Vietnamese – German University
DEPARTMENT OF COMPUTER SCIENCE

Frankfurt University of Applied Sciences
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Distributed Systems: Multi-index search with LLM

Team 20: Ha Nguyen Nhat Minh - 10421038
Nguyen Ha Phan - 10421099
Bui Vu Khoa - 10421028
Le Nguyen Quynh An - 10421118
Nguyen Xuan Tien - 10421058

Lecturer: Dr. Phan Trong Nhan

Binh Duong, Vietnam

Contents

Chapter 1

Problem Statement

1.1 Context and Background

In recent years, Large Language Models (LLMs) like ChatGPT, Claude, and Gemini have grown in strength and popularity in a variety of fields. These models show remarkable abilities in the generation, reasoning, and understanding of natural language; making them highly popular in with common people for day-to-day usage across a variety of simple tasks. However, most commonly accessible implementations of the technology suffers issues with memory and factual consistency, especially when it comes to domain-specific or real-time tasks. To enhance their performance on this front, LLMs are frequently combined with Retrieval-Augmented Generation (RAG) systems, giving them better capabilities for working with information that was not initially built into their training data. This project aims to allow the querying and processing of multi-index distributed databases LLMs using implementations of the RAG technology.

1.2 Existing Solution and Limitations

RAG presents a way to improve LLM outputs by obtaining pertinent documents from an outside knowledge base, but it also presents a number of difficulties.

- **Dependency on embedding quality**: Relevant context might not be recovered if the embedding model fails to accurately capture semantic similarity. For instance, a simple query for a numeric value or specific code may not align well with the semantic space of the document vectors, leading to missed retrievals

- **Single-source bottleneck**: A central point of failure and limited flexibility are introduced by the majority of RAG systems' reliance on a single vector store or knowledge base.

- **Problems with latency and scaling**: RAG systems may experience higher latency and decreased responsiveness as document size or retrieval scope increases.

- **Over-reliance on static context**: Retrieved passages may take up too much space in the prompt and limit the underlying LLM's ability to generate new content.

1.3 Proposed Approach

To solve these problems, this project proposes a smarter chat bot that uses a two-step hybrid search process.

- **1. Understand the User's Real Question**: First, the chat bot sends the user's query to a fast Large Language Model (like Mistral). This LLM acts as a parser, pulling out specific filters (i.e. department name, job title, etc.) and structures them as a JSON object.

- **2. Perform a "Smart Search"**: The application combines the specific filters from the LLM with a general semantic search in the Qdrant Vector Database.

The end result is a useful chatbot platform that enables comparative querying across models and data sources, such as DeepSeek R1, GPT 4.1, and Mistral, facilitating more resilient and contextually aware interactions in multi-agent settings.

1.4 Supporting Elements and Scope

Large Language Models (LLMs) such as GPT, Claude, and Gemini continue to excel in natural language processing and are increasingly being used in applications that require domain-specific or real-time knowledge. Retrieval-Augmented Generation (RAG) has emerged as an effective method for improving LLM outputs by retrieving relevant data from external knowledge bases. However, most existing RAG systems rely on a single vector store or index, which presents significant scalability, fault tolerance, and domain isolation challenges, especially in complex, multi-departmental environments.

In a real-world organization, different teams (e.g., HR, Engineering, Finance) frequently work with different datasets, have different access requirements, and require domain-specific interpretations. A single centralized index fails to address these needs, resulting in:

- Retrieval noise from irrelevant domains.
- Difficulty in managing updates or access policies per department.
- A single point of failure that reduces robustness.

As a result, using multiple indexes, each representing a distinct domain or department, provides numerous benefits, including increased relevance, scalability, and modular design. However, this distributed setup presents new technical challenges that must be explicitly addressed.

- **Routing Complexity**: Determine which index or set of indexes to query based on user intent.
- **Load balancing and failure isolation**: Ensuring that no index is overloaded while others are underutilized, and that failures in one do not affect the other.

- **Consistency management**: entails keeping multiple indexes in sync and up to date across departments, preventing stale responses.

This project looks into a distributed, multi-index RAG architecture to address these limitations. It aims to create a system that understands user intent using lightweight LLM parsing, performs hybrid retrieval across multiple Qdrant collections, and manages routing, load balancing, and consistency in a scalable manner. The use of simulated multi-department data creates a realistic scenario for assessing the design under distributed conditions.

Chapter 2

System Architecture

2.1 Overview

Our system is built on a distributed microservices architecture designed for scalability and fault tolerance. The flow begins with a User submitting a User Query to the Chainlit Client web application. This query is then dispatched as a task to a Task dispatch module, which pushes it to a Redis Broker (Task Queues).

From the queues, a pool of Celery Workers pulls and processes tasks based on their type. These workers interact with various data storage components, including a PostgreSQL DB for source data, a Qdrant DB for vector indexes, and AWS S3 for logging storage. The workers also make calls to external LLM APIs for tasks like filter extraction and synthesis. Once the processing is complete, the Chainlit client receives the processed context from the workers and synthesises the final answer to display back to the user.

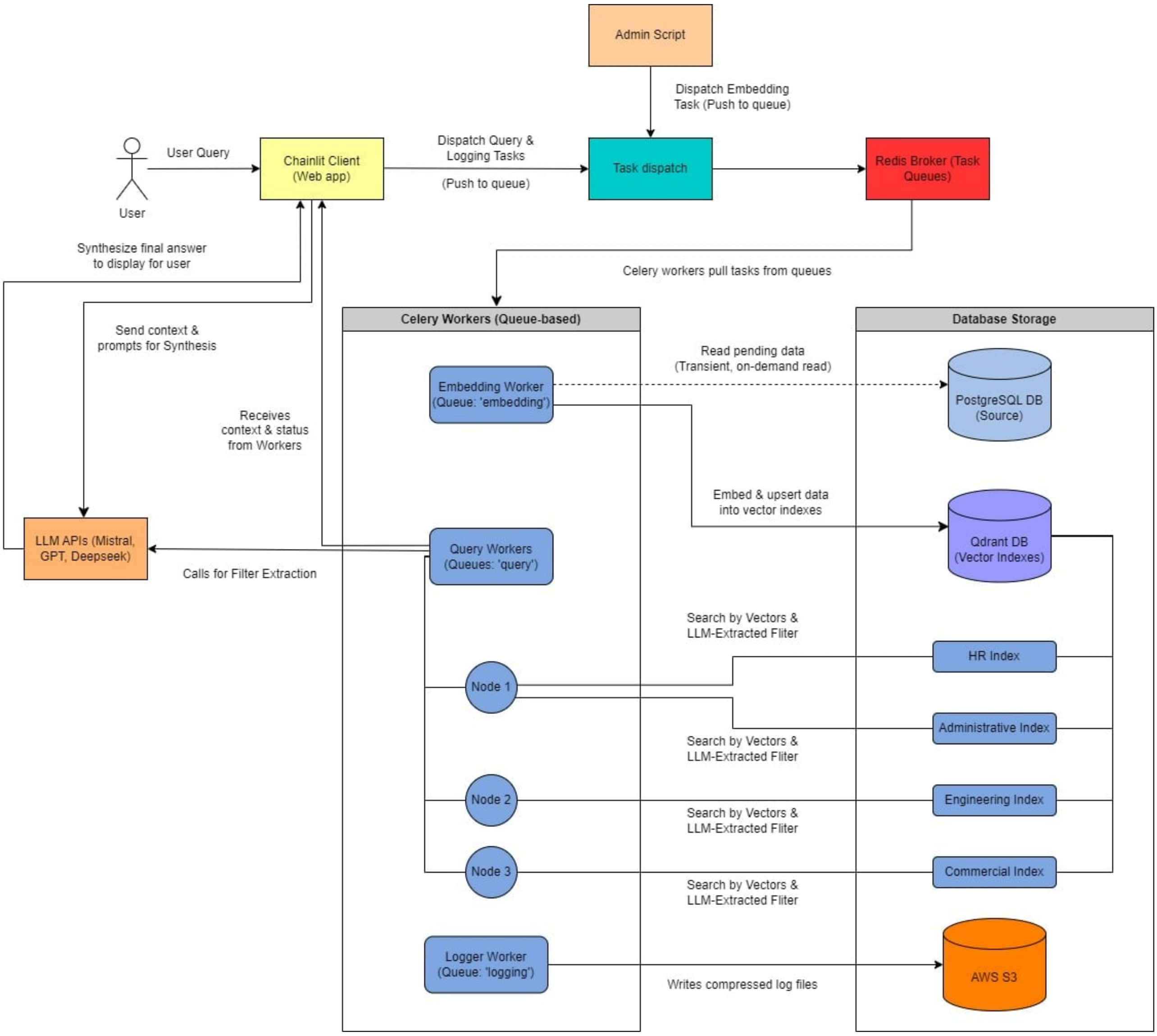

Figure 2.1: Employee Q&A System Architecture

2.1.1 Client

The Chainlit Client (Web app) serves as the user-facing web application. It is responsible for handling user input and displaying the chat interface. It dispatches query and logging tasks to the task queue. After receiving context and status from the workers, the client makes calls to LLM APIs for filter extraction and synthesis of the final answer, which is then displayed to the user.

2.1.2 Backend Services

The backend is composed of several decoupled services that work together through the task queue system, as depicted in the diagram.

- **Admin Script**: This is a command-line interface (`main.py`) used for administrative actions, specifically for dispatching embedding generation tasks to the queue.

- **Task Dispatch**: This is the initial entry point for tasks from the frontend or the admin script, pushing them into the Redis Broker.

