CSCI 370 Final Report

AskQL

Kevin Bamwisho
Ryker Phelps
Sang Bui
Thai Tran

June 20, 2026

Qualcomm

CSCI 370 Summer 2026

Advisor: Dr. Donna Bodeau

Table 1: Revision history

Revision	Date	Comments
New	May 18, 2026	Xi. Team Profiles
Rev – 2	May 20, 2026	I. Introduction II. Functional Requirements III. Non-functional Requirements IV. Risks V. Definition of Done
Rev – 3	May 28, 2026	VI. Systems Architecture
Rev - 4	Jun 3, 2026	VII. Software Test and Quality VIII. Project Ethical Considerations
Rev - 5	Jun 12, 2026	IX. Working Document Refactor A. Risks B. System Architecture C. Project Ethical Considerations D. Team Profile X. Project Completion Status XI. Future Work XII. Lessons Learned
Rev - 6	Jun 20, 2026	Reviewed all sections for corrections, tense, formatting, and grammar.

Table of Contents

I. Introduction....4
Project Overview.... 4
Client....4
Existing Software....4
Data Sources.... 4
Definitions, Acronyms, and Abbreviations.... 4
Stakeholders and Users....5
Maintenance Responsibility....5
II. Functional Requirements....5
III. Non-Functional Requirements....6
IV. Risks....7
1. Inaccurate Text-to-SQL Translation (AI Hallucination)....7
2. Multi-Database Conflicts....7
3. Dynamic Visualization Generation Failures....7
4. Frontend Development Capability Gap.... 8
5. Steep Learning Curve for Agentic Frameworks....8
6. SQL Injection and Security Vulnerabilities....8
7. Quantitative Evaluation Metrics for Evaluability....8
Risk Level Matrix.... 9
V. Definition of Done....9
VI. System Architecture....11
Figure 1: System Architecture.... 11
Architecture Layers....12
VII. Software Test and Quality.... 14
1. Overview of Quality Assurance Methodology.... 14
2. Requirement-to-Test Mapping Matrix 15
3. Detailed Test and Quality Specifications....16
VIII. Project Ethical Considerations....20
Pertinent Principles:....20
Most Vulnerable Principles:....20
Michael Davis Tests Applied:....20
Software Quality Plan Impact:....20
Security considerations:....20
IX. Project Completion Status.... 22
Backend / Front End Development....22
Agent Development.... 23
RAG Development.... 24
MCP Development.... 25
X. Future Work.... 27
XI. Lessons Learned.... 28
Backend / Front End Development....28

Agent Development.......... 28
RAG Development.......... 29
MCP Development.......... 29
XII. Acknowledgments.......... 30
XIII. Team Profile.......... 31
References.......... 32
Appendix A – Key Terms.......... 32

I. Introduction

Project Overview

AskQL is a GenAI powered analytics chatbot built for Qualcomm's Test Base Station (TBS) engineering team. Engineers currently need SQL expertise and multiple tools to query operational data across heterogeneous databases. This project replaces that workflow with a single conversational interface: ask a question in plain English, get back a result as a table, a chart, and a natural language summary. The system connects to MySQL and Microsoft SQL via a connector framework, and uses a RAG pipeline to study internal schema and business terminology to generate accurate, safe, read-only queries. Qualcomm engineers will maintain the system after handoff.

Client

The Client is Qualcomm's TBS team. They support a global team with over 2,000 internal test base station systems used daily. Querying that data today requires writing raw SQL, switching between multiple database tools, and manually building visualizations. It takes an engineer who specializes in data analytics.

What the client needs is a single conversational interface that can route queries across multiple data sources, enforce guardrails, and show results in a format any engineer can understand.

Existing Software

Qualcomm has tools for single database querying. This project is not a revision of any existing software. The team built it from scratch. Their current tool does not use techniques like RAG, which improve accuracy and consistency. The goal was to go well beyond single database querying by supporting multiple databases simultaneously with a dedicated RAG layer.

Data Sources

The chat bot connects to Qualcomm's databases including MySQL and Microsoft SQL Server.

