed several key insights that will guide the design and implementation of the AI supported code analysis proof of concept.

Key Insights and Design Implications

Although there are different users, they share common needs for understanding of code, database interactions and historical information. A key challenge is keeping the context after chunking the code, especially with large amounts of code. Uncertainty remains about whether LLMs can effectively understand complex PL/SQL code relationships. These findings suggest several design implications: prioritize modular components for independent testing, preserve semantic relationships between code elements, incorporate domain specific knowledge about PL/SQL where possible, balance technical accuracy with readability, and clearly indicate confidence levels in responses.

4. System Design

This chapter presents the design of the AI supported code analysis proof of concept for Oracle PL/SQL code. Building on the requirements and analysis from the previous chapter, it outlines the experimental system architecture, key components, and design decisions that will guide our prototype implementation. The design acknowledges both the potential capabilities and limitations of applying current LLM technology to specialized code comprehension tasks.

4.1 System Architecture Overview

The main challenge in this project is that the system needs to give an LLM access to the PL/SQL codebase and any other information that should be provided. It is of course possible to just give the LLM code snippets in the queries directly but then the system wouldn't be able to retrieve information from other files or parts in the code. Therefore this proof of concept's goal is to build a Retrieval Augmented Generation (RAG) architecture that enhances responses by retrieving relevant code information before generating answers, following established approaches for developing RAG based systems [1].

A RAG represents a system that combines information retrieval with a selected large language model, enabling the system to treat the entire codebase and associated documentation as a queryable knowledge base rather than being constrained to individual code fragments.

Additionally, there are full customization possibilities regarding editing prompts, enhancing security mechanisms and more.

As shown in Figure 4.1, the system consists of five key components: a Frontend interface for query submission, a Query Processor that orchestrates the flow, an Embedding Service that vectorizes text, a Vector Database for storing code representations, and an LLM Response Generator that produces the final answer. This architecture allows for systematic evaluation of LLM capabilities for PL/SQL code comprehension. How the RAG system exactly works will be explained in the next section.

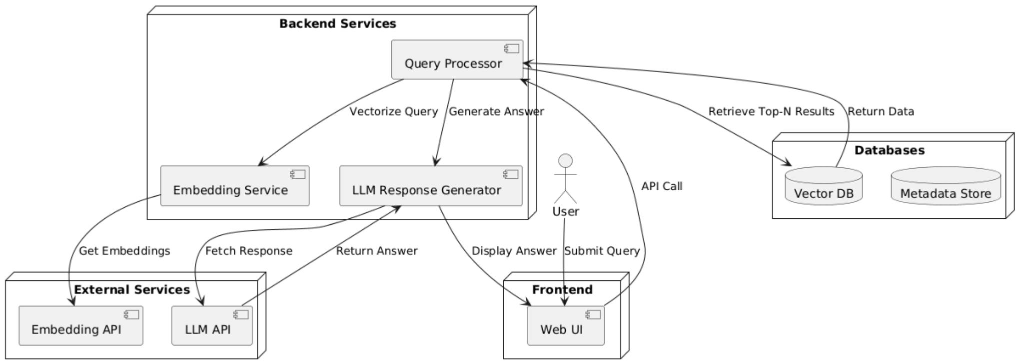

Figure 4.1: Retrieval Augmented Generation (RAG) System Architecture for the Proof of Concept

4.2 Process Flow

The prototype system operates through two main processes: data ingestion and query processing. Each process will be implemented in a way that allows for experimental evaluation of its effectiveness with PL/SQL code.

Data Ingestion Process

Before the system can answer queries, it must first process and index a representative sample of the PL/SQL codebase:

1. **Code Processing**: Selected PL/SQL code samples are processed and divided into meaningful chunks. This step will test different chunking strategies to determine which best preserves the semantic structure needed for dependency analysis.
2. **Embedding Generation**: Each code chunk is converted into a vector representation using a code specific embedding model. This step will evaluate how well current embedding models capture the semantics of specialized PL/SQL code.
3. **Vector Storage**: The generated embeddings, along with metadata about the code chunks, are stored in a vector database for efficient retrieval. This step will test different metadata schemas to determine which best supports dependency queries.
4. **Incremental Updates**: The design includes provisions for handling code changes by only processing modified portions, though full implementation of this capability may extend beyond the initial proof of concept.

Each step in this process presents specific challenges for PL/SQL code that will be systematically evaluated during the prototype implementation.

Query Processing Flow

When a user submits a query, the experimental system processes it through the following steps:

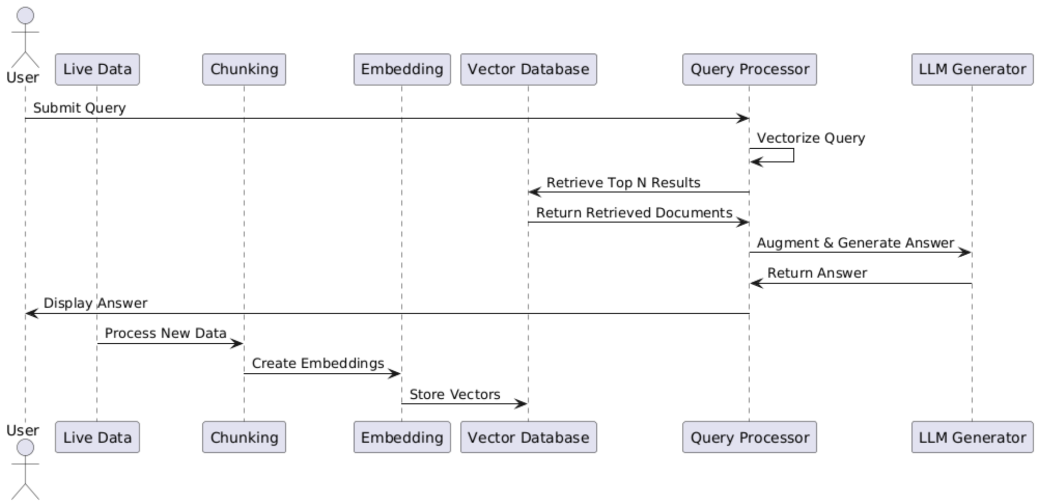

Figure 4.2: Query Processing Sequence Diagram for the Prototype

As illustrated in Figure 4.2, the query processing flow begins when the user submits a natural language query about the PL/SQL code. The Query Processor then vectorizes the query using the Embedding Service, testing how well the embedding model captures the intent of code specific questions. Next, the vectorized query is used to retrieve the most relevant code chunks from the Vector Database, evaluating the effectiveness of similarity search for PL/SQL code. The retrieved code chunks are passed to the LLM Response Generator, which augments the original query with this context. The LLM then attempts to generate a comprehensive response based on the query and the retrieved code context, testing its ability to understand and explain PL/SQL dependencies. Finally, the response is returned to the user through the frontend interface, along with confidence indicators and source references to help evaluate accuracy.