- **Redis Broker (Task Queues)**: Configured via `config.py` as the `CELERY_BROKER_URL`, Redis acts as the central message broker for Celery. It holds different queues (`embedding`, `query`, `logging`) to manage and distribute tasks to the workers.

- **Celery Workers (Queue-based)**: These are stateless worker processes that pull tasks from the Redis queues and execute them asynchronously. The diagram shows three types of workers, each handling a specific queue:

  - **Embedding Worker**: This worker pulls tasks from the `embedding` queue. It reads pending data from the PostgreSQL database, generates vector embeddings using the `all-MiniLM-L6-v2` model, and upserts the data into the Qdrant vector indexes. This process is implemented as a chained task (`prepare_employee_vectors_task` followed by `upsert_vectors_task`) to ensure a sequential flow.

  - **Query Workers**: These workers pull tasks from `query` queues, which are dynamically routed based on the Qdrant collection being queried. Their primary role is to search the relevant Qdrant indexes (e.g., `HR Index`, `Administrative Index`) using the query's vector and filters extracted by an LLM.
  - **Logger Worker**: This worker pulls tasks from the `logging` queue. Its sole responsibility is to write compressed chat log files to **AWS S3** for long-term storage.

- **LLM APIs (Mistral, GPT, Deepseek)**: These external services are called by the Chainlit client to perform two distinct functions: extracting structured filters from the user query and synthesizing the final human-readable answer from the retrieved context.

2.1.3 Database Storage

The system uses a multi-layered storage approach to handle different data types and access patterns.

- **PostgreSQL DB (Source)**: This is the primary source of truth for all structured employee data. The `db_pool_setup.py` module manages a connection pool to ensure efficient and robust access from the Celery workers.
- **Qdrant DB (Vector Indexes)**: This is a high-performance vector database that holds vector indexes for different departments. The diagram shows multiple indexes, which correspond to collections in Qdrant as defined in `config.py`. This multi-index design enables parallel and targeted searches.
- **AWS S3**: This is an object storage service used for the long-term archival of compressed chat logs, written to by the Logger Worker.

Chapter 3

Chat Application Functionality

The Employee Q&A System is designed to provide users with a powerful and interactive experience for retrieving employee information. Here are the core and additional functionalities implemented:

3.1 Core Functionalities

3.1.1 Natural Language Querying:

The system allows users to ask questions about employees using natural language queries, such as "Who is the lead engineer on the new security project?" or "Find all employees with Python skills in the Sales department."

3.1.2 Multi-Departmental Search

The system can simultaneously search for relevant employee information across multiple departments (*e.g., Engineering, Sales, HR, Marketing, Finance, Operations)* by querying different Qdrant collections in parallel. This ensures comprehensive results and reduces search time. The departments are configured in `config.py`

3.1.3 Context-Aware Answering

Based on the retrieved context from the vector database, an LLM synthesises a clear and concise answer to the user's query. The LLM is instructed to use only the provided context and to state if the information is not available.

3.1.4 Real-Time Asynchronous Processing:

The system leverages Celery to process user queries asynchronously. When a user submits a query, tasks are dispatched to workers in the background, allowing the frontend to remain responsive. A loading spinner indicates that the system is searching for information.

3.2 Additional Functionalities

1. Structured Filter Extraction:

The system can automatically analyse a user's query and extract structured filters such as `department`, `job_title`, `employee_id`, `full_name`, `location`, `email`, and `min_satisfaction`. These filters are used to refine the search results in the vector database, improving retrieval accuracy.

2. LLM Selection:

The user interface displays a live task list with a running status, showing that the system is actively searching across different departments for context.

3. Live Task Tracking:

The user interface displays a live task list with a running status, showing that the system is actively searching across different departments for context.

4. Conversation History Logging:

User queries and the assistant's responses are buffered in Redis and then asynchronously stored as compressed JSON files in an AWS S3 bucket for long-term storage and analysis.

5. Configurable Relevance Threshold:

The system only considers retrieved context if its similarity score exceeds a configurable `relevance threshold` of 0.70, ensuring that only highly relevant information is used to answer the query.

3.3 Technology Stack

The application is built on a robust and scalable technology stack, leveraging modern tools and frameworks to ensure high performance, real-time capabilities, and maintainability.

1. PostgreSQL:
As the database solution for storing raw, structured employee data, PostgreSQL is used. It is a powerful open-source relational database that is highly reliable and extensible. The system utilises a connection pool to manage database connections efficiently, as configured in `db_pool_setup.py`.

2. Sentence-Transformers:
The `sentence-transformers` library is used to generate dense vector embeddings from employee profile text. The specific model used is all-MiniLM-L6-v2, which is configured in `config.py` with a dimension of 384.

3. Qdrant:
Qdrant is a high-performance vector database used to store the generated embeddings. It supports fast semantic search and filtering on payloads, which is crucial for the retrieval phase of the RAG pipeline. The application interacts with Qdrant via the `qdrant_client` library. The local instance of the Qdrant service is hosted using Docker.

4. Celery:
Celery serves as the distributed task queue for asynchronous processing. It is used to dispatch and manage background tasks such as embedding generation, vector upserting, and parallel queries to different departments, ensuring the main

application remains responsive.

5. Redis:

Redis is utilised for two main purposes: as the message broker for Celery and as a temporary buffer for chat logs. It provides a fast, in-memory data store for these functions. The Redis server is run using a Docker container, simplifying its deployment and management.

6. Chainlit:

Chainlit is the framework used for building the conversational user interface. It handles user input and displays messages, and also provides UI elements like task lists and action buttons for selecting LLMs.

7. LLM APIs (Mistral, OpenAI, DeepSeek):

The system integrates with multiple large language models via their respective APIs. The OpenAI client is used for interacting with all three models, with configurations for each defined in `config.py`.

8. AWS S3:

Amazon S3 is used for durable, long-term storage of compressed chat logs. Logs are buffered in Redis and then asynchronously uploaded to a designated S3 bucket.

9. Boto3:

The `boto3` library is used to interact with AWS services, specifically to upload compressed chat logs to the S3 bucket.

10. Docker:

The modular, microservice-oriented architecture leverages Docker to containerise key services. This approach simplifies deployment and ensures consistent environments. Specifically, Docker is used to run the Redis server and host the Qdrant vector database locally, as specified by the `QDRANT_HOST` and `REDIS_HOST` environment variables.

Chapter 4

Distributed Systems Characteristics

4.1 Scalability

An application's capacity to scale up or down in response to changing user load is the main goal of scalability testing, a subset of performance testing. Scalability is important for real-time chat applications since it affects the application's capacity to support multiple users at once and continue to function well under heavy load.

Our system is designed to scale horizontally to handle increased load through several key architectural decisions:

- **Celery and Redis for asynchronous task processing**. This design decouples resource-intensive operations from the user-facing application, ensuring the system remains highly responsive even under heavy load.

- **Asynchronous Task Dispatch**: The user query is not processed synchronously by the web application. Instead, the Chainlit frontend dispatches a task to the Celery broker (Redis). This allows the web application to immediately handle new user requests without waiting for the query processing to complete.

Below is the code snippet from `chainlit_app.py` that dispatches the tasks:

```
departments_to_query = config.DEPARTMENTS
        task_group = group(
            process_employee_query_task.s(query=query_text, target_department=dept)
            for dept in departments_to_query
        )
        group_result = task_group.apply_async()
```

- **Parallel Query Execution**: For a single user query, the `chainlit_app.py` frontend dispatches a Celery `group` of tasks, which runs a separate `process_employee_query_task` for each configured department simultaneously. This parallel processing drastically reduces the total search time, as searches across different Qdrant collections occur concurrently. The use of a Redis task queue allows these tasks to be executed in parallel by multiple Celery workers.

- **Horizontal Scaling of Workers**: The use of a Redis broker and Celery workers allows for adding more worker processes to handle a higher volume of tasks concurrently. The `ProjectRouter` class dynamically routes tasks to specific queues (`query-collection_name`, `embedding`, `logging`) based on the task type or department. This enables load balancing and resource allocation strategies, where workers can be dedicated to specific, high-demand queues, such as those handling queries for a busy department.

- **Batch Processing for Efficiency**: The embedding generation process fetches employee IDs in batches of 90 (`QDRANT_UPSERT_BATCH_SIZE`) and processes them in a single task. This asynchronous batching optimizes database reads and Qdrant upsert operations, improving the efficiency of bulk data ingestion and preventing the ingestion process from blocking other tasks.

- **Asynchronous Logging**: The `buffer_chat_log` function adds logs to a Redis list and then dispatches a Celery task to store them in S3 only when a certain `log_buffer_threshold` is reached. This prevents synchronous logging from becoming a bottleneck during peak usage and allows the system to handle high log volume without performance degradation.

4.2 Fault Tolerance

Our real-time Employee Q&A System is designed with a fault-tolerant mechanism that guarantees business continuity. This is achieved through several key mechanisms.

Firstly, the application uses a **database connection pool** to maintain a set of ready-to-use database connections. This prevents the application from failing under high load by ensuring connections are efficiently reused. The `db_connection()` context manager ensures that connections are always returned to the pool after use, even if an exception occurs during a database transaction, preventing resource leaks.