Definitions, Acronyms, and Abbreviations

Term	Definition
TBS	Test Base Station (Qualcomm's internal cellular test infrastructure)
NL	Natural Language
SQL	Structured Query Language (for querying relational databases)
RAG	Retrieval Augmented Generation (gives context to an LLM to improve accuracy)
LLM	Large Language Model (AI model that interprets questions)
MCP	Model Context Protocol (a standard interface for exposing tools and data to an LLM agent
RBAC	Role-Based Access Control (restricts data access based on user role)
AC	Acceptance Criteria (conditions a feature must meet to be considered complete)
UAT	User Acceptance Testing (end user validation before final delivery)

DOD	Definition of Done (the checklist that determines when the project is complete)
SSO	Single Sign On (authentication used in company systems)
Vector DB	Vector Database (stores embeddings used for semantic search in the RAG pipeline

Table 2: Definitions Table

Stakeholders and Users

The primary user is anyone in the Qualcomm TBS team who needs fast access to operational and performance data without writing SQL by hand. Users also include engineers who track system reliability and release quality metrics.

Maintenance Responsibility

Qualcomm TBS has ownership of the github repository project. The system was designed with extensibility in mind and documentation was produced as part of the definition of done to make handoff simple. TBS engineers will maintain the project and develop it further as needed.

II. Functional Requirements

The functional requirements define the components that had to be present for the client to consider the product satisfactory. The table below records each requirement, its objective, how it was tested, and a cross reference to the related non-functional constraints.

Every requirement listed was implemented and verified against its related non-functional constraints before being marked complete.

Title - Completion	Objective Description	Testing	Notes from Team	Related Section(s)
☑ Chat UI	Chatbot is **interfaced seamlessly** into the provided UI which allows users to **interact** and **view conversation** metrics.	Verify that natural language inputs, structured data tables, and dynamic graphical charts render correctly across the UI canvas without formatting degradation.	We are provided the framework for a chat UI by a client who stressed the importance of functionality over aesthetic. Separate session history is **NOT** in MVP.	☑ ~~N-FR:2~~
☑ LLM Orchestration	Chatbot is able to **process** incoming **natural language** text to accurately **detect user intent**, determine **context** boundaries, and dynamically **infer** the necessary **tooling** or **operations** required to fulfill the request.	Verify that natural language inputs are correctly classified by intent, that the agent selects the appropriate tools, generates valid read-only SQL, and maintains conversational context across at least five consecutive follow-up turns without degradation	N/A	☑ ~~N-FR:5~~ ☑ ~~N-FR:6~~ ☑ ~~N-FR:3~~
☑ MCP-style connectors	Chatbot is able to utilize **Model Context Protocol** to **decouple** the core orchestration layer from **varying data sources** to **discover**, **query**, and **interact** with **provisioned resources**	Verify successful handshake and secure connection with all provided test databases and document schemas.	N/A	☑ ~~N-FR:1~~
☑ RAG ingestion pipeline	Chatbot is capable of **parsing, chunking, embedding, and indexing** privatized and internal organizational data into a **vector database maintaining semantic context** and **data enforcement**	Evaluate the pipeline using benchmark evaluation metrics (e.g., Ragas framework for Faithfulness and Answer Relevance) against client-provided test ground-truth datasets	Client has provided sample questions in a specialized ticket. Team members are **NOT** to share any information of the question nor result regardless of its	☑ ~~N-FR:1~~

			'relevance' outside of their respective virtual machine	
☑ Query result	Chatbot **returns** a **highly formatted**, **deterministic**, and **cohesive response** structure. **Unless** explicitly **configured otherwise** by the **user's prompt parameters**	Chat results are returned with: Natural Language Summary, SQL table, SQL graph	N/A	☑ ~~N-FR:8~~
☑ Observability	Chatbot **maintains** a **comprehensive**, **immutable**, and **measurable observability logging pipeline**. This pipeline must **capture end-to-end execution** telemetry	Validate that every user transaction creates a traceable, honest log history capturing exact system states without gaps.	N/A	☑ ~~N-FR:4~~ ☑ ~~N-FR:7~~

Table 2: Functional Requirements Check

Subject to changes or modifications by request or event

III. Non-Functional Requirements

The non-functional requirements are the constraints that shaped how the functional requirements were built. They are not standalone features but quality attributes the system had to satisfy across every component.

Each constraint below was satisfied in the final system, reflected in the Implemented column.