This process will allow us to systematically evaluate whether current LLM technology can effectively understand and communicate complex relationships in PL/SQL code, particularly for the View Function Dependencies use case.

4.3 Component Design and Alternatives

Each component addresses specific requirements identified during analysis. Multiple alternatives were evaluated to determine the most promising approach for the proof of concept. This section focuses on the selections that were made; how they are implemented and work will be explained in Chapter 5.

4.3.1 Code Chunking Component

The purpose of this component is to divide the PL/SQL codebase into segments that preserve a certain structure, critical for retrieving relevant information in the system.

Chunking Strategy Options

For this we evaluated several chunking strategies [2]:

- **Fixed Size Chunking**: Divides code into chunks of fixed token count; computationally cheap and simple but may break logical code units mid function
- **Recursive Chunking**: Splits text hierarchically using separators (paragraphs, sections) to preserve logical structure
- **Semantic Chunking**: Creates chunks based on content meaning and logical PL/SQL structure (procedures, functions, packages)
- **Document Specific Chunking**: Tailored approach that leverages PL/SQL syntax patterns and Oracle specific code organization
- **Agentic Chunking**: Uses AI agents to determine optimal chunk boundaries based on content analysis

Each approach presents different tradeoffs in implementation complexity, retrieval effectiveness, and the preservation logic.

Selected Approach: Semantic Chunking with ZPA Integration

For the implementation, a semantic chunking approach that preserves PL/SQL object boundaries was selected. This works with regex pattern matching, object level boundaries, where each chunk represents a complete logical unit, and additionally a parser is used which provides code quality analysis and metadata enrichment.

While this approach requires more implementation effort than fixed size chunking, it significantly improves dependency analysis quality by preserving semantic integrity. ZPA was chosen for its local execution capability, open source license, and PL/SQL specialization, aligning with security requirements and budget constraints.

4.3.2 Embedding Model Evaluation

The Embedding Service converts code chunks and queries into vector representations for similarity comparison. Model selection proved critical for understanding PL/SQL code semantics and retrieving relevant segments.

Model Selection Process

Initially, GraphCodeBERT was selected based on its superior performance on established code understanding benchmarks [3]. However, during further testing with PL/SQL queries, limitations were observed in retrieving semantically relevant business logic.

Testing Methodology

To objectively compare both models, a testing framework was implemented:

- **Data Collection**: A representative set of PL/SQL code snippets was selected, covering various database objects (tables, procedures, functions, triggers).
- **Embedding Generation**: Each code snippet was processed through both GraphCodeBERT and BGE-M3 to generate vector representations.
- **Similarity Analysis**: Cosine similarity metrics were calculated within and across different code categories.
- **Performance Metrics**: Contrast ratio (ratio of intra category to inter category similarity) served as the primary performance metric.

The testing infrastructure utilized Python with scikit learn, numpy, and Qdrant client libraries, allowing for reproducible evaluation of embedding quality.

Comparative Analysis Results

The quantitative analysis revealed significant differences between the models:

Table 4.1: GraphCodeBERT vs. BGE-M3 Performance Comparison

Metric	**GraphCodeBERT**	**BGE-M3**
Samples tested	22	20
Avg intra-category similarity	0.9186	0.7542
Avg inter-category similarity	0.8681	0.5570
Contrast ratio	1.0582	1.3540

The results demonstrate that BGE-M3 achieved a significantly better contrast ratio (1.35 vs. 1.06), indicating superior ability to differentiate between code categories, which is critical for retrieval tasks.

GraphCodeBERT showed very high similarity scores overall (0.80-0.95 range), suggesting its embeddings cluster tightly together, potentially making discrimination between different code types more difficult. In contrast, BGE-M3 produced more distinct embeddings with wider similarity ranges (minimum similarity of 0.39), creating clearer separation between different code segments.

Additionaly the model BGE-M3 offers better characteristics to the project than the GraphCodeBERT model:

Table 4.2: Embedding Model Evaluation Criteria

Criteria	**GraphCodeBERT**	**BGE-M3**
Code Structure Understanding	Strong data flow sensitivity; optimized for function-level code relationships	More generalized text understanding with better performance on diverse text granularities
Language Support	Primarily trained on 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go); limited PL/SQL exposure	Supports 100+ languages with better generalization to less common languages like PL/SQL
Context Length	Limited context window (512 tokens) restricting analysis of longer procedures	Supports up to 8192 tokens, allowing complete procedure analysis
Semantic Search Performance	Strong for explicit code references but weaker for implicit semantic relationships	Higher performance on MTEB benchmarks (Massive Text Embedding Benchmark) for semantic search tasks
Implementation Complexity	Requires specialized data flow graph extraction for optimal performance	Simpler integration with standard text processing pipelines
Resource Requirements	Lower memory footprint but requires specialized preprocessing	Higher memory requirements but more straightforward processing pipeline
Vector Differentiation	High overall similarity (low contrast ratio of 1.06)	Better category separation (higher contrast ratio of 1.35)

Practical Validation

Through the comparative analysis by embedding test PL/SQL code and going through the characteristics of each model, BGE-M3 was chosen as the better fitting model for this project.

4.3.3 Vector Database Evaluation

The Vector Database stores and indexes code embeddings for efficient similarity search [4].

Vector Database Alternatives and Selection

We considered three primary options:

- **Qdrant**: Dedicated vector database with advanced filtering and search capabilities [5]
- **Chroma**: Simpler, lightweight database designed for ease of use
- **FAISS with SQLite**: Custom solution combining Facebook AI Similarity Search with SQLite for metadata

We selected Qdrant as its filtering capabilities are well suited to handling PL/SQL code with multiple schemas and object types. While the prototype will use a limited code sample, Qdrant's scalability provides a path forward if successful. Its query capabilities should allow for complex filtering needed for dependency analysis.

4.3.4 LLM Component Evaluation

This component produces human readable answers based on retrieved code chunks and the original query, presenting particular challenges for understanding specialized PL/SQL syntax.

LLM Alternatives and Evaluation Criteria

To select an LLM model, multiple models were evaluated which include CodeLlama, Mistral and Qwen. The evaluation criteria include code comprehension, context utilization, resource efficiency, and accuracy indicators. The models were tested by connecting them to the RAG system, running different queries, and then looking for the best balance of these criteria for PL/SQL code comprehension.