Below is the code snippet from `db_pool_setup.py` that demonstrates the context manager's rollback and return logic:

```python
@contextmanager
def db_connection():
    conn = None
    pool = get_pool()

    try:
        conn = pool.getconn()
        if not conn:
            raise ConnectionError("Failed to acquire a database connection from the pool")
```

```

        logger.debug(f"Acquired connection {id(conn)} from pool")

        yield conn

        conn.commit()
        logger.debug(f"Committed transaction on connection {id(conn)}")

    except Exception as e:
        if conn:
            try:
                conn.rollback()
                logger.warning(f"Rolled back connection {id(conn)} due to an exception")

            except Exception as rollback_err:
                logger.exception(f"Rollback failed on connection {id(conn)}: {rollback_err}")

        logger.exception("Error during DB transaction")
        raise

    finally:
        if conn:
            if db_pool:
                try:
                    pool.putconn(conn)
                    logger.debug(f"Returned connection {id(conn)} to pool")

                except Exception as e:
                    logger.exception(f"Error returning connection to pool: {e}")

                    try:
```

```
                    conn.close()
                    logger.warning(f"Force-closed connection {id(conn)} after pool failure")

                except Exception as e2:
                    logger.exception(f"Failed to force-close connection {id(conn)}: {e2}")

        else:
            logger.warning(
                f"Pool object (or global db_pool) became None in finally. "
                f"Attempting to close conn id={id(conn)}."
            )
            try:
                conn.close()

            except Exception as e_close:
                logger.exception(f"Failed to force-close connection id={id(conn)} when pool was None: {e_close}")
```

Furthermore, the application implements robust **error handling and retry mechanisms** for asynchronous tasks. Celery tasks like `prepare_employee_vectors_task`, `upsert_vectors_task`, and `process_employee_query_task` are configured with `max_retries=3`. This ensures that temporary failures, such as network disruptions or service unavailability, are handled gracefully by automatically re-attempting the operation after a delay.

Below is a snippet from `prepare_employee_vectors_task` showing the retry logic:

```
except Exception as e:
        logger.warning(f"Task for {table_name} failed on embedding 
```

```
attempt {self.request.retries + 1}. Error: {e}")
        try:
            raise self.retry(exc=e, countdown=60)
        except MaxRetriesExceededError:
            logger.error(f"Task for {table_name} has failed permanently. Marking records as FAILED.")
            update_employee_status_in_db(employee_ids_batch, table_name, ProcessingStatus.FAILED)
            raise e
```

If a task fails permanently after all retries, the system updates the corresponding records in the database to a `FAILED` status, providing a persistent record of the failure and preventing indefinite retries.

The application also uses a **threading lock** during the initialization of the database pool to prevent race conditions and ensure the pool is set up only once, even if called by multiple threads.

Finally, the system ensures **graceful degradation** and **component-level skipping**. As seen in the `query_service.py` code, if a Qdrant collection for a specific department is not found, the system logs a warning and skips that department's search instead of failing the entire query. This allows the system to continue searching in other available collections, ensuring a partial result can still be provided to the user. Here is the relevant code snippet:

```
dept_info = config.DEPARTMENT_CONFIG.get(target_department)
        if not dept_info:
            logger.warning(f"No config found for department '{target_department}'. Skipping.")
            return {"status": "skipped_no_config", "context": None, "source": target_department}

        collection_name = dept_info['qdrant_collection']

```

```
    try:
         client.get_collection(collection_name=collection_name)
    except Exception:
        logger.warning(f"Collection '{collection_name}' not found. Skipping department '{target_department}'.")
        return {"status": "skipped_no_collection", "context": None, "source": target_department}
```

4.3 Concurrency

The system is designed to handle multiple tasks simultaneously, which is a fundamental aspect of concurrency.

- **Parallel Task Execution**: When a user submits a query, the `chainlit_app.py` component uses Celery's `group` primitive to dispatch a separate `process_employee_query_task` for each department defined in the configuration. This allows the searches to run concurrently on different worker nodes, enabling the system to retrieve context from all relevant departments at the same time. The `await await_group_task_result` function then polls the `GroupResult` to wait for all the parallel tasks to complete. Below is is the snippet demonstrating this parallel execution:

```
task_group = group(
        process_employee_query_task.s(query=query_text, target_department=dept)
        for dept in departments_to_query
    )

    group_result = task_group.apply_async()

    results = await await_group_task_result(group_result, task_list)
```

- **Thread Safety**: A `threading.Lock` is used in `db_pool_setup.py` to ensure that the database connection pool is initialized only once, even if `initialize_pool()` is called by multiple threads concurrently. This prevents race conditions and ensures a consistent state for the connection pool in a multi-threaded environment.

4.4 Data Consistency

Maintaining consistency across multiple distributed data stores is a significant challenge. In our system, which uses both PostgreSQL and Qdrant, we ensure data consistency between the source data and its vector representation through a controlled, sequential task flow.

This is achieved by implementing the data ingestion process as a **Celery task chain**. A chain guarantees that a task will only execute after the preceding task in the chain has completed successfully.

The following code from `text_embedding.py` shows how a chain is created to link the vector preparation and upsert tasks:

```
task_chain = chain(
                        prepare_employee_vectors_task.s(batch_ids, table_name, dept_name),
                        upsert_vectors_task.s(collection_name, table_name)
                    )
                    task_chain.apply_async(queue="embedding")
```

As shown, `upsert_vectors_task` is configured to run only after `prepare_employee_vectors_task` has successfully generated the embeddings. This ensures that invalid or incomplete vectors are not written to the Qdrant index.

Furthermore, a critical aspect of consistency is the atomic update of the embedding status in the source database. The `embedding_status` in the PostgreSQL table is updated to `COMPLETED` only after the vectors have been successfully upserted to Qdrant. If the upsert task fails, the status remains `PENDING`, allowing the record to be picked up for re-processing in a subsequent run.

Here is the relevant code snippet from the `upsert_vectors_task`:

```
upserted_ids = [int(p.payload["employee_id"]) for p in point_structs]
        update_employee_status_in_db(upserted_ids, table_name, ↩
            ProcessingStatus.COMPLETED)
```

This status tracking mechanism guarantees that the `qdrant_collection` is eventually consistent with the `postgres_table` and provides a reliable way to monitor the data ingestion pipeline.

Chapter 5

Implementation Details

5.0.1 Project Set-up

5.0.2 Package installation

The project environment is configured by importing several key libraries necessary for the distributed RAG system. These include `psycopg2` for PostgreSQL connectivity, `qdrant_client` for vector database operations, `sentence-transformers` for creating embeddings, `celery` for task management, `redis` for the broker and buffer, `boto3` for AWS S3 interactions, and `openai` for LLM integrations. Environment variables, crucial for configuring services like Redis, PostgreSQL, and LLM APIs, are loaded using `python-dotenv`. The central configuration for the entire system is managed through the `config.py` module.

```python
import os
import logging
from logging.handlers import RotatingFileHandler
from dotenv import load_dotenv

dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
if os.path.exists(dotenv_path):
    load_dotenv(dotenv_path=dotenv_path)

else:
    load_dotenv()
    if not os.getenv("POSTGRES_DBNAME"):
        print("Warning: .env file not found or not loaded correctly.")

```

```
APP_NAME = "Employee_Q&A_System"

# Celery Configuration
CELERY_BROKER_URL = os.getenv("CELERY_BROKER")
CELERY_RESULT_BACKEND = os.getenv("CELERY_BACKEND")

# Redis Configuration
REDIS_HOST = os.getenv("REDIS_HOST")
REDIS_PORT = os.getenv("REDIS_PORT")
REDIS_DB = os.getenv("REDIS_DB")
REDIS_LOG_LIST_KEY_PREFIX = "chatlogs_list:"

# PostgreSQL Configuration
POSTGRES_DB_MIN_CONN = os.getenv("POSTGRES_DB_MIN_CONN")
POSTGRES_DB_MAX_CONN = os.getenv("POSTGRES_DB_MAX_CONN")
POSTGRES_DBNAME = os.getenv("POSTGRES_DBNAME")
POSTGRES_USER = os.getenv("POSTGRES_USER")
POSTGRES_PASSWORD = os.getenv("POSTGRES_PASSWORD")
POSTGRES_HOST = os.getenv("POSTGRES_HOST")
POSTGRES_PORT = os.getenv("POSTGRES_PORT")

DEPARTMENT_CONFIG = {
    "Engineering": {
        "postgres_table": "employees_engineering",
        "qdrant_collection": "corp-dir-engineering"
    },
    "Sales": {
        "postgres_table": "employees_sales",
        "qdrant_collection": "corp-dir-commercial"
    },
    "HR": {
        "postgres_table": "employees_hr",
        "qdrant_collection": "corp-dir-hr"
    },
    "Marketing": {
        "postgres_table": "employees_marketing",
```

```
            "qdrant_collection": "corp-dir-commercial"
        },
        "Finance": {
            "postgres_table": "employees_finance",
            "qdrant_collection": "corp-dir-administrative"
        },
        "Operations": {
            "postgres_table": "employees_operations",
            "qdrant_collection": "corp-dir-administrative"
        }
}
DEPARTMENTS = list(DEPARTMENT_CONFIG.keys())

# AWS S3 Configuration
AWS_BUCKET_NAME = os.getenv("AWS_BUCKET_NAME")

# Qdrant Configuration ---
QDRANT_HOST = os.getenv("QDRANT_HOST", "localhost")
QDRANT_PORT = int(os.getenv("QDRANT_PORT", 6333))
QDRANT_UPSERT_BATCH_SIZE = 90
QDRANT_NAMESPACE = "employee-data"

# Local Embedding Model
LOCAL_EMBEDDING_MODEL_NAME = "all-MiniLM-L6-v2"
LOCAL_EMBEDDING_MODEL_DIMENSION = 384

# LLM Models Information
MISTRAL_MODEL_CHOICE = 'Mistral'
MISTRAL_MODEL = 'mistral-small-latest'
OPENAI_MODEL_CHOICE = 'GPT 4.1'
OPENAI_MODEL = 'gpt-4.1-2025-04-14'
DEEPSEEK_MODEL_CHOICE = 'DeepSeekR1'
DEEPSEEK_MODEL = 'deepseek/deepseek-1:free'
DEEPSEEK_BASEURL = os.getenv("DEEPSEEK_BASEURL")

# LLM API Keys
```

```
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
MISTRAL_API_KEY = os.getenv("MISTRAL_API_KEY")
DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY")

# Logging Configuration
LOG_DIR = os.path.join(os.path.dirname(__file__), '..', 'logs')
LOG_FILENAME = os.path.join(LOG_DIR, 'Employee_Q&A_System.log')
LOG_LEVEL = logging.INFO
LOG_FORMAT = '%(asctime)s - %(levelname)s - [%(name)s:%(funcName)s] -↩
    %(message)s'
DATE_FORMAT = '%Y-%m-%d %H:%M:%S'
LOG_MAX_BYTES = 5 * 1024 * 1024
LOG_BACKUP_COUNT = 3
LOG_BUFFER_THRESHOLD = 100
LOG_BUFFER_TTL_SECONDS = 3600

# Logging Setup Function
def setup_logging():
    os.makedirs(LOG_DIR, exist_ok=True)
    logger = logging.getLogger(APP_NAME)
    logger.setLevel(LOG_LEVEL)

    if logger.hasHandlers():
        logger.handlers.clear()

    formatter = logging.Formatter(LOG_FORMAT, datefmt=DATE_FORMAT)

    # Rotating File Handler
    file_handler = RotatingFileHandler(
        LOG_FILENAME,
        maxBytes = LOG_MAX_BYTES,
        backupCount = LOG_BACKUP_COUNT,
        encoding = 'utf-8'
    )
    file_handler.setLevel(LOG_LEVEL)
    file_handler.setFormatter(formatter)
```

```
    logger.addHandler(file_handler)

    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.INFO)
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    logger.info("Logging setup complete. Logging to %s", LOG_FILENAME)
    return logger
```

