A
Project Report
On

"Food Delivery Chatbot"

(AI-powered Agent System)

Prepared by
Nipun Shah
(PGP-AIML, November 2024 Batch)

A Report Submitted to

The University of Texas at Austin,

Great Lakes Institute of Management

For partial fulfillment of the requirements for the

Post Graduate Program in Artificial Intelligence & Machine Learning

Submitted on:

November 2025

Submitted to

The University of Texas at Austin
McCombs School of Business

GREAT LAKES EXECUTIVE LEARNING

CANDIDATE'S DECLARATION

I hereby declare that the project entitled **"Food Delivery Chatbot"** is my own work conducted under the capstone project module at great learning pgp-aiml program.

I further declare that to the best of my knowledge, the project for Post Graduate Program does not contain any part of the work, which has been submitted for the award of any degree either in this University or in other University without proper citation.

Nipun Shah
(PGP-AIML Nov'24 Batch)

Date: November 2025

ABSTRACT

With the rapid growth of online food delivery platforms, managing increasing customer queries has become a major operational challenge. Customers frequently seek updates about order status, delivery times, cancellations, and payment confirmations. Traditional human-based support systems often result in long waiting times, inconsistent responses, and high operational costs.

To address this, the FoodHub AI Chatbot project implements an AI Agent System that automates customer query handling using a Large Language Model (LLM) integrated with an SQL Agent and a conversational Chat Agent. The system retrieves accurate order information from a structured database and presents it in concise, polite, and human-friendly language. The chatbot maintains conversational context across multiple interactions, applies input and output guardrails to ensure safe interactions, and can escalate complex queries to human agents when necessary.

This project demonstrates how intelligent agent systems can bridge structured database systems with natural language interfaces, enabling efficient, scalable, and secure customer support automation. The implementation successfully handles order-related queries while maintaining privacy, preventing misuse, and providing a seamless conversational experience.

ACKNOWLEDGEMENT

This capstone project, *"Food-Hub AI Chatbot"*, has been an insightful and rewarding experience as part of the **PGP in Artificial Intelligence and Machine Learning** program at **Great Learning**.

I would like to express my sincere gratitude to the program mentors and academic team at Great Learning for their valuable guidance, course structure, and continuous support throughout the learning journey. Their insights on AI, LLMs, and agent systems greatly contributed to the successful completion of this project.

I also appreciate my peers for their collaboration, constructive discussions, and encouragement during the implementation phase.

Finally, I am thankful to **Great Learning** for providing a platform that enabled practical, hands-on exploration of real-world AI applications, which has significantly enhanced my technical understanding and problem-solving skills.

With sincere regards,

Nipun Shah

Table of Contents

Abstract .....

Acknowledgement .....

Chapter 1 Introduction 1

1.1 Project Overview 1

1.2 Business Context 1

1.3 Key Terminologies 2

1.4 Objectives 2

1.5 Scope of Work 2

1.6 Tools & Technologies 3

Chapter 2 Theoritical Background 4

2.1 Large Language Models 4

2.2 AI Agent Systems 5

2.3 Sql Agents 6

2.4 Prompt Engineering 7

2.5 Tools 7

2.6 Conversation Memory 8

Chapter 3 System Design 9

3.1 System Overview 8

3.2 Architecture Flow 8

3.3 System components 10

3.4 Key Design Insights 11

3.5 Components Interactions 12

3.6 System Design 13

Chapter 4 Methodology 15

4.1 Overview 15

4.2 LLM Configuration 15

4.3 Question Answering LLM 16

4.4 Building SQL Agent 16

4.5 Building Chat Agent 17

4.6 Summary 18
Chapter 5 Observation 19
5.1 LLM Behavior 19
5.2 SQL Agent Insights 19
5.3 Chat Agent Insights 19
5.4 Practical Tips 19
5.5 Key Learnings 20
Chapter 6 Challenges 21
Chapter 7 Future Scope 23
References 24
Experience 25

Chapter 1
INTRODUCTION

1.1 Project Overview

In the modern era of digital transformation, online food delivery platforms have become an integral part of urban living. As the number of daily orders increases, so do customer queries related to delivery status, cancellations, payments, and refunds. Handling these interactions manually often leads to long response times, inconsistent communication, and higher operational costs.

FoodHub, a food aggregator company, aims to improve customer support by automating order-related queries. The project builds an AI-powered chatbot that uses Large Language Models (LLMs) and SQL agents to automate customer support interactions.