Evaluation Framework: While formal benchmarks like HumanEval and MBPP focus primarily on code generation tasks, our evaluation prioritized code understanding and explanation capabilities, which are more relevant for legacy code analysis. The assessment included qualitative testing with representative PL/SQL queries to evaluate:

- **Business Logic Comprehension**: Understanding procedural workflows and business rules
- **Cross Package Relationship Analysis**: Identifying dependencies between different code modules
- **Context Utilization**: Effectively using retrieved code chunks to provide accurate responses
- **PL/SQL Syntax Understanding**: Handling Oracle specific constructs and patterns

Selected Model: Qwen3-32B

For the prototype implementation, Qwen3-32B was selected based on, code understanding, resource balancing, and context handling.

The model is specifically fine tuned for code comprehension tasks [6], has 32B parameters which provides complexity without excessive resource requirements, and has effective processing of formatted code chunks and metadata.

The model currently runs on external GPU infrastructure for development, with plans to transition to local deployment for production security requirements.

4.4 Implementation Considerations

Security considerations will influence the implementation. All components will run within the company's secure environment with no code transmitted externally. The prototype will use a controlled subset of code rather than the full production codebase, with all queries and responses logged for audit purposes.

5. Implementation

In this chapter the implementation of the components will be explained, what discoveries were made in the process and what results the implementations provided. The plan was to create one component at a time and test them on their own. This approach allowed the system to be as modular as possible and therefore create an interchangeable system. Meaning if a component's technology doesn't work as expected or if there is a need to update a model in the future, this could be done without affecting any other parts of the system.

5.1 Chunking and Parsing

First the chunking and parsing of the PL/SQL code will be explained as this is the first step taken, processing the data available. This also lays the foundation for how the system continues working with the code in the other components.

ZPA Integration and Code Quality Analysis

In order to enrich the data, an external tool was used which analyzes the code quality, is designed for PL/SQL code and runs locally. The tool is called 'ZPA Runner' 5.1 and first the location of the tool needs to be specified. Afterwards it runs on the directory and produces a JSON file with issues like:

- "Line 45: Variable never used"
- "Line 67: SQL injection risk"

```
    # Run external tool to analyze code quality
    zpa_runner = ZpaRunner(zpa_cli_path="path/to/zpa")
    zpa_runner.run_analysis(
        source_dir="my_plsql_code/",
        output_file="analysis_results.json"
    )
```

Listing 5.1: Running ZPA analysis

The reason the ZPA runner specifically was chosen is that it runs fully locally, and therefore complying with the security goals. It's also an open source tool meaning it doesn't cause any licensing costs. Lastly there are not many parsing tools available for PL/SQL so there weren't many other options.

Later this additional information can be used to help with finding issues in the codebase. But now the issues and other information need to be added to the correct file. For this the parse output function 5.2 is used.

```
# Read the analysis results
parser = ZpaOutputParser(source_dir="my_plsql_code/")
processed_data = parser.parse_output("analysis_results.json")

# Result: List of files with their issues
processed_data = [
    {
        'path': 'customer.sql',
        'content': 'CREATE FUNCTION get_customer...',
        'issues': [
            {'line': 10, 'severity': 'MAJOR', 'message': 'Unused variable'},
            {'line': 25, 'severity': 'MINOR', 'message': 'Missing comment'}
        ]
    }
]
```

Listing 5.2: Parsing ZPA output

First thing that is done is reading the previously generated JSON file, taking the new scattered code issue and then grouping them by file. Afterwards the original source code of said file is loaded and combined with the belonging issue group. At this point there is a data structure that includes the code content and the associated issues. Figure 5.3 shows an example of processed data output and as seen there is now a cleaner structure which then will be returned.

```
# Read the analysis results
parser = ZpaOutputParser(source_dir="oracle_samples/")
processed_data = parser.parse_output("analysis_results.json")

# Result: List of files with their issues
processed_data = [
    {
        'path': 'oracle_samples/01_tables.sql',
        'content': 'CREATE TABLE employees (\n employee_id NUMBER(6) PRIMARY KEY,\n first_name VARCHAR2(20),\n  ...',
        'issues': [
            {
```

```
                    'line': 84,
                    'severity': 'MINOR',
                    'message': 'Use VARCHAR2 instead of CHAR.',
                    'ruleId': 'CharacterDatatypeUsage'
                }
            ]
        }
    ]
```

Listing 5.3: processed ZPA data

Semantic Chunking Implementation Since the goal is to save the data as optimally as possible the code still needs to be separated by some category. For this the decision was made to chunk based on PL/SQL object types, where each database object (function, procedure, package, trigger) forms a natural boundary for chunking.

```
    # Look for different types of PL/SQL objects
    patterns = {
        'FUNCTION': r'CREATE\s+FUNCTION\s+(\w+)',
        'PROCEDURE': r'CREATE\s+PROCEDURE\s+(\w+)',
        'PACKAGE': r'CREATE\s+PACKAGE\s+(\w+)',
        'TRIGGER': r'CREATE\s+TRIGGER\s+(\w+)'
    }

    # For each line, check if it starts a new block
    for line in file_lines:
        if "CREATE FUNCTION get_customer" in line:
            # Start new function block
            current_block = "FUNCTION_get_customer"
        elif "END get_customer;" in line:
            # End current block, save it as a chunk
            save_chunk(current_block)
```

Listing 5.4: How chunker identifies code blocks

In 5.4 it can be seen how the split by patterns works. Certain keywords are looked for and based on that fitting chunks can be created.

Processing Results and Metrics

After the data went through all steps structured smaller chunks are produced, which are ready to be embedded and stored.

In 5.5 an example can be seen of how the data looks initially and how it looks in the end.

```
    # Input: One large file
    large_file = """
    CREATE PACKAGE customer_pkg AS
      FUNCTION get_name(id NUMBER) RETURN VARCHAR2;
      PROCEDURE update_address(id NUMBER, addr VARCHAR2);
    END;

    CREATE PACKAGE BODY customer_pkg AS
      FUNCTION get_name(id NUMBER) RETURN VARCHAR2 IS
        -- 200 lines of code here
      END;

      PROCEDURE update_address(id NUMBER, addr VARCHAR2) IS
        -- 300 lines of code here
      END;
    END;
    """

    # Output: Multiple smaller chunks
    chunks = [
        {
            'code': 'CREATE PACKAGE customer_pkg AS...',
            'block_type': 'PACKAGE',
            'object_name': 'customer_pkg',
            'issues': []
        },
        {
            'code': 'FUNCTION get_name(id NUMBER)...',
            'block_type': 'FUNCTION',
            'object_name': 'get_name',
            'issues': [{'line': 5, 'message': 'Unused
               variable'}]
        }
    ]
```

Listing 5.5: Chunker input and output example

5.2 Vector Storage and Retrieval

After everything is chunked, it goes straight to the embedding service. The previous steps helped to create structured, logical code segments with their associated metadata and quality issues. Now these text based code chunks need to be converted into numerical vectors that the AI system can understand and compare mathematically.