5.0.3 Backend Routing and Task Set-up

Instead of traditional REST routes for core functionality, the backend's routing is managed by Celery's task queuing system. The `ProjectRouter` class defines custom routing logic to direct tasks to specific queues based on their purpose or the data they need to access. This is a crucial component of the distributed architecture, enabling efficient workload distribution and resource management across different Celery workers.

Below is the complete code for the `celery_app.py` file, which sets up the Celery application and its routing logic:

```
import config
from celery import Celery, signals
from backend import db_pool_setup

if not config.CELERY_BROKER_URL:
    raise ValueError("An error has occurred. CELERY_BROKER_URL not found in config")

class ProjectRouter:
    def route_for_task(self, task, args=None, kwargs=None):
        # 1. Dynamic routing for our main query task.
```

```
        if task == 'backend.query_service.process_employee_query_task':
            if kwargs and 'target_department' in kwargs:
                dept = kwargs['target_department']

                if dept in config.DEPARTMENT_CONFIG:
                    collection_name = config.DEPARTMENT_CONFIG[dept]['qdrant_collection']
                    return {
                        'queue': f'query-{collection_name}'
                    }

        # 2. Static routing for embedding tasks.
        if task.startswith('backend.text_embedding.'):
            return {'queue': 'embedding'}

        # 3. Static routing for logging tasks.
        if task.startswith('backend.chatlog_storage.'):
            return {'queue': 'logging'}

        return None

# Celery App Initialization
celery_app = Celery(
    config.APP_NAME,
    broker=config.CELERY_BROKER_URL,
    backend=config.CELERY_RESULT_BACKEND,
    include=[
        'backend.text_embedding',
        'backend.query_service',
        'backend.chatlog_storage'
    ]
)

celery_app.conf.update(
    task_serializer='json',
```

```
    accept_content=['json'],
    result_serializer='json',
    timezone='Asia/Ho_Chi_Minh',
    enable_utc=True,
    task_acks_late=True,
    worker_prefetch_multiplier=1,
    task_routes = (ProjectRouter(),)
)

import logging
logger = logging.getLogger(config.APP_NAME)

@signals.worker_process_init.connect
def initialize_worker_process(**kwargs):
    logger.info("Initializing Celery worker process...")
    try:
        db_pool_setup.initialize_pool()
        logger.info("Database pool initialized for worker process.")
    except Exception as e:
        logger.error(f"Error initializing database pool in worker: {e↩
            }", exc_info=True)

@signals.worker_process_shutdown.connect
def shutdown_worker_process(**kwargs):
    logger.info("Shutting down Celery worker process")
    try:
        db_pool_setup.close_pool()
    except Exception as e:
        logger.error(f"Error closing database pool in worker: {e}", ↩
            exc_info=True)
```

5.1 Database and Vector Index Implementation

5.1.1 Database Set-up

The system connects to a PostgreSQL database using `psycopg2` and a connection pool to manage concurrent access efficiently. The `initialize_pool()` function, protected by a `threading.Lock`, ensures the pool is set up safely and only once. This pool is initialized when a Celery worker process starts and is gracefully closed upon shutdown.

```
import psycopg2
import threading
import config
from contextlib import contextmanager
from psycopg2 import pool as psycopg_pool

import logging
logger = logging.getLogger(config.APP_NAME)

db_pool = None
pool_init_lock = threading.Lock()

def initialize_pool():
    global db_pool

    with pool_init_lock:
        if db_pool:
            logger.debug("Database pool already initialized")
            return db_pool

        logger.info("Initializing PostgreSQL connection pool")

        try:
```

```python
            conn_args = {
                "dbname": config.POSTGRES_DBNAME,
                "user": config.POSTGRES_USER,
                "password": config.POSTGRES_PASSWORD,
                "host": config.POSTGRES_HOST,
                "port": config.POSTGRES_PORT
            }

            db_pool = psycopg_pool.SimpleConnectionPool(
                minconn=config.POSTGRES_DB_MIN_CONN,
                maxconn=config.POSTGRES_DB_MAX_CONN,
                **conn_args
            )


            logger.info(
                f"Successfully initialized PostgreSQL pool "
                f"(min: {config.POSTGRES_DB_MIN_CONN}, max: {config.POSTGRES_DB_MAX_CONN})"
            )

            return db_pool

        except Exception as e:
            db_pool = None
            logger.exception("Failed to initialize connection pool")
            raise RuntimeError("PostgreSQL connection pool initialization failed") from e

def close_pool():
    global db_pool

    if not db_pool:
        logger.warning("Attempted to close PostgreSQL connection pool, but it was not initialized")
        return
```

```

    logger.info("Closing PostgreSQL connection pool")

    try:
        db_pool.closeall()
        logger.info("Successfully closed PostgreSQL connection pool")

    except Exception as e:
        logger.exception(f"Exception while closing PostgreSQL ↩
            connection pool: {e}")

    finally:
        db_pool = None

def get_pool():
    global db_pool
    if not db_pool:
        logger.debug("Connection pool not initialized, calling ↩
            initialize_pool()")
        pool = initialize_pool()
        if not pool:
             raise ConnectionError("Database pool could not be ↩
                initialized")
        return pool
    return db_pool

# Context Manager using global db_pool
@contextmanager
def db_connection():
    conn = None
    pool = get_pool()

    try:
        conn = pool.getconn()
        if not conn:
             raise ConnectionError("Failed to acquire a database ↩
```

```
            connection from the pool")

        logger.debug(f"Acquired connection {id(conn)} from pool")

        yield conn

        conn.commit()
        logger.debug(f"Committed transaction on connection {id(conn)}")

    except Exception as e:
        if conn:
            try:
                conn.rollback()
                logger.warning(f"Rolled back connection {id(conn)} due to an exception")

            except Exception as rollback_err:
                logger.exception(f"Rollback failed on connection {id(conn)}: {rollback_err}")

        logger.exception("Error during DB transaction")
        raise

    finally:
        if conn:
            if db_pool:
                try:
                    pool.putconn(conn)
                    logger.debug(f"Returned connection {id(conn)} to pool")

                except Exception as e:
                    logger.exception(f"Error returning connection to pool: {e}")

```

```
                try:
                    conn.close()
                    logger.warning(f"Force-closed connection {id(conn)} after pool failure")

                except Exception as e2:
                    logger.exception(f"Failed to force-close connection {id(conn)}: {e2}")

        else:
            logger.warning(
                f"Pool object (or global db_pool) became None in finally. "
                f"Attempting to close conn id={id(conn)}."
            )
            try:
                conn.close()

            except Exception as e_close:
                logger.exception(f"Failed to force-close connection id={id(conn)} when pool was None: {e_close}")
```

5.1.2 Data Schema in PostgreSQL and Qdrant

The data is structured across two main storage systems: PostgreSQL for raw data and Qdrant for vectorized data. In PostgreSQL, employee data is stored in separate tables for each department (e.g., `employees_engineering`), as mapped in `config.DEPARTMENT_CONFIG`. Each employee record includes fields such as `employee_id`, `full_name`, `job_title`, `start_date`, `skills`, `project_history`, `email`, and `location`. A crucial field, `embedding_status`, tracks whether the record has been processed for vectorization, which is key for data consistency.

When an employee's profile is vectorized, the resulting vector is stored in a

Qdrant collection along with a structured `payload`. This payload acts as the schema for the vector index, containing key-value pairs like `"text"`, `"employee_id"`, `"full_name"`, `"job_title"`, and other attributes. This dual storage of the raw text and structured metadata is essential for enabling both semantic search and filtered search.