The *FoodHub AI Chatbot* project aims to address these challenges by introducing an **AI-powered conversational assistant** that can understand natural language queries and fetch real-time order information directly from the company's structured database. This system combines **Large Language Models (LLMs)** and **SQL Agent frameworks** to automate query resolution and provide customers with quick, accurate, and human-friendly responses.

Having the database absolutely and accurately mapped ability you can offer your clients the latest content and the high quality deals handy on the market. A database that is not accurately cleaned and continuously mapped, might cause to miss the latest additions provided by suppliers, book wrongly allocated hotels and increase engine's response time.

1.2 Business Context

Online food delivery has grown in urban areas, driven by students, working professionals, and busy families. Customers frequently ask about delivery times, order status, payment details, and return/replacement policies. These queries are largely handled manually, leading to long wait times, inconsistent responses, and higher costs as volume increases.

1.3 Key Terminologies

- **LLM (Large Language Model)**: An advanced AI model capable of understanding and generating human-like language based on contextual cues.
- **AI Agent**: A reasoning entity that can interpret instructions, access tools (like databases), and produce intelligent actions or responses.
- **SQL Agent**: A specific type of AI agent that converts natural language queries into SQL commands to interact with structured databases.
- **Guardrails**: Predefined rules and filters ensuring that the AI behaves safely, ethically, and within defined business constraints.
- **Chat Agent**: An orchestrating agent that manages multiple tools (e.g., SQL query tool, response formatting tool) and maintains conversational context across interactions.
- **Tools**: Reusable functions that agents can invoke to perform specific tasks (e.g., database queries, response formatting).
- **Conversation Memory**: A mechanism that stores and retrieves previous interactions to maintain context across multiple turns in a conversation.
- **Prompt Engineering**: The practice of designing and refining input prompts to guide LLM behavior and improve response quality.

1.4 Objective

Design and implement a functional AI-powered chatbot that:

- Connects to the order database via an SQL agent
- Retrieves accurate order details
- Provides concise, polite, customer-friendly responses
- Applies input and output guardrails for safe interactions
- Prevents misuse and escalates complex queries to human agents when needed
- Improves efficiency and customer satisfaction
- Maintains conversational context across multiple interactions
- Handles multi-turn conversations seamlessly

1.5 Scope of Work

This report covers the complete implementation of the FoodHub AI Chatbot, from foundational LLM configuration through the full conversational agent system.

Key activities completed:

- Setting up and validating the base LLM using ChatGroq with appropriate parameters for different tasks
- Refining prompts and analyzing response clarity through iterative testing
- Integrating the SQL Agent with the FoodHub database using prefix-based prompt engineering
- Building the Chat Agent with tool orchestration (fetch_sql_data, format_response)
- Implementing conversation memory for multi-turn interactions and context retention
- Applying input and output guardrails for safe interactions and misuse prevention
- Implementing escalation logic for complex queries
- Verifying query accuracy, data consistency, and conversational flow
- Testing the complete chatbot system

1.6 Tools and Technologies

Category	**Tools / Frameworks**
LLM Provider	**Groq**
Framework	**LangChain**
Database	**SQLite (.db)**
Language	**Python**
IDE / Environment	**Google Colab, Jupyter Notebook**
Libraries	**langchain, sqlite3, numpy, pandas, ...**

Chapter 2
THEORITICAL BACKGROUND

2.1 Large Language Models

Large Language Models (LLMs) are AI models trained on massive text corpora to understand and generate human-like language. They predict the next word in a sequence based on context, enabling natural language understanding and generation.

Deterministic vs Stochastic Behavior:

- LLMs are probabilistic by default, meaning the same input can yield different outputs.
- **Temperature** controls randomness:
    - Temperature = 0: more deterministic outputs, suitable for SQL/data tasks
    - Temperature = 0.7-1.0: more creative, useful for open-end generation

Tip: For SQL/data tasks, use **temperature=0** to ensure deterministic, reliable results.

Max Tokens

- Limits response length. Too low truncates answers; too high wastes resources. Balanced settings ensure complete, concise outputs.

Text vs Chat Models

- **Text models (Groq):** Single string prompts – string completion
- **Chat models (ChatGroq):** Message-based interfaces with roles (system, user, assistant). Better for conversations and system instructions. Recommended for agent systems.

2.2 AI Agent Systems

AI agent systems combine an LLM with tools to accomplish tasks. The LLM acts as the orchestrator, understanding queries, selecting tools, interpreting outputs, and generating responses.

Why Agents Over RAG?

- **RAG (Retrieval-Augmented Generation):** For unstructured text (PDFs, documents, knowledge bases). Retrieves relevant chunks and summarizes
- **AI Agent Systems**: For structured data (databases). Translates natural language to structured queries and back.
  For FoodHub's structured order database, SQL agents are appropriate.