The embedding service transforms each code chunk into a 1024-dimensional vector representation, enabling semantic similarity search. This means that functionally similar PL/SQL code will produce similar vector representations, regardless of differences in variable names or minor syntactic variations. For example, two functions that both handle customer address updates will have vectors pointing in similar directions in the mathematical space, even if one uses different variable names or slightly different SQL syntax.

BGE-M3 Embedding Implementation

The goal of the Embedding Service is to take code or text and transform it into a vector dimension.

```
# Input: PL/SQL code as text
code_text = """
FUNCTION get_customer_name(customer_id NUMBER) RETURN VARCHAR2 IS
    customer_name VARCHAR2(100);
BEGIN
    SELECT name INTO customer_name
    FROM customers
    WHERE id = customer_id;
    RETURN customer_name;
END;
"""

# Output: A list of 1024 numbers representing this code
embedding = [0.123, -0.456, 0.789, ..., 0.321]  # 1024 numbers total
```

Listing 5.6: Embedding example

5.6 shows basic PL/SQL code at the top, which should be given to the embedding service. After the service processed the said code, the goal is to get the 'embedding' result which represents a vector space. To achieve this multiple steps have to be taken. First the wished embedding model is loaded as seen in 5.7

```
from sentence_transformers import SentenceTransformer

# Load a pre-trained model that understands code
model = SentenceTransformer("BAAI/bge-m3")
```

Listing 5.7: Loading the embedding model

After the model is loaded and ready, the code can already start being converted into vectors as shown in 5.8.

```python
def embed_code(self, code_text):
    """Convert code text to numerical vector"""

    # Generate embedding using pre-trained model
    embedding = self.model.encode(code_text)

    # Convert numpy array to list for JSON serialization
    return embedding.tolist()
```

Listing 5.8: Basic Embedding Function

Additionally, not only one code snippet should be embedded but later on there will be multiple chunks which need to be embedded separately.

```python
    # We have many pieces of code
    code_chunks = [
        "FUNCTION get_customer_name...",
        "PROCEDURE update_address...",
        "FUNCTION calculate_total..."
    ]

    # Convert each piece to numbers
    embeddings = []
    for code in code_chunks:
        numbers = embed_code(code)
        embeddings.append(numbers)
```

Listing 5.9: Embedding chunks

In 5.9 an example list of PL/SQL code chunks can be seen and each of these is embedded on their own and then added to a list, which will later be returned by the function. This process is done in batches in the code, meaning multiple chunks are processed together at once to improve performance and reduce the computational overhead of loading the model repeatedly for each individual chunk.

Now that the code is properly embedded it needs to be saved somewhere.

Qdrant Database Configuration The Qdrant Database enables efficient storage of large quantities of embeddings and searching those. Instead of going through thousands of lines manually, searching can be done via semantic search and similar naming.

This works by creating a collection first in the database (5.10).

```
    def create_collection(self, vector_size: int = 1024):
    # Create a "folder" in the database for our code
    self.client.create_collection(
        collection_name="plsql_code",
        vectors_config=VectorParams(size=1024, distance=
            Distance.COSINE)
    )
```

Listing 5.10: Creating qdrant collection

Here first a vector size of 1024 is set and the distance type is set to 'cosine'.

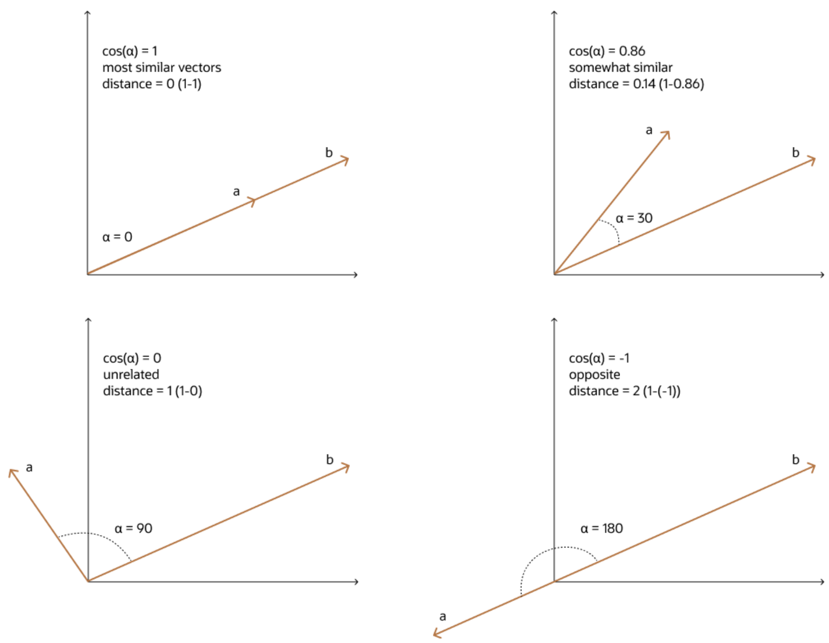

Figure 5.1: Cosine similarity visualization showing how vector angles determine similarity

The cosine distance measures the angle between two vectors, instead of the absolute distance. If the angle is smaller or if the vectors point in a similar direction, it means the vectors are more similar, as illustrated in Figure 5.1 [7].

After the collection is set and created, the code chunks can already be stored.

```
def store_chunks(self, chunks):
points = []
for chunk in chunks:
    # Take the code information
    vector = chunk['vector']        # The 1024 numbers
    metadata = chunk['file']        # File name, function name, etc.

    # Create a storage point
    point = PointStruct(
        id="unique-id-123",
        vector=vector,              # The numbers for searching
        payload=metadata            # Extra information about the code
    )
    points.append(point)

# Save all points to the database
self.client.upsert(collection_name="plsql_code", points=points)
```

Listing 5.11: Storing Chunks in collection

In 5.11 it starts by pulling the vector and the needed metadata, which later helps to have more accurate searches. But this will be explained in depth later in the report. Afterwards the Point can be created which is the central entity that Qdrant uses (one collection consists of multiple points) [8]. In that Point the vector, metadata and a unique ID which is generated using uuid64 are stored. Lastly all of this is added to the collection.

Hybrid Search Implementation

This setup enables searching the vectors using normal search 5.12 and hybrid search 5.13.

```
def search(self, query_vector, limit=5):
# Find code chunks with similar numbers
results = self.client.search(
    collection_name="plsql_code",
    query_vector=query_vector,      # What we're looking for
    limit=5                         # Give me top 5 matches
)
```

```

    return results
```

Listing 5.12: standard Qdrant search function

In the normal search function 5.12 simply the collection that should be searched is given, the vectorized query and how many results should be received. In this case the function will then return the top 5 results based on semantic search.

```
    def hybrid_search(self, query_vector, query_text):
    # If user asks "show me functions"
    if "function" in query_text.lower():
        # Search only in function code
        results = self.client.search(
            query_vector=query_vector,
            query_filter=Filter(
                must=[FieldCondition(key="block_type", match
                   ="FUNCTION")]
            )
        )
    else:
        # Search everywhere
        results = self.search(query_vector)
```

Listing 5.13: hybrid Qdrant search function

In the hybrid search function 5.13 the filter from Qdrant can now be utilized. The filter then only gives the chunks that contain 'Function' and if no result is received, it falls back to the normal search function.