The core logic for fetching data from PostgreSQL, generating embeddings, and preparing the Qdrant payload is contained in the `text_embedding.py` file, shown below:

```
import logging
import uuid
from enum import Enum
from celery import chain
from celery.exceptions import MaxRetriesExceededError
from qdrant_client import QdrantClient, models
from sentence_transformers import SentenceTransformer

import config
from backend import db_pool_setup

logger = logging.getLogger(config.APP_NAME)

# Celery App Import
try:
    from celery_app import celery_app
    logger.info("text_embedding.py: celery_app imported successfully.")
except ImportError as e:
    logger.exception("text_embedding.py: Could not import celery_app. %s", e)
    raise

# Initialize Sentence Transformer Model
try:
```

```
    embedding_model = SentenceTransformer(config.LOCAL_EMBEDDING_MODEL_NAME)
    logger.info(f"Successfully loaded sentence-transformer model: {config.LOCAL_EMBEDDING_MODEL_NAME}")
except Exception as e:
    logger.critical(f"Failed to load sentence-transformer model: {e}", exc_info=True)
    embedding_model = None


# State Management
class ProcessingStatus(Enum):
    PENDING = 'pending'
    COMPLETED = 'completed'
    FAILED = 'failed'

def launch_embedding_tasks():
    logger.info("Starting Multi-Department Embedding Dispatch")
    total_tasks_launched = 0

    for dept_name, dept_config in config.DEPARTMENT_CONFIG.items():
        table_name = dept_config.get('postgres_table')
        collection_name = dept_config.get('qdrant_collection')

        if not table_name or not collection_name:
            logger.warning(f"Skipping department '{dept_name}' due to missing configuration.")
            continue

        all_pending_ids = fetch_pending_ids_for_department(table_name)

        if not all_pending_ids:
            logger.info(f"No pending employees found for department '{dept_name}'.")
            continue
```

```python

        batch_size = getattr(config, 'QDRANT_UPSERT_BATCH_SIZE', 90)

        for i in range(0, len(all_pending_ids), batch_size):
            batch_ids = all_pending_ids[i : i + batch_size]

            try:
                task_chain = chain(
                    prepare_employee_vectors_task.s(batch_ids, table_name, dept_name),
                    upsert_vectors_task.s(collection_name, table_name)
                )
                task_chain.apply_async(queue="embedding")
                total_tasks_launched += 1
                logger.info(f"Dispatched task chain for {dept_name} with {len(batch_ids)} IDs.")
            except Exception as e:
                logger.exception(f"Failed to dispatch Celery task chain for {dept_name}: {e}")

    logger.info(f"Finished Dispatching. Total tasks launched: {total_tasks_launched}")
    return {"status": "dispatch_complete", "total_tasks_launched": total_tasks_launched}

def fetch_pending_ids_for_department(table_name: str) -> list:
    logger.info(f"Fetching pending IDs from table: '{table_name}'")
    try:
        with db_pool_setup.db_connection() as conn:
            with conn.cursor() as cur:
                query = f"SELECT employee_id FROM {table_name} WHERE embedding_status = %s ORDER BY employee_id"
                cur.execute(query, (ProcessingStatus.PENDING.value,))
                return [row[0] for row in cur.fetchall()]
    except Exception as e:
```

```python
        logger.exception(f"Error fetching pending IDs from {table_name}: {e}")
        return []

def fetch_employees_by_ids(employee_ids: list, table_name: str) -> list:
    if not employee_ids:
        return []
    try:
        with db_pool_setup.db_connection() as conn:
            with conn.cursor() as cur:
                query = f"""
                    SELECT
                        employee_id, full_name, job_title, start_date,
                        skills, project_history, email, location
                    FROM {table_name}
                    WHERE employee_id = ANY(%s)
                """
                cur.execute(query, (employee_ids,))
                return [dict(zip([col.name for col in cur.description], row)) for row in cur.fetchall()]
    except Exception as e:
        logger.exception(f"Error fetching employee data from {table_name}: {e}")
        return []

def update_employee_status_in_db(employee_ids: list, table_name: str, status: ProcessingStatus):
    if not employee_ids:
        return
    logger.debug(f"Updating status to '{status.value}' for {len(employee_ids)} IDs in table '{table_name}'")
    try:
        with db_pool_setup.db_connection() as conn:
            with conn.cursor() as cur:
```

```
                query = f"UPDATE {table_name} SET embedding_status = %s WHERE employee_id = ANY(%s)"
                cur.execute(query, (status.value, employee_ids))
    except Exception as e:
        logger.error(f"DB status update to '{status.value}' failed for table '{table_name}': {e}", exc_info=True)

@celery_app.task(bind=True, max_retries=3, default_retry_delay=60, acks_late=True)
def prepare_employee_vectors_task(self, employee_ids_batch: list, table_name: str, dept_name: str):
    logger.info(f"[PREPARE TASK START] Preparing {len(employee_ids_batch)} employees from '{table_name}' ({dept_name}).")

    if not embedding_model:
        logger.error("Embedding model is not available. Task cannot proceed.")
        raise RuntimeError("SentenceTransformer model not loaded.")

    employees = fetch_employees_by_ids(employee_ids_batch, table_name)
    if not employees:
        logger.warning(f"No employee data found for IDs in '{table_name}'. Task will end.")
        return {"status": "no_data", "points": []}

    texts_to_embed = []
    employee_map = {}
    for emp in employees:
        text = (
            f"Name: {emp.get('full_name', '')}. "
            f"Title: {emp.get('job_title', '')}. "
            f"Department: {dept_name}. "
            f"Location: {emp.get('location', '')}. "
            f"Contact Email: {emp.get('email', '')}. "
```

```
            f"Start Date: {str(emp.get('start_date', ''))}. "
            f"Skills: {emp.get('skills', '')}. "
            f"Projects: {emp.get('project_history', '')}"
        )
        texts_to_embed.append(text)
        employee_map[text] = emp

    try:
        embeddings = embedding_model.encode(texts_to_embed, show_progress_bar=False).tolist()

    except Exception as e:
        logger.warning(f"Task for {table_name} failed on embedding attempt {self.request.retries + 1}. Error: {e}")
        try:
            raise self.retry(exc=e, countdown=60)
        except MaxRetriesExceededError:
            logger.error(f"Task for {table_name} has failed permanently. Marking records as FAILED.")
            update_employee_status_in_db(employee_ids_batch, table_name, ProcessingStatus.FAILED)
            raise e

    points_to_upsert = []
    namespace = uuid.UUID('f2b4e448-86d6-4c28-8b1b-5e6e5e0c8b4e')

    for i, text in enumerate(texts_to_embed):
        emp = employee_map[text]
        employee_id_str = str(emp['employee_id'])
        point_id = str(uuid.uuid5(namespace, employee_id_str))

        points_to_upsert.append({
            "id": point_id,
            "vector": embeddings[i],
            "payload": {
                "text": text,
```

```
                "employee_id": employee_id_str,
                "full_name": emp.get('full_name', ''),
                "job_title": emp.get('job_title', ''),
                "department": dept_name,
                "location": emp.get('location', ''),
                "email": emp.get('email', ''),
                "start_date": str(emp.get('start_date', '')),
                "client_satisfaction": int(emp.get('client_satisfaction', 0)),
                "skills": emp.get('skills', ''),
                "project_history": emp.get('project_history', '')
            }
        })

    logger.info(f"[PREPARE TASK SUCCESS] Prepared {len(points_to_upsert)} points for '{table_name}'.")
    return {"status": "prepared", "points": points_to_upsert}


@celery_app.task(bind=True, max_retries=3, acks_late=True)
def upsert_vectors_task(self, vectors_payload: dict, collection_name: str, table_name: str):
    points = vectors_payload.get("points")
    if not points:
        logger.warning(f"Upsert task skipped for collection '{collection_name}': No points received from prepare task.")
        return {"status": "skipped"}

    logger.info(f"[UPSERT TASK START] Upserting {len(points)} points to Qdrant collection: '{collection_name}'")

    try:
        qdrant_client = QdrantClient(host=config.QDRANT_HOST, port=config.QDRANT_PORT)

        try:
```

```
            qdrant_client.get_collection(collection_name=collection_name)
        except Exception:
            logger.info(f"Collection '{collection_name}' not found. Creating it now.")
            qdrant_client.create_collection(
                collection_name=collection_name,
                vectors_config=models.VectorParams(
                    size=config.LOCAL_EMBEDDING_MODEL_DIMENSION,
                    distance=models.Distance.COSINE
                ),
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="employee_id",
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="full_name",
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="job_title",
```

```
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="department",
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="location",
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="email",
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )
```

```

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="start_date",
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="client_satisfaction",
                field_schema=models.PayloadIndexParams(
                    index_type=models.PayloadIndexType.INTEGER
                )
            )

            qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name="project_history",
                field_schema=models.TextIndexParams(
                    tokenizer=[models.TokenizerType.WORD, models.TokenizerType.WHITESPACE],
                    min_token_len=2
                )
            )

        point_structs = [models.PointStruct(**point) for point in points]

        qdrant_client.upsert(
            collection_name=collection_name,
            points=point_structs,
            wait=True
```

```
        )

        upserted_ids = [int(p.payload["employee_id"]) for p in point_structs]
        update_employee_status_in_db(upserted_ids, table_name, ProcessingStatus.COMPLETED)

        logger.info(f"[UPSERT TASK SUCCESS] Successfully upserted {len(points)} points to '{collection_name}'.")
        return {"status": "success", "count": len(points), "collection_name": collection_name}

    except Exception as e:
        logger.error(f"[UPSERT TASK RETRY] Upsert failed for collection '{collection_name}': {e}", exc_info=True)
        try:
            raise self.retry(exc=e)
        except MaxRetriesExceededError:
            logger.error(f"Upsert task for {collection_name} has failed permanently. Marking records as FAILED.")
            if points:
                failed_ids = [int(p["payload"]["employee_id"]) for p in points]
                update_employee_status_in_db(failed_ids, table_name, ProcessingStatus.FAILED)
            raise e
```

5.2 Back-end Implementation

5.2.1 Core Functionality Set-up

The backend's core logic is implemented through a set of Celery tasks and helper functions. The `prompt_templates.py` file defines the system and user prompts used to instruct the LLM for both answer generation and filter extraction. These prompts ensure the LLM behaves as an expert HR assistant and returns structured data when needed.