The Orchestration Flow:

1. User query → LLM interprets intent
2. LLM selects appropriate tool(s)
3. Tool executes (e.g., SQL query)
4. LLM interprets tool output
5. LLM generates human-friendly response

The LLM coordinates the process; tools perform the actions.

Types of Agents relevant to this project:

1. **Tool Agents:** Interface with external tools (e.g., APIs, calculator, …)
2. **SQL Agents:** Convert natural language queries into SQL, execute them, and return human-friendly results.

2.3 SQL Agents:

(Natural Language to Database Intelligence)

SQL Agents translate natural language queries into SQL, execute them, and format results as natural language responses.

SQL Agent Flow:

1. User provides natural language input.
2. Agent reasons about intent and constructs SQL queries.
3. SQL executes on the database.
4. Agent formats the output in a concise, polite, and human-readable manner.

Natural Language Query
↓
LLM Reasoning (understands intent)
↓
SQL Generation (constructs query)
↓
SQL Execution (database tool runs query)
↓
Result Interpretation (LLM processes raw output)
↓
Human-Friendly Response (formatted natural language)

2.4 Prompt Engineering: System Messages

System Messages in Agent Systems:

- **Standalone LLM:** System messages can be passed directly to llm to guide behavior
- **LLM within SQL Agent:** Do not pass system messages directly to the LLM. Use the SQL Agent's system_message or prefix parameter instead. SQL agents manage prompt construction internally (prefix/suffix templates), and direct system messages can conflict.

Why This Separation?

- SQL agents use internal prompt templates for SQL generation.
- External system messages can override or conflict with agent logic
- Agent-level prompts preserve SQL generation while adding domain guidance.

Best Practice

- Keep LLM configuration simple (model, temperature, max_tokens)
- Handle domain-specific behavior through the SQL Agent's *system_message*
- This separation maintains SQL capabilities while adding communication guidelines

Quick Tips / Good Practices

- Always use deterministic parameters (temperature=0) for database queries.
- Refine prompts with system messages to guide tone, clarity, and safety.
- Inspect database schema before building agents to avoid runtime errors.
- Modular coding ensures cleaner notebooks and easier report documentation.

2.5 Tools and Tool Orchestration

Tools in Agent Systems:

- Tools are reusable functions that agents can invoke to perform specific tasks (e.g., database queries, response formatting)
- Agents select tools based on user queries and tool descriptions
- Tool descriptions guide the agent's selection process

Tool Orchestration:

- Agents can chain multiple tools in sequence (e.g., fetch data → format response)
- Each tool receives input from the previous tool's output
- Tool sequencing enables complex workflows while keeping components modular

Benefits:

- Modularity: Each tool handles a specific task
- Reusability: Tools can be used across different agent configurations
- Separation of concerns: Data retrieval vs. response formatting

2.6 Conversation Memory and Context Retention

Why Memory Matters:

- Enables multi-turn conversations where users can reference previous context
- Maintains continuity across interactions (e.g., "What's the status?" after "Where is my order O12488?")
- Improves user experience by reducing repetitive information

ConversationBufferMemory:

- Stores complete conversation history (user queries and agent responses)
- Provides context to the agent for each new query
- Enables implicit references (e.g., pronouns, follow-up questions)

How It Works:

- Each interaction is appended to the conversation buffer
- The agent receives full conversation history with each new query
- Context is maintained throughout the session until explicitly reset

Example:
Query 1: "Where is my order O12488?" → Agent retrieves order details
Query 2: "What's the delivery time?" → Agent uses context to know this refers to O12488

Chapter 3

SYSTEM DESIGN

3.1 System Overview

The FoodHub AI Chatbot is an AI Agent System that automates customer query handling through a multi-layered architecture. The system integrates Large Language Models (LLMs) with structured database operations via an SQL Agent, orchestrated by a Chat Agent that manages tool execution and maintains conversational context.

The architecture consists of three main layers:

- **Service Layer**: Handles input validation, guardrails, and intent classification
- **Agent Layer**: Orchestrates tools (SQL query tool, response formatting tool) and maintains conversation memory
- **Data Layer**: SQL Agent connects to the order database to retrieve accurate information

The system processes natural language queries, retrieves relevant data from the database, and generates concise, polite, customer-friendly responses while maintaining context across multiple interactions and applying safety guardrails to prevent misuse.