Now embeddings for code chunks can be created, stored in Qdrant and search functions can be run on the database.

5.3 Query Processing and Response Generation

The last component and the most crucial now is processing the users' queries and generating proper responses. For that multiple steps are needed. The user's query needs to be embedded, fitting code chunks need to be retrieved out of the database, this package needs to be sent to the fitting large language model and a response needs to be received from it.

Query Processing Pipeline

Processing the query from the user is one of the easier steps. In 5.14 first the embedding for it is generated using the previously shown embedding service. Based on the received vectors the "top k" code chunks are retrieved, where "top k" is set to 15. Meaning the top 15 most fitting chunks are received which gives a broader information output. But to ensure quality afterwards all results are filtered out, which are under the set similarity threshold, which is set to 0.6. Lastly these results are returned by the function.

```
    def process_query(self, query: str) -> List[Dict[str, Any]]:
        # Convert user question to numbers
        query_embedding = self.embedding_service.generate_query_embedding(query)

        # Search for similar code chunks using hybrid search
        results = self.storage.hybrid_search(query_embedding, query, limit=self.top_k)

        # Filter by similarity threshold (only good matches)
        filtered_results = [
            result for result in results
            if result.get('score', 0) >= self.similarity_threshold
        ]

        return filtered_results
```

Listing 5.14: Query processing workflow

Context Formatting and Prompt Engineering Before the final prompt can be passed on it needs to be enriched with instructions and tailored to a format which the LLM (Large Language Model) can handle better. Here the user's prompt is also combined with all the relevant code chunks and AI hallucination is tried to be prevented, which describes when AI generates information that is incorrect but sounds plausible in the first moment.

The challenge is that the retrieved code chunks are in raw database format with metadata, but the AI needs clean, structured text to understand and work with the code effectively.

```
    def build_prompt(self, query: str, chunks: List[Dict[str, Any]]) -> Dict[str, Any]:
        # Format code chunks into readable context
        context = self._format_context(chunks)

        # Create system prompt with instructions
        system_prompt = """You are an Oracle PL/SQL expert assistant.
```

```
        Use the following code context to answer the user's question.

        CODE CONTEXT:
        {context}

        IMPORTANT: Only use the exact code provided above.""".format(context=context)

        return {"system": system_prompt, "prompt": query}
```

Listing 5.15: Building LLM prompts with context

In 5.15 first the chunk is sent to the "format context" function 5.16 which takes the raw chunks from the database and transforms them into a clean, readable format. In more detail it numbers each code snippet (Snippet 1,2,3...), creates headers showing the function name, type and file, wraps the actual code in proper formatting blocks, and lists any quality issues found below each chunk.

This structured formatting is crucial because the AI needs clear boundaries to understand where one piece of code ends and another begins.

In the next step, a "system prompt" is created, which represents job instructions for the AI. This tells the AI it should act as a PL/SQL expert and, most importantly, only use the exact code that is provided. Then the formatted code context is embedded and everything is combined with the user's original question.

The result is a complete prompt that gives the AI both the context (relevant code) and clear instructions on how to use it.

````
    def _format_context(self, chunks):
        context = ""
        for i, chunk in enumerate(chunks, 1):
            # Add chunk header with metadata
            context += f"### Snippet {i}: {chunk['block_name']} ({chunk['block_type']})\n"
            context += f"File: {os.path.basename(chunk['file_path'])}\n"

            # Add the actual code
            context += f"```sql\n{chunk['code']}\n```\n"

            # Add any code quality issues
            if chunk['issues']:
                context += "Issues found:\n"
                for issue in chunk['issues']:
````

```
                    context += f"- Line {issue['line']}: {issue['message']} ({issue['severity']})\n"

                context += "\n"

            return context
```

Listing 5.16: Formatting code for AI understanding

5.4 Complete System Integration

In the last step all these functions and classes are called, in order to send the data through the pipeline.

End-to-End Processing The code in 5.17 displays the top level calls.

First the query is sent to be fully processed then the prompt is built with the help of the prompt builder and lastly everything is sent to the chosen LLM and the response is retrieved.

```
def process_query(self, query: str) -> str:
    # Step 1: Find relevant code chunks
    results = self.query_processor.process_query(query)

    # Step 2: Build prompt with context
    prompt = self.prompt_builder.build_prompt(query, results)

    # Step 3: Generate AI response
    response = self.llm_client.generate(
        model="Qwen/Qwen3-32B",
        prompt=prompt,
        temperature=0.1  # Low temperature for factual responses
    )

    return response
```

Listing 5.17: Complete RAG pipeline

The `llm_client` represents a connection service to the large language model. Currently it is connected to an external GPU which runs the model Qwen-32B. Of course later no external services will be used but for testing and implementation it is needed as the laptop where everything is being developed on doesn't have the necessary hardware.

In the implementation and testing process a few different models were tested but currently Qwen-32B is used as it's specifically fine tuned for code understanding. The "32B" stands for the amount of parameters it uses, in this case 32 billion, making it capable of understanding complex code relationships while still having a manageable size compared to larger models.

The temperature parameter (0.1) controls the randomness of the AI's responses. A low temperature like 0.1 makes the model more factual, while higher values (0.7-1.0) would make responses more creative but potentially less accurate. For code analysis, factual accuracy is prioritized over creativity.

User Interface Implementation To make the interaction with the system more accessible and visually appealing the framework "GradioUI" is being used, which is Python based and made for LLM interactions. The reason why GradioUI was chosen is because it automatically generates clean and professional interfaces with minimal code.

How the Web Interface Works

The GradioUI framework creates a locally hosted web server that developers can access through their browser. The interface consists of two main components:

- **Input Box:** Where users type their natural language questions about the PL/SQL code
- **Response Area:** Displays the AI generated answers with formatted code snippets and explanations

When a user submits a query, GradioUI automatically calls the `process_query` function, passes the user input through the complete RAG pipeline, and displays the response in real time.