Below is the complete content of the `prompt_templates.py` file:

```python
def get_employee_qa_prompt(context: str, query: str) -> tuple[str, str]:
    system_prompt = (
        "You are an expert HR assistant. Your task is to answer questions about employees "
        "based on the internal profile data provided as context. Synthesize information from different profiles if needed. "
        "If you list employees, provide their full name and job title. Be friendly and professional."
    )
    user_prompt = (
        f"""Based ONLY on the context containing employee profiles below, provide a clear and concise answer to the user's query.
        If the context does not contain the answer, you MUST state that you could not find the information. Do not use outside knowledge.

        ## Context ##
        {context}

        ## User Query ##
        "{query}"
        """
    )
    return system_prompt, user_prompt


def get_filter_extraction_prompt(query: str) -> tuple[str, str]:
    system_prompt = """
```

```
    You are an expert at analyzing user queries and extracting structured search filters.
    Your output MUST be a single, valid JSON object and nothing else.
    The valid keys for the JSON object are: "department" (string), "job_title" (string), "min_satisfaction" (integer), "employee_id" (string), "full_name" (string), "location" (string), "email" (string), "start_date" (string), "skills" (string), and "project_history" (string).
    If the user's query does not contain any information for a filter, do not include the key in the JSON.
    If no filters are found at all, return an empty JSON object {}.
    """

    user_prompt = f"Analyze the following user query and extract the filters:\n\n'{query}'"

    return system_prompt, user_prompt
```

The core logic for filter extraction and Qdrant querying is handled in the `context_layer.py` module. It uses an LLM to extract structured filters from the query and then applies them as a `query_filter` during the vector search, improving retrieval precision.

Below is the complete content of the `context_layer.py` file:

```
import logging
import config
import re
import json
from qdrant_client import QdrantClient
from qdrant_client.http import models
from sentence_transformers import SentenceTransformer
from openai import OpenAI
from tenacity import retry, stop_after_attempt, wait_fixed

```

```python
from backend.prompt_templates import get_filter_extraction_prompt

logger = logging.getLogger(config.APP_NAME)

# Initialize Sentence Transformer Model
try:
    embedding_model = SentenceTransformer(config.LOCAL_EMBEDDING_MODEL_NAME)
    logger.info(f"Successfully loaded sentence-transformer model in context_layer: {config.LOCAL_EMBEDDING_MODEL_NAME}")
except Exception as e:
    logger.critical(f"Failed to load sentence-transformer model in context_layer: {e}", exc_info=True)
    embedding_model = None

relevance_threshold = 0.70

# LLM Helper for Filter Extraction
@retry(stop=stop_after_attempt(2), wait=wait_fixed(1))
def get_filters_from_llm(query: str) -> dict:
    """
    Uses an LLM to analyze the user's query and extract structured filters as a JSON object.
    """
    try:
        client = OpenAI(api_key=config.MISTRAL_API_KEY, base_url="https://api.mistral.ai/v1")

        system_prompt, user_prompt = get_filter_extraction_prompt(query)

        response = client.chat.completions.create(
            model=config.MISTRAL_MODEL, #
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": user_prompt}
```

```
            ],
            temperature=0.0,
            response_format={"type": "json_object"}
        )

        response_text = response.choices[0].message.content
        return json.loads(response_text)

    except Exception as e:
        logger.error(f"LLM filter extraction failed: {e}", exc_info=True)
        return {}

# Qdrant Search Function
def query_qdrant(query: str, client: QdrantClient, collection_name: str, top_k=5, query_filter=None) -> list[dict]:
    """
    Queries a Qdrant collection using a locally generated embedding and an optional filter.
    """
    if not embedding_model:
        logger.error("Embedding model is not available. Cannot query Qdrant.")
        return []

    try:
        query_embedding = embedding_model.encode(query, show_progress_bar=False).tolist()

        search_responses = client.search(
            collection_name=collection_name,
            query_vector=query_embedding,
            query_filter=query_filter,
            limit=top_k,
            with_payload=True,
            score_threshold=relevance_threshold
```

```
        )

        return [{"text": match.payload["text"], "score": match.score}↩
            for match in search_responses]

    except Exception as e:
        logger.error(f"Qdrant query failed in collection '{↩
            collection_name}': {e}", exc_info=True)
        return []

# Context Retrieval Handling
def get_context_for_query(query: str, client: QdrantClient, ↩
    collection_name: str) -> str:
    """
    Orchestrates the process of analyzing a query, building filters, ↩
        and retrieving context.
    """
    extracted_filters = get_filters_from_llm(query)
    logger.info(f"Extracted filters from LLM for collection '{↩
        collection_name}': {extracted_filters}")

    search_filter_conditions = []

    if "min_satisfaction" in extracted_filters and isinstance(↩
        extracted_filters["min_satisfaction"], int):
        search_filter_conditions.append(
            models.FieldCondition(key="client_satisfaction", range=↩
                models.Range(gte=extracted_filters["min_satisfaction"↩
                ]))
        )

    if "department" in extracted_filters and isinstance(↩
        extracted_filters["department"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="department", match=models.↩
                MatchValue(value=extracted_filters["department"]))
```

```
        )

    if "employee_id" in extracted_filters and isinstance(extracted_filters["employee_id"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="employee_id", match=models.MatchValue(value=extracted_filters["employee_id"]))
        )

    if "job_title" in extracted_filters and isinstance(extracted_filters["job_title"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="job_title", match=models.MatchText(text=extracted_filters["job_title"]))
        )

    if "full_name" in extracted_filters and isinstance(extracted_filters["full_name"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="full_name", match=models.MatchText(text=extracted_filters["full_name"]))
        )

    if "location" in extracted_filters and isinstance(extracted_filters["location"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="location", match=models.MatchText(text=extracted_filters["location"]))
        )

    if "email" in extracted_filters and isinstance(extracted_filters["email"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="email", match=models.MatchText(text=extracted_filters["email"]))
        )
```

```

    if "start_date" in extracted_filters and isinstance(extracted_filters["start_date"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="start_date", match=models.MatchText(text=extracted_filters["start_date"]))
        )

    if "skills" in extracted_filters and isinstance(extracted_filters["skills"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="skills", match=models.MatchText(text=extracted_filters["skills"]))
        )

    if "project_history" in extracted_filters and isinstance(extracted_filters["project_history"], str):
        search_filter_conditions.append(
            models.FieldCondition(key="project_history", match=models.MatchText(text=extracted_filters["project_history"]))
        )

    final_filter = models.Filter(must=search_filter_conditions) if search_filter_conditions else None

    qdrant_results = query_qdrant(query, client, collection_name, query_filter=final_filter)

    if not qdrant_results:
        logger.warning("No results returned from Qdrant for query: '%s' in collection '%s'", query, collection_name)
        return ""

    relevant_passages = [res["text"] for res in qdrant_results]

    if not relevant_passages:
```

```
        logger.warning(
            f"No results in collection '{collection_name}' met the relevance threshold of {relevance_threshold}. "
            f"Discarding context."
        )
        return ""

    logger.info(
        f"Retrieved {len(relevant_passages)} relevant passages from collection '{collection_name}' for the context."
    )
    return "\n---\n".join(relevant_passages)
```

5.2.2 Functionality Implementation

The core backend functionalities are executed asynchronously by Celery workers, as defined in the `query_service.py` and `chatlog_storage.py` files. The `process_employee_query_task` is the workhorse for retrieving context from each department's Qdrant collection, while `store_batch_chat_logs_task` handles the asynchronous archival of conversation logs.