Title - Implemented	Objective Description	Testing	Notes from Team
☑ Multi-Database access	Chatbot is able to access **multiple** databases that are **expected** to have **redundant tables** and **different semantics**	Context from other databases and proper translation of semantics across tables	Client will handle additional database onboarding beyond MySQL, SSMS, and Oracle
☑ Tables and Graph Visualizations	Chatbot is able to **utilize tools** to **represent** the **results** in a digestible **format**	Observe graphs, tables, and other requested formats of answers	N/A
☑ Context Permanence	Chatbot is able to **maintain a persistent memory** of the **current conversation** and **reference / iterate** on its response.	Reference earlier comments or results for verification	N/A
☑ SQL Code Trace	Chatbot is able to **log its SQL code use**, allowing further insight into intention for debugging and reference	Ensure proper SQL code trace output	N/A
☑ Read-Only Access	Chatbot is only able to do **non-destructive actions** to query for results	Ensure minimum level access for chatbot	N/A
☑ Performance Optimization	Chatbot is able to **optimize its queries** to **avoid overburdening or clogging** the systems it will be accessing	Performance metric benchmark	N/A
☑ Security	Chatbot must be able to **recognize** and **handle dangerous,** sensitive, or otherwise **malicious requests** and **redirect** or **cease** operations accordingly.	Edge-case testing regarding malicious tasks must be properly observed to be handled properly	Client assumes responsible users / access only to secure personnel. This item can be looked at during quality building phase

☑ Chat Generation from result	Chatbot must be able to **parse** and **accurately utilize former results** to **iterate** and create more specific answers.	Iterate or further query responses to test extent of precision	N/A

Table 3: Non-Functional Requirements Check

Subject to changes or modifications by request or event

IV. Risks

Deploying a generative AI chatbot within an enterprise data environment introduces a unique set of challenges that extend beyond traditional software development. Among these are the technical hurdles of orchestrating multiple database dialects and the inherent risk of AI hallucinations misrepresenting critical engineering data. The following list outlines the most prominent risks associated with the AskQL project, detailing their impact and the strategies the team used to address them.

1. Inaccurate Text-to-SQL Translation (AI Hallucination)

a. **Likelihood:** Very Likely
b. **Impact:** Major. Providing incorrect analytics data to an engineering team breaks the core objective of the tool.
c. **Mitigation:** A multi-agent validation workflow was implemented using LangGraph. The model executes a dry run against the database schema before returning a result, drawing on the local vector DB to retrieve exact schema mappings and verified query examples.

2. Multi-Database Conflicts

a. **Likelihood:** Likely
b. **Impact:** Moderate. Query failures or syntax errors will occur if the chatbot applies the wrong syntax to a specific database.
c. **Mitigation:** Specialized prompt templates and individual LangChain tools were built for each database type. The first step in the LangGraph pipeline is a Router node that identifies the target database and enforces its specific SQL syntax.

3. Dynamic Visualization Generation Failures

a. **Likelihood:** Likely
b. **Impact:** Moderate. It will degrade the user experience but the raw data (table format) can still be provided as a fallback.
c. **Mitigation:** Visualization options were constrained to tables, bar charts, line charts, and a few other types. Strict data-type checking was added in the FastAPI backend before passing the payload to the React frontend.

4. Frontend Development Capability Gap

a. **Likelihood:** Very Likely
b. **Impact:** Major. If the student team lacks frontend experience, the UI prototype will bottleneck the entire project, even if the backend AI works perfectly.
c. **Mitigation:** The team used the Qualcomm Code Assistant to generate ReactJS boilerplate. The client provided the initial chat UI framework, allowing the team to focus on data rather than building UI architecture from scratch.

5. Steep Learning Curve for Agentic Frameworks

a. **Likelihood:** Likely
b. **Impact:** Moderate. Initial development may be slow as we struggle to architect the LangGraph state machine and RAG pipelines correctly.
c. **Mitigation:** The team used the Qualcomm Gen AI SDKs to abstract complexity and held early, frequent architectural reviews with Ed Sunarto to keep the LangGraph pipeline logically sound before writing extensive code.

6. SQL Injection and Security Vulnerabilities

a. **Likelihood:** Unlikely
b. **Impact:** Major. Executing raw SQL strings generated directly from user input opens up massive security risks if a user tries to maliciously delete or alter tables.
c. **Mitigation**: Read-only database connections were enforced for the chatbot service account. FastAPI middleware sanitizes inputs and LLM validation guards intercept commands containing DROP, ALTER, or DELETE.