System Demonstration To illustrate the entire process from user input to final response, here is a step by step walkthrough of how the system handles a real query:

```
# Step 1: User submits query through web interface
user_input = "what packages are used in employee processing"

# Step 2: System processes query
print("Converting query to vector...")
query_embedding = embedding_service.generate_query_embedding(
    user_input)
```

```
# Output: [0.123, -0.456, 0.789, ...] (1024 numbers)

print("Searching for relevant code...")
search_results = storage.hybrid_search(query_embedding,
    user_input, limit=5)
# Output: 3 relevant code chunks found with scores: 0.87,
    0.82, 0.76

print("Generating AI response...")
final_response = llm_client.generate(
    prompt=formatted_context + user_input,
    temperature=0.1
)

# Step 3: Response displayed in web interface
print("Final Response:")
print("""
The following packages are used in employee processing:
- validation_pkg: Used for validating input data
- error_pkg: Used for handling errors
- employee_pkg: Used for managing employee records
""")
```

Listing 5.18: Complete system flow example

This example shows how a simple question triggers the entire pipeline from vector conversion, to database search, then context formatting, followed by AI generation and lastly a user friendly response.

6. Results and Discoveries

This will explain the results from the implementation and the discoveries that were made. This includes the capabilities of the model and metrics that were collected. This was done by feeding the system with a sample PL/SQL base and running queries on the system.

6.1 Evaluation Methodology

Test Dataset

The evaluation used a representative PL/SQL codebase which was generated by AI. It still maintains logic and functionality and is supposed to replicate a normal Oracle Forms application. It consists of 11 files in total which include:

- Database schemas and table definitions
- PL/SQL packages (validation_pkg, error_pkg, employee_pkg)
- Stored procedures and functions
- Database triggers
- Typical enterprise patterns found in legacy systems

Query Categories

After the codebase was embedded and stored successfully in the database, a set of queries were prompted, as shown in appendix A.3. The goal was to test the understanding of the procedure workflows, identifying package relationships, extracting business rule implementations and receiving simple information like table definitions. To showcase the capabilities and current limitations of the system, only a few queries and their responses will be shown and the responses will be shorten. The full responses can be found in Appendix A.

6.2 System Performance Results

Successful Query Processing

Process Flow Analysis Query: "What happens when a new hire is processed"

System Response: The process_new_hire procedure handles the onboarding of new employees with validation and error handling:

1. **Input Parameters**: Accepts employee details like name, email, phone, job ID, department ID, salary, and optional manager ID.
2. **Validation Checks**:
   - Job ID validation using validation_pkg.is_valid_job
   - Department ID validation using validation_pkg.is_valid_department
   - Salary validation (likely checking ranges for the specific job)
3. **Employee Creation**: If validations pass, calls employee_pkg.create_employee to insert the new employee record into the database, generating a new employee ID.
4. **Document Creation**: Creates an initial document request entry in the employee_documents table for onboarding documentation.
5. **Output**: Displays success message and the newly assigned employee ID.

The procedure uses centralized error handling through error_pkg.handle_error for validation failures and unexpected exceptions, ensuring consistent error management across the application.

Analysis: The system response demonstrates comprehensive understanding of the procedure's functionality, going beyond simply listing steps to explaining the architectural design patterns. This query showcases advantages over standard IDEs like SQL Developer or PL/SQL Developer, which only provide code navigation without semantic understanding.

In a workflow without an AI assistant, the process would include:

1. Locate the process_new_hire procedure in the package body
2. Read through all the code line by line
3. Mentally extract and organize the key operations
4. Potentially follow additional function calls to understand the complete workflow

The RAG system accomplished this with a simple natural language query, delivering an organized summary and providing semantic meaning with it. This addresses the challenge explained in Chapter 1, in which developers like Sarah need to manually trace the code to understand its pieces. The system not only identified the technical steps but also recognized their business purpose (e.g., "onboarding"), but also identified architectural patterns like modular design with separate packages for validation, error handling, and employee management.

Schema Understanding Edge Case

Query: "What does table xy contain?"

System Response: The provided PL/SQL code defines two tables, `employee_documents` and `job_history`, which are part of a database schema likely related to employee management. Below is a detailed explanation of each table's structure, functionality, and key details:

employee_documents Table

Purpose: Stores documents uploaded by or related to employees (e.g., resumes, ID copies, contracts).

Structure:

- `document_id`: Primary key (unique identifier for each document)
- `employee_id`: Foreign key referencing the `employees` table. Ensures documents are linked to valid employees
- `document_type`: Describes the document's category (e.g., "Resume", "Passport Copy"). Cannot be null

... (additional fields as needed)

job_history Table

Purpose: Records an employee's past job roles, departments, and employment dates.

... (structure details as needed)

Analysis: This response demonstrates a limitation in the current implementation. When asked about a non existent table ("xy"), the system retrieved the closest matching content (information about `employee_documents` and `job_history` tables) rather than correctly identifying that the table doesn't exist.

This behavior reveals an important issue: the system currently lacks a verification mechanism to confirm the existence of entities before providing information about them. The RAG system retrieved semantically similar content based on vector similarity, but without a validation step to verify that "table xy" actually exists in the codebase.

This highlights a key area for improvement: implementing entity verification to ensure the system can distinguish between requests for existing versus non existent database objects. Without this capability, users might receive seemingly authoritative information about non existent components, potentially leading to confusion or errors in development work.

A more appropriate response would have been to inform the user that no table named "xy" exists in the codebase and offer information about available tables instead.

Additionally when asked "What tables exist in this database" the system only retreives the information of the first chunk it can find. the displayed infos are correct but information is missing.

System Response: The database contains multiple interconnected tables for employee management:

employees Table

Purpose: Stores core employee information and their current job/department assignments.

Key Columns:

- `employee_id` (Primary Key): Unique identifier for each employee
- `first_name`, `last_name`: Employee's name
- `email` (Unique, Not Null): Unique email address
- `hire_date`, `salary`, `job_id`, `manager_id`, `department_id`

departments Table

Purpose: Stores department information and their managers.

Key Columns:

- `department_id` (Primary Key): Unique identifier for each department
- `department_name` (Not Null): Name of the department
- `manager_id`: Links to the department's manager
- `location_id`: Links to a physical location

job_history Table

Purpose: Tracks an employee's past job roles and department assignments.

Key Columns:

- Composite Primary Key: (`employee_id`, `start_date`)

- `end_date`, `job_id`, `department_id`

Key Relationships:

- Self-referencing relationship in `employees` via `manager_id`
- `employees` linked to `departments` and `jobs` (referenced but not shown)
- `job_history` tracks historical employment records

The system only retrieved information from the first content chunk, missing other tables referenced in the schema (like `jobs` and `locations`).

The response does correctly identify the table structures and their relationships, showing understanding of database design concepts like primary keys, foreign keys, and constraints. It also recognizes the business purpose of each table within the employee management system.

7. Conclusion

This report systematically researched how to efficiently implement a RAG system for PL/SQL code analysis while using resource efficient components. The proof of concept investigated whether current LLM technology can handle PL/SQL code while operating under strict security and computational constraints.