3.2 Architecture Flow

User Query → Service Layer (Validation/Guardrails) → Agent Layer (Tool Orchestration) → Data Layer (SQL Agent → Database) → Response Formatting → Output Sanitization → User

Key Components in Flow:

1. **Service Layer**: Input validation, guardrails, intent classification, output sanitization
2. **Agent Layer**: Tool orchestration (fetch → format), conversation memory
3. **Data Layer**: SQL Agent, database connection, raw data retrieval
4. **LLM Layer**: Multiple LLMs (orchestrator, data retriever, reply generator)

Detailed Flow:

1. **User submits a query**
   - Example: "Where is my order O12488?"

2. **Service Layer: Input Validation & Guardrails**
   - `ChatService` validates input (non-empty, length checks)
   - Applies input guardrails (malicious pattern detection, SQL injection prevention)
   - Classifies intent (order query, general help, etc.)

3. **Agent Layer: Query Orchestration**
   - `ChatAgent` receives query with embedded prompt (legacy API approach)
   - Maintains conversation context via `ConversationBufferMemory`
   - Determines tool sequence based on query type

4. **Tool 1: Data Retrieval (`fetch_sql_data`)**
   - `SQLAgentManager` receives natural language query
   - SQL Agent (LLM) interprets query and constructs SQL
   - Ensures guardrails (WHERE clause with specific identifier, read-only)
   - `DatabaseClient` executes query and returns raw results
   - Returns structured data

5. **Tool 2: Response Formatting (`format_response`)**
   - Receives raw database results from Tool 1
   - Reply Generator LLM converts structured data to human-friendly text
   - Example: "Your order O12488 has been delivered at 1:00 PM."

6. **Service Layer: Output Sanitization**
   - `ChatService` applies output guardrails (sensitive data redaction, content filtering)

7. **Response to User**
   - Final formatted, sanitized response returned to user

3.3 System Components

Component	Role / Function
LLM (ChatGroq)	Multiple LLM instances with different roles: • Orchestrator LLM: Reasons about tool selection and workflow orchestration • Data Retriever LLM: Powers SQL Agent to convert NL queries into SQL • Reply Generator LLM: Converts raw database results into human-friendly responses
SQL Agent	Converts NL queries into SQL, executes queries, and formats responses.
Database (SQLite)	Stores structured order information such as order_id, cust_id, status, items, timestamps.

System Message	Provides behavior guidelines, polite tone, privacy rules, and guardrails for agent responses.
Guardrails	Ensures security, restricts data to the requested customer, and prevents destructive queries.
Chat Agent	Orchestrates tool execution, maintains conversation memory, and manages multi-turn interactions
Conversation Memory	Maintains context across multiple interactions using ConversationBufferMemory
ChatService	Service layer that handles input validation, guardrails, intent classification, and output sanitization

3.4 Key Design Insights

- **Separation of Concerns:** Three-layer architecture (Service, Agent, Data) enables modularity, easier debugging, and independent component testing.
- **Tool-Based Orchestration:** Chat Agent uses tools (`fetch_sql_data`, `format_response`) instead of monolithic logic, allowing flexible workflow and tool reuse.
- **Multiple LLM Roles:** Different LLMs for different tasks (orchestrator, data retriever, reply generator) optimize performance and cost - larger models for SQL reasoning, smaller models for formatting.
- **System Messages as Policy Enforcers**: System messages/prompts enforce behavior guidelines (politeness, privacy, guardrails) while agents handle technical reasoning (SQL generation, tool selection).
- **Conversation Memory**: `ConversationBufferMemory` maintains context across interactions, enabling natural multi-turn conversations without repeated information.
- **Guardrails at Service Layer:** Input/output guardrails at the service layer provide security before agent execution, preventing malicious queries and sensitive data leakage.
- **Factory Pattern for Dependency Injection**: `ChatServiceFactory` encapsulates complex initialization logic, making the system easier to configure and test.

- **Raw Data → Formatted Response Pipeline**: Two-step process (fetch raw data → format response) separates data retrieval from presentation, improving maintainability and allowing independent optimization.