7. Quantitative Evaluation Metrics for Evaluability

a. **Likelihood**: Likely
b. **Impact:** Major. The project brief highlights a strong emphasis on evaluability, meaning the project fails if the team cannot mathematically prove the accuracy of the chatbot.
c. **Mitigation:** A synthetic evaluation dataset of natural language questions mapped to verified ground-truth SQL queries was built, and an evaluation pipeline calculates precision, recall, and exact-match scores automatically.

ID	Risk	Category	Likelihood	Impact	Risk Level	Treatment	Owner	Status
1	Inaccurate Text-to-SQL Translation (AI Hallucination)	Technical	Very Likely (5)	Major (4)	High (20)	Implement multi-agent validation	Kevin	Done

2	Multi-Database Conflicts	Technical	Likely (4)	Moderate (3)	Moderate (12)	Develop specialized prompt templates	Thai	Done
3	Dynamic Visualization Generation Failures	Technical	Likely (4)	Moderate (3)	Moderate (12)	Constrain visualizatio n options	Team	Done
4	Frontend Development Capability Gap	Technical	Very Likely (5)	Major (4)	High (20)	Leverage Claude Code	Sang	Done
5	Steep Learning Curve for Agentic Frameworks	Operational	Likely (4)	Moderate (3)	Moderate (12)	Utilize Claude Code	Ryker	Done
6	SQL Injection and Security Vulnerabilities	Security	Unlikely (2)	Major (4)	Low (8)	Enforce read only database connections	Ryker	Done
7	Quantitative Evaluation Metrics for Evaluability	Technical	Likely (4)	Major (4)	Moderate (16)	Build synthetic evaluation dataset	Team	Done

Table 4: Risk Registry

Risk Level Matrix

Likelihood	Low Impact	Medium Impact	High Impact
High Likelihood	Low	High	Critical
Medium Likelihood	Low	Medium	High
Low Likelihood	Low	Low	Medium

Table 5: Risk Level Matrix

V. Definition of Done

Criterion	Definition
☑ ~~Unit and integration tests passing~~	All tests created throughout the project passed. Edge cases were handled and bugs found during testing were documented and addressed in subsequent iterations.

☑ ~~Security review completed~~	Security assessments were completed. Privatized information and data protection procedures were followed throughout development.
☑ ~~High Accuracy Chatbot~~	The chatbot consistently produces correct reproducible results. Users are able to clarify queries to receive more precise answers.
☑ ~~Monitoring Dashboards~~	Auditable and observable logs are provided and properly maintained for future reference and debugging. Basic logged elements include SQL code, access, and thought trace.
☑ ~~Deployment Pipeline~~	Sequences of deployment were explicitly outlined for usage including, though not limited to, procedures to implement additional sources of information retrieval such as databases, documents, etc.
☑ ~~Documentation~~	Team members logged all relevant knowledge and outcomes incurred throughout development into their respective fields. Information regarding usage, caution, and supplementary context appears in relevant sections.
☑ ~~Sample Database Validated~~	Provided sample inputs passed verification with oversight from respective data specialists while maintaining core functionality. Team members exchanged communications with correlated points of contact to confirm and modify as needed.