For this purpose, a RAG system was created which showed that the results not only depend on the chosen LLM but heavily rely on the underlying components. The research answered what the best approach is to embed code and which model to choose. Through controlled experimentation with embedding models, it was identified that BGE-M3 significantly outperforms GraphCodeBERT for PL/SQL code retrieval tasks, achieving a 28% improvement in category differentiation (contrast ratio 1.35 vs 1.06).

Afterwards, methods were discovered to preprocess the code to receive better embedding results. To preserve logical code boundaries, semantic chunking was implemented, and to discover issues in the code, the ZPA parser was integrated. To search Qdrant, which is the chosen vector database, a hybrid search approach was implemented, combining vector similarity with metadata filtering to provide optimal retrieval accuracy.

The implemented architecture successfully demonstrates that AI assisted code analysis is able to operate within constrained computational environments. The locally deployed approach using Qwen3-32B achieved interactive responses in under 3 minutes while maintaining full data sovereignty. Critical resource optimization strategies were identified: semantic chunking reduces vector storage requirements while preserving code semantics, BGE-M3's 8192-token context window enables complete procedure analysis without fragmentation, and Qdrant's filtering capabilities minimize computational overhead during retrieval operations.

Despite the successful retrieval of helpful and rich responses, there are still limitations and issues in the current system. The LLM only takes the selected chunks as context at the moment and retrieves the next best chunk when looking for a non existing table.

By running the test queries with a success rate of over 60%, the first acceptance criterion was achieved (SC-001). The project also generated further insights in applying LLMs for legacy code comprehension, leading to the fulfillment of SC-003. The only criterion not fully met yet is SC-002, because the current system hasn't been tested to an extent yet where clear limitations regarding LLM technology processing PL/SQL code can be definitively established. Therefore, the project hasn't fully achieved all acceptance criteria as of June 1, 2025.

7.1 Future work

Therefore, the next steps planned for the project include research on how to provide the LLM with more context while assessing the resource efficiency that comes with it. Also, there will be more extensive testing with larger codebases and a more diverse set of information (code, documentation, tickets) and harder testing of the LLM's limits. This will provide a clearer outlook for future use of this system.

Key areas for development include implementing entity verification mechanisms to address the "table xy" limitation discovered during testing, where the system provided information about non existent database objects. Additionally, the similarity threshold of 0.6 requires optimization through systematic testing to balance retrieval accuracy with response completeness.

The current semantic chunking approach, while preserving PL/SQL object boundaries effectively, could be enhanced to maintain cross package relationships that span multiple chunks. Integration of the ZPA parser's code quality analysis into the response generation process would provide developers with immediate insights about potential issues in retrieved code segments.

Based on the results obtained, the system will be adapted and development will continue with focus on these identified improvement areas.

A. System Query Responses

A.1 Schema Analysis Responses

A.1.1 Query: "What tables exist in this database"

Full System Response

The provided PL/SQL code defines three tables (‘job_history‘, ‘employees‘, and ‘departments‘) that form a relational database structure for managing employee, department, and job history data. Below is a breakdown of their functionality, relationships, and key details:

1. ‘employees‘ Table Purpose: Stores core employee information and their current job/department assignments.
Key Columns:

- ‘employee_id‘ (Primary Key): Unique identifier for each employee.
- ‘first_name‘, ‘last_name‘: Employee’s name.
- ‘email‘ (Unique, Not Null): Unique email address for identification.
- ‘hire_date‘ (Not Null): Date the employee was hired.
- ‘job_id‘ (Foreign Key → ‘jobs.job_id‘): Links to the employee’s current job role.
- ‘salary‘ (Check Constraint: ‘salary > 0‘): Ensures valid positive salary.
- ‘manager_id‘ (Foreign Key → ‘employees.employee_id‘): Self-referencing relationship.
- ‘department_id‘ (Foreign Key → ‘departments.department_id‘): Links to department.

Key Functionality:

- Tracks employee details and their current job/department.
- Enforces data integrity via constraints (e.g., salary must be positive, unique emails).
- Supports hierarchical management structures via ‘manager_id‘.

2. ‘departments‘ Table Purpose: Stores department information and their managers.
Key Columns:

- ‘department_id‘ (Primary Key): Unique identifier for each department.
- ‘department_name‘ (Not Null): Name of the department.
- ‘manager_id‘ (Foreign Key → ‘employees.employee_id‘): Links to the manager.
- ‘location_id‘ (Foreign Key → ‘locations.location_id‘): Links to a location.

Key Functionality:

- Manages departmental structure and assigns managers.
- Connects departments to employees and locations.

3. ‘job_history‘ Table Purpose: Tracks an employee’s past job roles and department assignments.
Key Columns:

- ‘employee_id‘ (Foreign Key → ‘employees.employee_id‘): Links to the employee.
- ‘start_date‘: Start date of the job/department assignment.
- ‘end_date‘ (Not Null): End date of the job/department assignment.
- ‘job_id‘ (Foreign Key → ‘jobs.job_id‘): Links to the previous job role.
- ‘department_id‘ (Foreign Key → ‘departments.department_id‘): Links to department.

Key Functionality:

- Maintains historical records of employee job/department changes.
- Uses a **composite primary key** (‘employee_id‘, ‘start_date‘).

Relationships and Constraints

1. **Foreign Key Relationships**:
   - ‘employees.job_id‘ → ‘jobs.job_id‘ (not shown).
   - ‘employees.department_id‘ → ‘departments.department_id‘.
   - ‘employees.manager_id‘ → ‘employees.employee_id‘ (self-referencing).
   - ‘job_history.employee_id‘ → ‘employees.employee_id‘.
   - ‘job_history.job_id‘ → ‘jobs.job_id‘ (not shown).
   - ‘job_history.department_id‘ → ‘departments.department_id‘.
2. **Constraints**:
   - **Primary Keys**: Ensure uniqueness for records.
   - **Foreign Keys**: Enforce referential integrity.
   - **Check Constraints**: ‘salary > 0‘ ensures valid salary values.
   - **Unique Constraints**: ‘email‘ must be unique for each employee.

Summary This schema is designed to manage employee data, departmental structure, and job history with strong data integrity. Key features include:

- Self-referencing hierarchical relationships (managers).
- Historical tracking via ‘job_history‘.
- Constraints to ensure valid and consistent data.

A.1.2 Query: "What does table xy contain?"