Below is the complete content of the `query_service.py` file, which defines the query processing task:

```
import logging
import config
from qdrant_client import QdrantClient
from backend import context_layer

logger = logging.getLogger(config.APP_NAME)

# Celery App Import
try:
    from celery_app import celery_app
```

```
    logger.info("celery_app has been successfully imported from Celery")

except ImportError as e:
    logger.exception("An exception has occured when importing celery_app instance")
    raise

@celery_app.task(bind=True, max_retries=3, acks_late=True)
def process_employee_query_task(self, query: str, target_department: str):
    task_id = self.request.id
    logger.info(f"[CONTEXT TASK START] Task ID: {task_id}. Dept: '{target_department}', Query: '{query}'")

    try:
        client = QdrantClient(host=config.QDRANT_HOST, port=config.QDRANT_PORT)

        dept_info = config.DEPARTMENT_CONFIG.get(target_department)
        if not dept_info:
            logger.warning(f"No config found for department '{target_department}'. Skipping.")
            return {"status": "skipped_no_config", "context": None, "source": target_department}

        collection_name = dept_info['qdrant_collection']

        try:
            client.get_collection(collection_name=collection_name)
        except Exception:
            logger.warning(f"Collection '{collection_name}' not found. Skipping department '{target_department}'.")
            return {"status": "skipped_no_collection", "context": None, "source": target_department}

```

```
        context_from_dept = context_layer.get_context_for_query(query, client, collection_name)

        if context_from_dept:
            logger.info(f"[CONTEXT TASK SUCCESS] Task ID: {task_id}. Found context in '{target_department}'.")
            return {"status": "success", "context": context_from_dept, "source": target_department}
        else:
            logger.info(f"[CONTEXT TASK SUCCESS] Task ID: {task_id}. No relevant context found in '{target_department}'.")
            return {"status": "no_context", "context": None, "source": target_department}

    except Exception as e:
        logger.error(f"Unhandled exception in context task {task_id} for department {target_department}: {e}", exc_info=True)
        return {"status": "failed", "context": None, "source": target_department, "error": str(e)}
```

Below is the complete content of the `chatlog_storage.py` file, which manages the chat log buffering and storage tasks:

```
import boto3
import json
import gzip
import uuid
import config
import redis
import io
import time
from botocore.exceptions import ClientError
from functools import lru_cache
import logging

```

```
logger = logging.getLogger(config.APP_NAME)

try:
    from celery_app import celery_app
    logger.info("celery_app imported successfully in chatlog_storage.")
except ImportError:
    logger.error("Could not import celery_app in chatlog_storage.")
    celery_app = None  # Allow file to be imported without Celery for non-worker processes

# AWS and Redis config
bucket_name = config.AWS_BUCKET_NAME
try:
    log_buffer_threshold = int(config.LOG_BUFFER_THRESHOLD)
    log_buffer_ttl_seconds = int(config.LOG_BUFFER_TTL_SECONDS)
except (TypeError, ValueError):
    log_buffer_threshold = 100
    log_buffer_ttl_seconds = 3600

# Redis Initialization
redis_client = None
try:
    redis_client = redis.StrictRedis(
        host=config.REDIS_HOST, port=config.REDIS_PORT, db=config.REDIS_DB, decode_responses=False
    )
    redis_client.ping()
except Exception as e:
     logger.error(f"Redis connection failed: {e}. Chatlog buffering will likely fail.", exc_info=True)

@lru_cache(maxsize=1)
def get_s3_client():
```

```
    try:
        return boto3.client('s3')
    except Exception as e:
        logger.error(f"Failed to create S3 client: {e}", exc_info=True)
        return None

def get_utc_iso_timestamp():
     return time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime()) + "Z"

def generate_s3_key(chat_id):
    date_folder = time.strftime("%Y-%m-%d", time.gmtime())
    uid = uuid.uuid4().hex[:8]
    return f"chatlogs/{date_folder}/{chat_id}/batch_{int(time.time())}_{uid}.json.gz"

def compress_json_payload(log_entries):
    try:
        json_bytes = json.dumps(log_entries).encode('utf-8')
        out = io.BytesIO()
        with gzip.GzipFile(fileobj=out, mode="wb") as f:
            f.write(json_bytes)
        return out.getvalue()
    except Exception as e:
         logger.error(f"Error during JSON compression: {e}", exc_info=True)
         return None

if celery_app:
    @celery_app.task(bind=True, max_retries=3, default_retry_delay=60, acks_late=True)
    def store_batch_chat_logs_task(self, chat_id):
        if not redis_client:
            logger.error(f"Redis client not available. Cannot store logs for chat_id: {chat_id}")
            return {"status": "failed_no_redis", "chat_id": chat_id}
```

```

        redis_key = f"{config.REDIS_LOG_LIST_KEY_PREFIX}{chat_id}"

        try:
            log_entries_bytes = redis_client.lrange(redis_key, 0, -1)
            if not log_entries_bytes: return {"status": "no_logs_found", "chat_id": chat_id}

            log_entries = [json.loads(entry.decode('utf-8')) for entry in log_entries_bytes]
            if not log_entries:
                try: redis_client.delete(redis_key)
                except Exception as del_e: logger.error(f"Redis error on cleanup: {del_e}")
                return {"status": "decoding_failed", "chat_id": chat_id}

            compressed_payload = compress_json_payload(log_entries)
            if not compressed_payload: raise ValueError("Compression failed")

            s3_key = generate_s3_key(chat_id)
            s3 = get_s3_client()
            if not s3: raise ConnectionError("S3 client unavailable.")

            s3.put_object(Bucket=bucket_name, Key=s3_key, Body=compressed_payload, ContentType='application/json', ContentEncoding='gzip')

            try: redis_client.delete(redis_key)
            except Exception as del_e: logger.error(f"Redis key deletion failed after S3 upload: {del_e}")

            return {"status": "success", "s3_path": f"s3://{bucket_name}/{s3_key}", "count": len(log_entries)}
```

```python
        except Exception as e:
            logger.error(f"Error in store_batch_chat_logs_task for {chat_id}: {e}", exc_info=True)
            raise self.retry(exc=e)

def buffer_chat_log(chat_id, user, message):
    if not redis_client:
        logger.warning(f"Redis not available. Cannot buffer log for chat_id: {chat_id}")
        return

    log_entry = {'timestamp': get_utc_iso_timestamp(), 'user': user, 'message': message}
    key = f"{config.REDIS_LOG_LIST_KEY_PREFIX}{chat_id}"

    try:
        entry_bytes = json.dumps(log_entry).encode('utf-8')
        pipeline = redis_client.pipeline()
        pipeline.rpush(key, entry_bytes)
        pipeline.expire(key, log_buffer_ttl_seconds)
        results = pipeline.execute()
        current_length = results[0]

        if current_length >= log_buffer_threshold and celery_app:
            logger.info(f"Log buffer threshold reached for {chat_id}. Triggering S3 storage task.")
            store_batch_chat_logs_task.delay(chat_id)
    except Exception as e:
        logger.error(f"Error buffering chat log for {chat_id}: {e}", exc_info=True)
```

The system also includes an administrative script for triggering manual tasks, as shown in `main.py` below:

```python
import argparse
```

```python
import logging
import sys
import os

project_root = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))
if project_root not in sys.path:
    sys.path.insert(0, project_root)

import config
from backend import text_embedding
from backend import db_pool_setup

try:
    if hasattr(config, 'setup_logging'):
        config.setup_logging()
    logger = logging.getLogger(config.APP_NAME)
except Exception:
    logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
    logger = logging.getLogger(__name__)
    logger.warning("Resorted to basic logging configuration.")

def run_embedding_generation():
    logger.info("Attempting to launch embedding generation tasks for all departments...")

    db_pool_setup.initialize_pool()

    try:
        text_embedding.launch_embedding_tasks()
        logger.info("Embedding generation tasks successfully dispatched. Check Celery worker logs for progress.")

    except Exception as e:
        logger.error(f"Failed to launch embedding generation tasks: {
```

```
            e}", exc_info=True)

    finally:
        logger.info("Closing database pool...")
        db_pool_setup.close_pool()
        logger.info("Database pool closed.")

def main():
    parser = argparse.ArgumentParser(description="Corporate Directory Admin CLI")
    parser.add_argument(
        "action",
        choices=["generate-embeddings"],
        help="The administrative action to perform."
    )
    args = parser.parse_args()

    if args.action == "generate-embeddings":
        run_embedding_generation()
    else:
        logger.error(f"Unknown action: {args.action}")
        parser.print_help()

if __name__ == "__main__":
    main()
```

5.3 Front-end Implementation

The frontend of the system is a conversational interface built using the Chainlit framework. It handles the user's interaction flow, displays task progress, and manages communication with the backend. Key functionalities include a user-friendly chat window, a live task list, and the ability to select different LLM models for generating answers.