Table 6: Definition of Done

VI. System Architecture

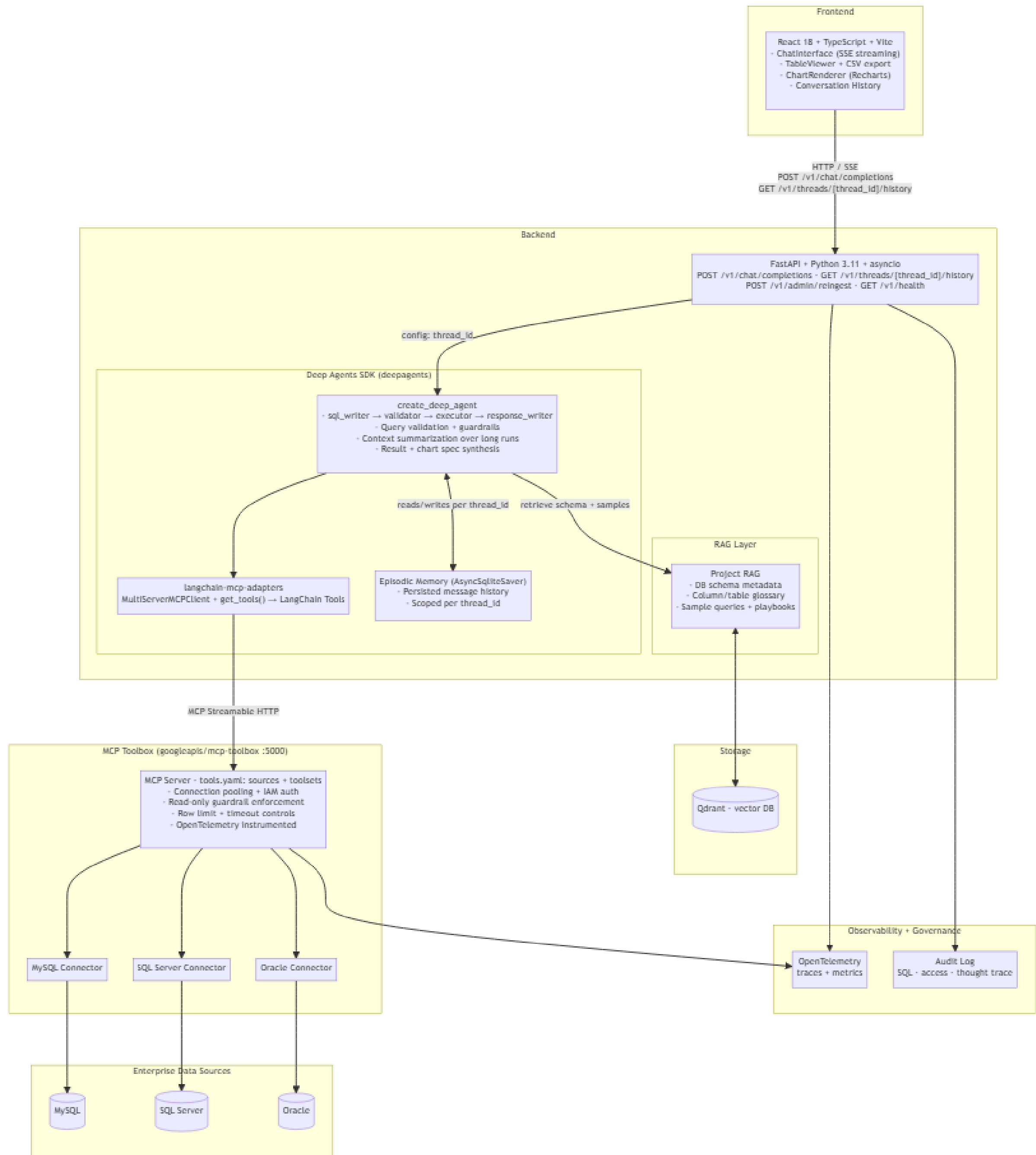

Figure 1: System Architecture

Figure 1 presents the end-to-end system architecture for the Data Analytics AI Chatbot. This is a low-level diagram of the components and tools used in the final system. The info-graphic is designed to unify and clarify team design and client needs by demystifying the ambiguous layer of connections; with edges showing how components connect.

Architecture Layers

1. **Frontend:** The user interface where queries are submitted and results are displayed.
   - Built with React 18, TypeScript, and Vite
   - Renders responses as a natural language summary, data table, and optional chart
   - Receives streamed responses in real time via SSE (Server-Sent Events)

2. **Backend**: The entry point that manages incoming requests and routes them to the agent.
   - Built with FastAPI and Python 3.11
   - Assigns a session-scoped thread_id to maintain conversation context
   - Exposes two endpoints: POST /v1/chat/completions and GET /v1/health

3. **Agent Layer:** The reasoning engine that interprets the user's question and determines how to answer it.
   - Powered by the Deep Agents SDK (create_deep_agent) built on LangChain
   - Handles multi-step tool calling, query validation, and response synthesis
   - Maintains in-session conversation memory via a built-in checkpointer scoped to thread_id

4. **RAG Layer:** Provides the agent with contextual knowledge before generating SQL.
   - Retrieves schema metadata, column definitions, and sample queries from Qdrant
   - Grounds SQL generation in the actual database structure to reduce hallucination

5. **Database Access:** All database interactions are routed through the MCP Toolbox.
   - Google-developed MCP server configured via tools.yaml
   - Enforces read-only access, row limits, and query timeouts at the infrastructure layer
   - Supports MySQL, SQL Server, and Oracle