Full System Response

The provided PL/SQL code defines two tables, 'employee_documents' and 'job_history', which are part of a database schema likely related to employee management. Below is a detailed explanation of each table's structure, functionality, and key details:

1. 'employee_documents' Table Purpose: Stores documents uploaded by or related to employees (e.g., resumes, ID copies, contracts). **Structure**:

- **'document_id'**: Primary key (unique identifier for each document).
- **'employee_id'**: Foreign key referencing the 'employees' table.
- **'document_type'**: Describes the document's category. Cannot be null.
- **'filename'**: Name of the uploaded file (up to 100 characters).
- **'file_data'**: Binary Large Object (BLOB) storing the actual file content.
- **'upload_date'**: Date when the document was uploaded.
- **'notes'**: Optional notes or metadata about the document.

Key Functionality:

- Tracks employee-related documents directly in the database using 'BLOB' storage.
- Enforces referential integrity via the 'employee_id' foreign key.
- Ensures every document has a type and is associated with a valid employee.

2. 'job_history' Table Purpose: Records an employee's past job roles, departments, and employment dates.
Structure:

- **'employee_id'**: Foreign key referencing the 'employees' table.
- **'start_date'**: Start date of the job role.
- **'end_date'**: End date of the job role (cannot be null).
- **'job_id'**: Foreign key referencing the 'jobs' table.
- **'department_id'**: Foreign key referencing the 'departments' table.
- **Primary Key**: Composite key of 'employee_id' and 'start_date'.

Key Functionality:

- Tracks career progression by recording past jobs, departments, and employment durations.
- Enforces referential integrity via foreign keys.
- Prevents duplicate entries using the composite primary key.

Summary

- **'employee_documents'** manages employee-related files with metadata and binary storage.
- **'job_history'** tracks an employee's past roles, departments, and employment dates.

These tables are critical for maintaining a comprehensive HR database, supporting document management and career history tracking.

A.2 Process Flow Analysis Responses

A.2.1 Query: "What happens when a new hire is processed"

Full System Response

The provided PL/SQL code defines procedures and packages for processing new employee hires and managing employee data. Here's a breakdown of its key functionality and important details:

1. ‘process_new_hire‘ Procedure Purpose: Handles the onboarding of new employees with validation and error handling.
Key Steps:

- **Input Parameters**: Accepts employee details like name, email, phone, job ID, department ID, salary, and optional manager ID.
- **Validation Checks**:
  - **Job ID**: Validates against ‘validation_pkg.is_valid_job‘.
  - **Department ID**: Validates using ‘validation_pkg.is_valid_department‘.
  - **Phone/Email**: Likely includes additional validations.
- **Error Handling**: Uses a centralized ‘error_pkg‘ to log errors.
- **Employee Creation**: After validation, calls ‘employee_pkg.create_employee‘.

2. ‘employee_pkg‘ Package Purpose: Centralized logic for employee management.
Key Components:

- **‘get_next_employee_id‘ Function**: Generates the next available employee ID.
- **‘create_employee‘ Procedure**:
  - Inserts a new employee record into the database.
  - Parameters include personal details, job, salary, department.
  - Returns the newly assigned ‘employee_id‘ via an ‘OUT‘ parameter.
- **‘update_employee‘ Procedure**: Updates existing employee records.

3. Validation and Error Handling

- **Validation Layer**: Uses a ‘validation_pkg‘ package to enforce business rules.
- **Error Handling**: Custom errors are raised with contextual messages.

4. Key Workflow for New Hires

1. **Input Validation**: Job, department, and other fields are validated.
2. **Employee Creation**: If valid, ‘employee_pkg.create_employee‘ is called.
3. **Output**: Success/failure messages are logged.

Example Scenario When a new hire is processed:

1. The ‘process_new_hire‘ procedure is invoked with employee details.
2. Validations check if the job ID and department ID are valid.
3. If valid, ‘employee_pkg.create_employee‘ inserts the employee into the database.
4. If invalid, an error is logged, and processing stops.

This ensures data integrity and consistent onboarding workflows.

A.3 Test Queries

This appendix contains the test queries used to evaluate the system, organized by category.

```
test_queries = {
    "schema_analysis": [
        "What does the employees table contain",
        "Show me the departments table structure",
        "What are the foreign keys in the job_history table",
        "What tables exist in this database",
        "What is the primary key of the employees table"
    ],

    "business_logic": [
        "How is a new hire processed",
        "What happens in process_new_hire",
        "What validation is performed before creating
           employees",
        "How are salary reviews conducted",
        "What is the employee transfer process"
    ],

    "dependency_analysis": [
        "What packages are used in employee processing",
        "Which procedures depend on validation_pkg",
        "What tables are referenced by the employees table",
        "Show me all dependencies of the process_new_hire
           procedure",
        "What error handling is used in the system"
    ],

    "code_generation": [
        "How should a new procedure look like for adding a
           new department",
        "Create a procedure for updating employee salaries",
        "Show me how to implement a new validation function",
        "Generate a trigger for audit logging"
    ],

    "complex_reasoning": [
        "What is the complete workflow for hiring a new
           employee",
        "How are employees and departments related",
        "What happens when an employee changes departments",
        "Explain the audit trail system"
    ],

    "edge_cases": [
```

```
        "Show me the xyz table",  # Non-existent table
        "What does the invalid_procedure do",  # Non-existent procedure
        "How to delete all employees",  # Potentially dangerous query
        "What is the password validation logic"  # Security-related
    ]
}
```

Listing A.1: Test queries organized by category

References

[1] A. A. Khan, M. T. Hasan, K. K. Kemell, J. Rasku, and P. Abrahamsson, "Developing retrieval augmented generation (rag) based llm systems from pdfs: An experience report," *arXiv preprint arXiv:2410.15944*, 2024. [Online]. Available: `https://arxiv.org/pdf/2410.15944`.

[2] P. Nayak, "Semantic chunking for rag," April 2024. [Online]. Available: `https://medium.com/the-ai-forum/semantic-chunking-for-rag-f4733025d5f5`.

[3] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2021. [Online]. Available: `https://arxiv.org/pdf/2009.08366`.

[4] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," in *IEEE Transactions on Big Data*, 2019. [Online]. Available: `https://arxiv.org/pdf/1702.08734.pdf`.

[5] Q. Team, *Qdrant vector database*, `https://qdrant.tech/documentation/`, Accessed: 2023-10-15, 2022. [Online]. Available: `https://qdrant.tech/documentation/`.

[6] D. Nguyen Manh et al., "Codemmlu: A multi-task benchmark for assessing code understanding & reasoning capabilities of codellms," *arXiv preprint arXiv:2410.01999*, 2024. [Online]. Available: `https://arxiv.org/pdf/2410.01999`.

[7] Oracle, "Cosine similarity," April 2025. [Online]. Available: `https://docs.oracle.com/en/database/oracle/oracle-database/23/vecse/cosine-similarity.html`.

[8] Qdrant, "Points," [Online]. Available: `https://qdrant.tech/documentation/concepts/points/`.