The complete code for the Chainlit application logic is provided in `chainlit_app.py` below:

```python
import sys
import os
import logging
import time

import chainlit as cl
from chainlit.element import TaskList, Task
from chainlit.message import Message
from celery import group
from celery.result import GroupResult, AsyncResult

project_root = None
try:
    project_root = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))
    if project_root not in sys.path:
        sys.path.insert(0, project_root)
except Exception as e:
    logging.basicConfig(level=logging.ERROR)
    logging.error(f"CRITICAL: Failed to set project root path: {e}", exc_info=True)
    raise

# Logging Configuration
try:
    import config
    logger = logging.getLogger(config.APP_NAME)
    if hasattr(config, 'setup_logging') and callable(config.setup_logging):
        config.setup_logging()
        logger.info(f"Logging configured. Project root '{project_root}' in sys.path.")
```

```
    else:
        logger.warning(f"config.setup_logging() not found/callable.")
except Exception as e:
    logger = logging.getLogger(__name__)
    logging.basicConfig(level=logging.INFO)
    logger.error(f"Error during config/logging setup: {e}. Using basic logger.", exc_info=True)

from backend import prompt_templates
from openai import OpenAI
from tenacity import retry, stop_after_attempt, wait_fixed

# Import Celery
try:
    from celery_app import celery_app
    from backend.query_service import process_employee_query_task
    from backend.chatlog_storage import buffer_chat_log
    logger.info("Successfully imported application modules for IntelChat.")
except ImportError as e:
    logger.error("ImportError while loading application modules.", exc_info=True)
    raise

def initialize_selected_llm(model_choice):
    safe_model_choice = model_choice.strip()
    try:
        if safe_model_choice == config.MISTRAL_MODEL_CHOICE.strip():
            return OpenAI(api_key=config.MISTRAL_API_KEY, base_url="https://api.mistral.ai/v1")

        elif safe_model_choice == config.OPENAI_MODEL_CHOICE.strip():
            return OpenAI(api_key=config.OPENAI_API_KEY)

        elif safe_model_choice == config.DEEPSEEK_MODEL_CHOICE.strip():
```

```
            return OpenAI(base_url=config.DEEPSEEK_BASEURL, api_key=config.DEEPSEEK_API_KEY)

        else:
            raise ValueError(f"Unsupported model: '{safe_model_choice}")
    except Exception as e:
        logger.critical(f"LLM Client initialization failed: {e}", exc_info=True)
        return None

@retry(stop=stop_after_attempt(3), wait=wait_fixed(2))
def call_llm_chat(client, system_prompt, user_prompt, model_name: str):
    try:
        response = client.chat.completions.create(
            model=model_name,
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": user_prompt}]
        )
        return response.choices[0].message.content
    except Exception as e:
        logger.error(f"LLM API call failed: {e}", exc_info=True)
        raise

async def await_group_task_result(result_obj: GroupResult, task_list_ui: cl.element.TaskList, timeout: int = 20):
    """
    Polls a Celery GroupResult object until it's ready, or until a timeout is reached.
    If a timeout occurs, it retrieves and returns results from only the completed tasks.
    """
    task_list_ui.tasks[0].status = cl.TaskStatus.RUNNING
    await task_list_ui.send()
```

```python

    start_time = time.time()

    while not result_obj.ready():
        if time.time() - start_time > timeout:
            logger.warning(
                f"Task group {result_obj.id} timed out after {timeout} seconds. "
                "Returning available results from completed tasks."
            )
            break
        await cl.sleep(1)

    retrieved_results = []
    if result_obj.children:
        for subtask_result in result_obj.children:
            # Check if the individual subtask has a result (i.e., it's ready)
            if isinstance(subtask_result, AsyncResult) and subtask_result.ready():
                try:
                    result = subtask_result.get(timeout=0)
                    retrieved_results.append(result)

                except Exception as e:
                    logger.error(f"Failed to retrieve result from a completed subtask: {e}", exc_info=True)

            elif isinstance(subtask_result, dict):
                retrieved_results.append(subtask_result)

    if result_obj.ready() and result_obj.successful():
        task_list_ui.tasks[0].status = cl.TaskStatus.DONE
    else:
        task_list_ui.tasks[0].status = cl.TaskStatus.FAILED
        task_list_ui.tasks[0].description = "Some tasks failed or 
```

```
timed out."
    await task_list_ui.send()

    return retrieved_results

@cl.on_chat_start
async def start_chat():
    cl.user_session.set("chat_id", f"chat_{int(time.time())}")
    cl.user_session.set("llm_choice", config.OPENAI_MODEL_CHOICE)
    await cl.Message(
        content="Welcome to IntelChat - the company's employee Q&A chatbot. What can I help you with?",
        author="IntelChat"
    ).send()

    ## CORRECTED ACTION SYNTAX
    await cl.Message(
        content="You can change the AI model at any time.",
        author="IntelChat",
        actions=[
            cl.Action(
                name="change_model",
                value="change_model",
                payload={"value": "change_model"},
                label="       Change Model"
            )
        ]
    ).send()

@cl.action_callback("change_model")
async def on_change_model(action: cl.Action):
    model_actions = [
        cl.Action(
            name="llm_selected",
            value=config.MISTRAL_MODEL_CHOICE,
            payload={"value": config.MISTRAL_MODEL_CHOICE},
```

```
            label=f"      {config.MISTRAL_MODEL_CHOICE}"
        ),
        cl.Action(
            name="llm_selected",
            value=config.OPENAI_MODEL_CHOICE,
            payload={"value": config.OPENAI_MODEL_CHOICE},
            label=f"      {config.OPENAI_MODEL_CHOICE}"
        ),
        cl.Action(
            name="llm_selected",
            value=config.DEEPSEEK_MODEL_CHOICE,
            payload={"value": config.DEEPSEEK_MODEL_CHOICE},
            label=f"      {config.DEEPSEEK_MODEL_CHOICE}"
        )
    ]
    await cl.Message(
        content="Please select your preferred AI model:",
        author="IntelChat",
        actions=model_actions
    ).send()

@cl.action_callback("llm_selected")
async def on_llm_selected(action: cl.Action):
    chosen_llm = action.payload.get('value')
    if chosen_llm:
        cl.user_session.set("llm_choice", chosen_llm)
        await cl.Message(
            content=f"Model has been set to **{chosen_llm}**. You can now ask me questions about our employees.",
            author="IntelChat"
        ).send()

@cl.on_message
async def main_logic(message: Message):
    query_text = message.content
    chat_id = cl.user_session.get("chat_id")
```

```python
    llm_choice = cl.user_session.get("llm_choice")
    buffer_chat_log(chat_id, "user", query_text)

    task_list = TaskList(tasks=[
        Task(title="Searching all departments for context...", status=cl.TaskStatus.RUNNING)
    ])
    await task_list.send()

    try:
        departments_to_query = config.DEPARTMENTS
        task_group = group(
            process_employee_query_task.s(query=query_text, target_department=dept)
            for dept in departments_to_query
        )

        group_result = task_group.apply_async()

        results = await await_group_task_result(group_result, task_list)

    except Exception as e:
        logger.error(f"Failed to dispatch Celery group task: {e}", exc_info=True)
        await task_list.remove()
        await cl.Message(content="Error: Could not connect to the processing service.").send()
        return

    all_context_fragments = []
    searched_sources = []
    for res in results:
        if isinstance(res, dict) and res.get('status') == 'success' and res.get('context'):
            all_context_fragments.append(f"Context from {res['source
```

```
']} Department\n{res['context']}")
        if isinstance(res, dict) and 'source' in res:
            searched_sources.append(res['source'])

    if not all_context_fragments:
        await cl.Message(content="I'm sorry, I could not find any relevant employee profiles for your query across any department.", author="IntelChat").send()
        await task_list.remove()
        return

    final_context = "\n\n---\n\n".join(all_context_fragments)
    system_prompt, user_prompt = prompt_templates.get_employee_qa_prompt(final_context, query_text)

    llm_client = initialize_selected_llm(llm_choice)

    model_to_use = None
    if llm_choice == config.MISTRAL_MODEL_CHOICE: model_to_use = config.MISTRAL_MODEL
    elif llm_choice == config.OPENAI_MODEL_CHOICE: model_to_use = config.OPENAI_MODEL
    elif llm_choice == config.DEEPSEEK_MODEL_CHOICE: model_to_use = config.DEEPSEEK_MODEL

    if llm_client and model_to_use:
        answer = call_llm_chat(llm_client, system_prompt, user_prompt, model_to_use)
    else:
        answer = "Error: Could not initialize the selected AI model to generate a final answer."

    await cl.Message(content=answer, author=llm_choice).send()

    if searched_sources:
        source_str = ", ".join(sorted(list(set(searched_sources))))
```

```
            await cl.Message(content=f"*(Searched in: {source_str})*", author="IntelChat", indent=1).send()

    buffer_chat_log(chat_id, "assistant", answer)
    await task_list.remove()
```

5.4 Source Code

The complete source code for the project is available on GitHub through this link: `https://github.com/MatrixNova/distribution_system-employee_Q-A`. You can refer to the repository for detailed setup instructions and deployment guidelines.

Chapter 6

Conclusions

In summary, the development of our Employee Q&A System has been aimed at resolving the critical challenges of latency and responsiveness that are associated with distributed systems, especially when dealing with large, multi-source datasets. We have been able to implement a real-time information retrieval platform that provides a responsive and friendly user experience by utilizing a robust technology framework that includes Celery, Qdrant, Redis, PostgreSQL, and various LLMs.

To achieve real-time query processing, we utilised asynchronous task processing with Celery and Redis, which facilitates instantaneous messaging and provide a smooth and engaging user experience. These technologies ensure that messages are delivered and received in real-time, creating a dynamic and interactive chat environment.

In terms of security, we implemented JWT-based authentication to secure user sessions and ensure that only authorised users can access specific functionalities. For maintaining application stability and security, we employed thorough input validation, efficient error-handling strategies, and detailed logging. These practices prevent invalid data from entering the system, avoid potential application crashes, and safeguard against security vulnerabilities.

Additionally, the application features a modern UI/UX design that enhances user interaction and satisfaction. The UI/UX design focuses on ease of use, aesthetic appeal, and a seamless navigation experience, contributing to a positive overall user experience.

With regard to scalability and maintainability, we structured the application to ensure it can grow and adapt to future needs. This involves writing clean, modular code and implementing best practices that allow for straightforward updates and feature additions.

To support the real-time chat functionality, security features, and scalability requirements, we built an efficient system architecture. This includes a well-defined client-server model where the client-side handles the user interface and real-time updates via WebSockets, while the server-side manages user authentication, message storage, and overall system logic. This separation of concerns ensures that the system is both performance and easy to maintain.

Overall, the Employee Q&A System is a comprehensive solution that combines several technical features with a user-centric design. The project effectively addresses the core requirements of real-time communication, ensuring users can engage in instantaneous messaging.

6.1 Development Potential

Going forward, the project has multiple ways in which it could still be expanded on from the iteration being shown here. An easy way of expanding is to simply integrate more choices of LLMs for processing and answer generation, as our system is already structured to support multiple LLM clients from providers like Mistral, OpenAI, and DeepSeek. The User Interface, while perfectly serviceable using Chainlit, could be adapted into something more customised should future needs or proprietary issues arise. As for the actual inner workings of the systems, there will always be space for optimization whether it be in the asynchronous function calls or how our Celery workers handle and deliver each query. For instance, fine-tuning the dynamic routing logic in ProjectRouter or optimising the polling mechanism in await_group_task_result are potential future improvements. But as of now, these are all uncertain plans and the program is for all intents and purposes able to be considered feature complete.