3.5 Components Interactions

LLM Configuration Layer:

- `LLMManager` initializes and manages multiple LLM instances (orchestrator, data retriever, reply generator)
- Provides a single point for model selection, temperature, and token limits
- Enables reproducibility with fixed parameters per role

Data Layer:

- `DatabaseClient` manages database connection and provides query interfaceProvides a single point for model selection, temperature, and token limits
- `SQLAgentManager` wraps SQL Agent, coordinates with DatabaseClient, and returns structured results

Agent Layer:

- `ChatAgent` orchestrates tools (`fetch_sql_data`, `format_response`) and maintains conversation memory
- Tools interact with `SQLAgentManager` and LLMs for data retrieval and response formatting

Service Layer:

- `ChatService` coordinates the pipeline: input validation → agent execution → output sanitization
- `ChatServiceFactory` assembles all components via dependency injection

Integration Flow:

ChatServiceFactory → LLMManager + DatabaseClient → SQLAgentManager → ChatAgent (with tools) → ChatService → FoodDeliveryBot

3.6 HLD MindMap

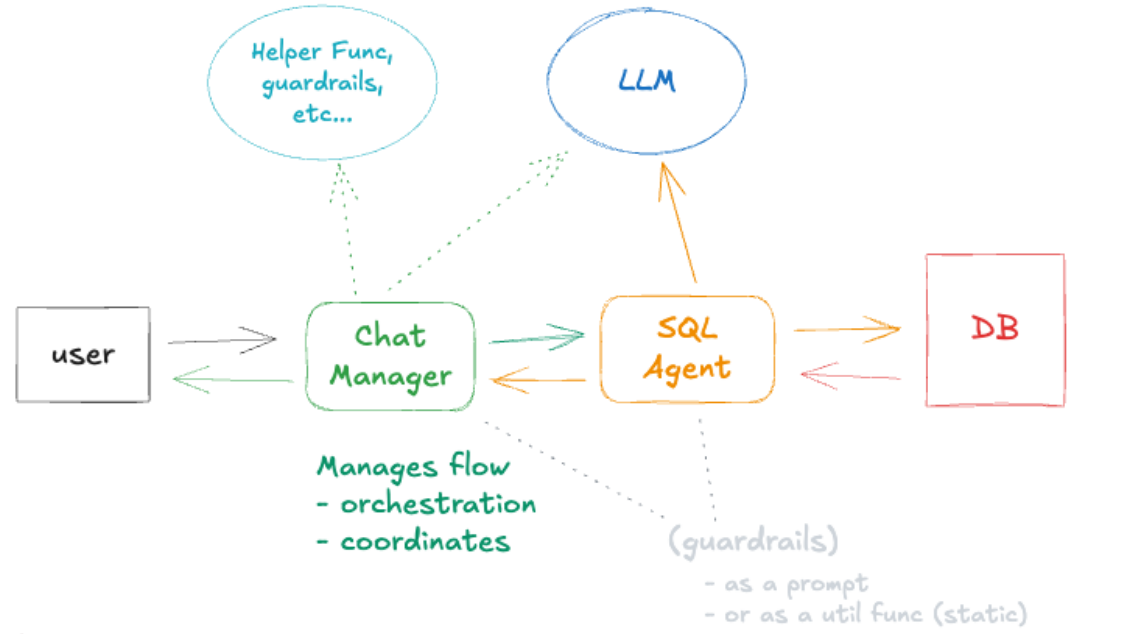

Figure 3.1 HLD Diagram1 (Minimal Architecture)

Flow Direction:

- **Request flow**: User → Service → Agent → Tools → Data Layer
- **Response flow**: Database → Tools → Agent → Service → User

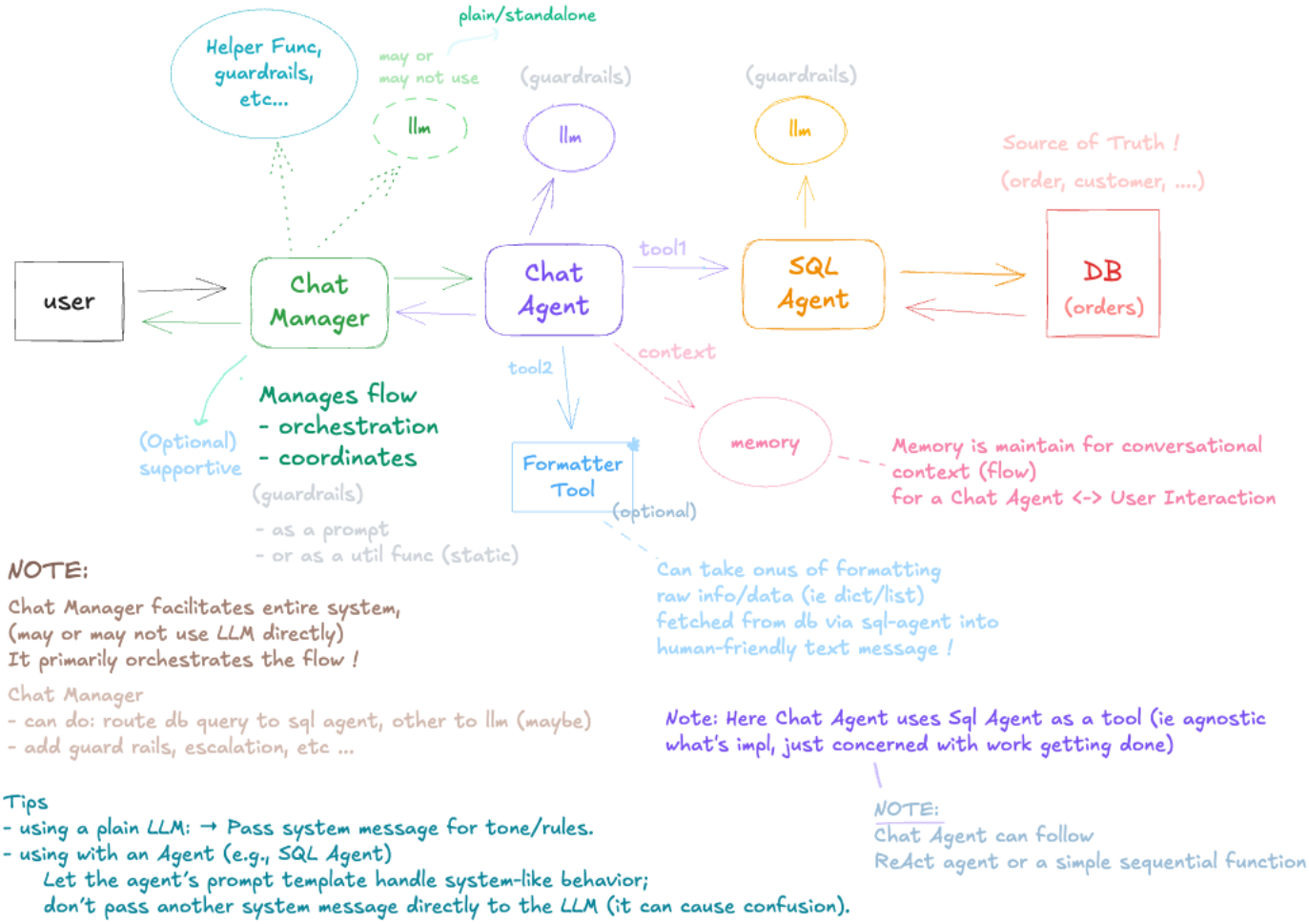

Figure 3.2 HLD Diagram 2 (Final Architecture)

Key Points:

- **User:** initiates queries through the main interface
- **ChatService**: validates input, applies guardrails, and routes queries
- **ChatAgent:** orchestrates tools and maintains conversation context
- **Tools**: execute specific tasks
- **SQLAgentManager:** converts natural language to SQL and queries the database
- **DatabaseClient:** manages the database connection
- **Database:** stores order information (source of truth)
- **LLMs:** power different components (orchestrator, data retriever, reply generator)
- **Memory:** maintains conversation history for context retention

Chapter 4
APPROACH

4.1 Overview

The implementation follows a modular approach using Python, LangChain, and Groq's LLM API. Separation of concerns across multiple layers (Service, Agent, Data) improves maintainability, testability, and extensibility. The system uses a factory pattern for dependency injection and tool-based architecture for flexible orchestration.

This section covers the complete implementation across all rubrics:

- Rubric 1: LLM Setup and Configuration
- Rubric 2: Question Answering and Prompt Refinement
- Rubric 3: SQL Agent Implementation
- Rubric 4: Chat Agent with Tool Orchestration
- Rubric 5: Complete Chatbot Integration with Guardrails and Memory

Each component is designed with modularity in mind, allowing independent testing and easy extension. The implementation emphasizes code reusability, clear separation between data retrieval and response formatting, and comprehensive guardrails for safe interactions.

4.2 LLM configuration

4.2.1 Library Imports and Environment Setup

Required libraries: LangChain Core, LangChain Groq, Pandas, SQLite3. Environment configuration loads GROQ_API_KEY from Google Colab's userdata and validates availability before proceeding.

4.2.2 Modular LLM Configuration

The system uses multiple LLM instances optimized for specific tasks:

Design Rationale:

- **Data Retriever (70B, temp=0)**: Larger model with deterministic output for accurate SQL generation
- **Orchestrator (70B, temp=0.3)**: Larger model with deterministic output for reasoning

- **Reply Generator (8B, temp=0.4)**: Balanced creativity for natural, conversational responses

Benefits:
- Task-specific optimization (accuracy vs. cost)
- Independent configuration per role
- Easy model swapping via `LLMManager`
- Reproducibility through fixed parameters

4.3: Question Answering LLM

Objective:

Evaluate the LLM's ability to understand natural language queries and refine responses through prompt engineering.

Approach:

- Provided sample queries to the LLM (general, not database-specific):
- Assessed clarity, accuracy, and relevance of answers.
- Refined prompts by adding contextual instructions to improve output quality.

Observations / Tips:

- Prompt refinement significantly improved clarity and relevance.
- System messages can guide tone and enforce business rules in later phases.

4.4: Building SQL Agent

Objective:

Connect the LLM with the order database using a SQL Agent and retrieve accurate, human-friendly responses.

Approach:

1. **Database Loading:**
    - Used SQLDatabase from LangChain to load the SQLite .db file.
    - Conducted basic sanity checks (row count, columns, sample data).

2. **SQL Agent Setup:**
    - Created agent using create_sql_agent() with ChatGroq LLM and **system message** to enforce guardrails.
    - Guardrails ensured customer-specific responses, polite messaging, and safe query execution.
3. **Testing Queries:**
    - Example: “Show all details for order ID 1001”
    - Compared agent response with manual SQL query to verify accuracy.

The SQLManager class provides modular agent management:

Key principle: System message passed via agent parameter (not LLM) to preserve default SQL generation while adding domain context.

4.5 Building Chat Agent

Objective: Orchestrate tools and maintain conversational context to enable multi-turn interactions with accurate, human-friendly responses.

Approach:

1. **Tool Creation**:
- Created `fetch_sql_data` tool wrapping `SQLAgentManager` to retrieve raw database results
- Created `format_response` tool using Reply Generator LLM to convert raw data into customer-friendly responses
- Tools return structured data format: `{"answer": "...", "output": [...], "input": "..."}`

2. **Chat Agent Setup**:
- Used `initialize_agent()` with `CHAT_CONVERSATIONAL_REACT_DESCRIPTION` agent type (legacy API)
- Integrated `ConversationBufferMemory` to maintain conversation history
- Embedded system prompt directly in user query (legacy API approach)
- Configured tool sequence: `fetch_sql_data` → `format_response`

3. **Memory Integration**:
- `ConversationBufferMemory` stores complete conversation history
- Enables context retention across multiple queries (e.g., "What's the status?" after "Where is my order O12488?")
- Memory cleared on session reset

4. **Testing Workflow**:
   - Example: "Where is my order O12488?" → Agent calls `fetch_sql_data` → Calls `format_response` → Returns formatted response
   - Verified multi-turn conversations maintain context
   - Tested tool sequencing and error handling

The ChatAgent class provides modular agent orchestration

Key principle: System prompt embedded in user query (legacy API) rather than using `system_message` parameter. Tools are orchestrated sequentially, and conversation memory enables natural multi-turn interactions without repeated information.

4.6 Summary of Methodology

Rubric	Key Focus	Outcome / Observation
1	LLM Setup	Deterministic, reproducible configuration; test queries executed successfully
2	LLM QA	Prompt refinement improved clarity and relevance; system messages optional at this stage
3	SQL Agent	Connected LLM with database; retrieved accurate, polite responses; guardrails enforced privacy
4	Chat Agent	Successful tool orchestration (fetch → format); conversation memory enabled context retention; multi-turn interactions working seamlessly

💡 Tips / Best Practices

- Always test LLM independently before integrating with SQL Agent.
- Refine prompts iteratively for clarity and correctness.
- Use **system messages** to enforce behavior and privacy.
- Modular code helps in scaling the pipeline for the final chatbot.

Chapter 5
OBSERVATION, NOTES & INSIGHTS

5.1 LLM Behavior

- Deterministic parameters (temperature=0) ensure consistent, repeatable outputs.
- Prompt refinement improves clarity, relevance, and user-friendliness of responses.
- LLM works effectively with generic QA even before connecting to database.

5.2 SQL Agent Insights

- Successfully translates natural language queries to SQL and returns human-readable results.
- System message is critical to enforce privacy, polite tone, and guardrails.
- Modular separation of LLM setup, agent logic, and database access improves maintainability.

5.3 Chat Agent Insights:

- Tool-based architecture enables modular, reusable components and flexible workflow orchestration.
- Conversation memory maintains context across interactions, enabling natural multi-turn conversations without repeated information.
- Sequential tool execution (fetch → format) separates data retrieval from presentation, improving maintainability and allowing independent optimization.
- Multiple LLM instances (orchestrator, data retriever, reply generator) optimize performance and cost for specific tasks.
- Legacy API approach (embedded prompt) works effectively for tool orchestration, though modern API offers better memory management.

5.4 Practical Tips

- Test LLM independently before integrating with SQL Agent.
- Inspect database schema prior to building queries to avoid runtime errors.
- Always handle missing or ambiguous data gracefully using system messages.

- Keep code modular for scalability and cleaner notebook/report documentation (e.g., `LLMConfig`, `SQLManager`, `ChatAgent` classes).
- Use factory pattern (`ChatServiceFactory`) for dependency injection and easier testing.
- Implement guardrails at service layer before agent execution for better security.
- Test tool sequencing and error handling to ensure robust workflow.
- Verify conversation memory by testing multi-turn interactions with implicit references.

5.5 Key Learnings

Insight Category	Learning	Impact
Temperature	0 ensures deterministic SQL	Reduced query errors
System Prompt	Agent-level > LLM-level	Preserves SQL generation
WHERE Clauses	Mandatory for privacy	Prevents data leakage
Prompt Refinement	Clear constraints improve output	80-93% verbosity reduction

Chapter 6
CHALLENGES

During the intermediate phase of the FoodHub AI Chatbot project, the following challenges were encountered:

Library Version Conflicts

- Certain versions of LangChain and ChatGroq libraries in Google Colab were incompatible.
- **Solution:** Explicitly installed compatible versions using pip and validated functionality before proceeding.

Ensuring Reproducibility

- LLM responses could vary across runs if parameters were not fixed.
- **Solution:** Set deterministic parameters (temperature=0) and fixed API keys to maintain consistent outputs.

System Message Design

- Designing a system message that enforced privacy, polite tone, and correct SQL query behavior required iterative testing.
- **Solution:** Created concise guardrails focusing on customer-specific queries and graceful handling of missing or ambiguous data.

Legacy API Limitations

- Using `initialize_agent` with legacy API required embedding system prompt in user query instead of using `system_message` parameter.
- Solution: Created `_prepare_prompt()` method to embed system instructions directly in the query string.

Error Handling in ToolChain

- Handling errors at different stages (SQL query failures, formatting errors, tool execution failures) while maintaining user-friendly responses.
- Solution: Implemented try-catch blocks at each layer and returned structured error messages that tools can handle gracefully.

Key Insight:

Resolving these challenges enhanced understanding of real-world AI system constraints, emphasizing careful environment setup, modular code design, and safe interaction with databases.

Understand environment dependencies, follow agent configuration best practices, and implement verification frameworks. Solutions implemented ensure reliability and maintainability for future phases.

Chapter 7
FUTURE SCOPE

Completed in Current Implementation:

- Multi-turn conversations with memory (implemented)
- Human escalation logic (implemented)

Future Enhancements:

- Real-time Integration: Connect with FoodHub app/API for live customer queries
- Advanced Analytics: Order trends, customer behavior patterns, and support metrics
- Web Interface: Deploy via Streamlit or web interface for user-friendly interaction
- Performance Optimization: Caching for frequent queries, rate limiting, response time optimization
- Multi-language Support: Handle queries in multiple languages
- External Service Integration: Payment gateways, delivery tracking APIs, notification services

Key Focus:

- Production deployment, scalability, enhanced user experience, and integration with existing FoodHub infrastructure.

REFERENCES

1. LangChain Documentation- https://www.langchain.com/docs
2. ChatGroq API Documentation – https://groq.com/chatgroq
3. SQLite Documentation – https://www.sqlite.org/docs.html

EXPERIENCE / OUTCOME

Working on the *FoodHub AI Chatbot* as part of the **PGP in Artificial Intelligence and Machine Learning (Great Learning)** was a highly enriching experience. This project allowed me to apply theoretical knowledge from the course into a **practical, real-world AI problem**, bridging the gap between learning and application.

Unlike previous academic projects, this capstone involved **hands-on work with live tools and APIs**, including ChatGroq LLM, LangChain, and SQLite databases. I gained experience in **modular code design, prompt engineering, SQL agent integration, and guardrail implementation**, which are critical for building production-ready AI systems.

Throughout the project, I learned the importance of **structured planning, iterative testing, and troubleshooting**, for example, resolving library version conflicts in Colab, ensuring reproducibility of LLM responses, and designing effective system messages for customer safety and data privacy.

This experience significantly enhanced my **practical problem-solving skills, professional coding practices, and understanding of AI agent systems**. It also provided insights into **real-world deployment considerations**, such as modular architecture, clean code, and safe human-AI interaction design.