6. **Observability**: All system activity is monitored and recorded.
   - OpenTelemetry captures traces and metrics across FastAPI and MCP Toolbox
   - Audit Log records SQL executed, tool calls made, and agent thought trace per request

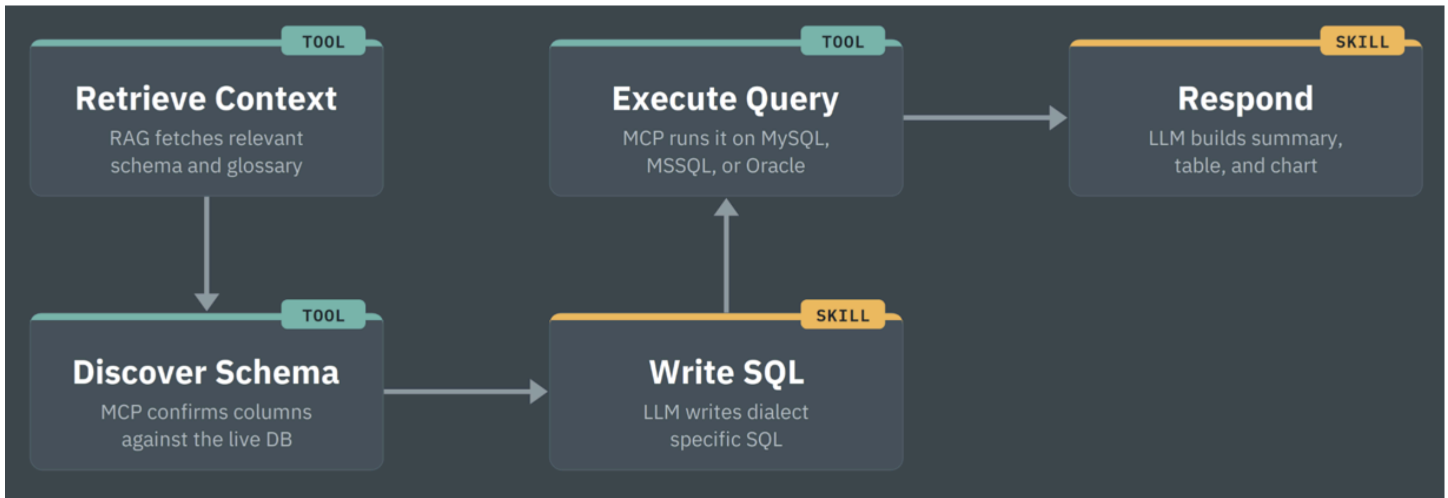

Figure 2: Data Query Flowchart

Figure 2 shows an ideal step by step process the chatbot follows from the moment a user types a question to the moment they receive an answer. This is a high-level overview that encapsulates the key states and respective ownership or tool invoked at the time of processing. This is an ideal path and not a fixed pipeline. The agent can re-decide what to do next at any point, repeating, skipping, or revisiting steps as needed.

Process Flow

7. **User submits a natural language question**
   - Example: "How many deployments happened last week?"
   - Questions are submitted in plain English.
8. **Retrieval of contextual knowledge**
   - The system retrieves:
     - Table names
     - Column definitions
     - Domain specific terminology
   - Information is pulled from a predefined knowledge base.
   - This retrieval process uses RAG (Retrieval Augmented Generation).
9. **SQL query generation**
   - The AI model uses:
     - The user's question
     - Retrieved contextual information
   - The system generates a SQL query targeting the correct tables and columns.
10. **Database execution**
    - Queries are executed through a controlled MCP connection layer.
    - The MCP layer enforces read only access.
    - The system cannot modify or delete database records.
11. **Result validation and reasoning**
    - The AI follows a ReAct (reason + action) loop to come to its final answer.
    - Return rows are structured into a readable format.
    - The AI determines the best visual for corresponding data if applicable.
12. **Response generation**
    - The final response includes:
      - SQL Query
      - Data table(s)
      - Graph(s)/Chart(s) if applicable
    - Results are streamed back to the user in real time.

System Responsibilities

1. **Orange Layer**
   a. Knowledge retrieval and contextual data lookup
2. **Blue Layer**
   a. LLM processing, reasoning, and SQL generation
3. **Purple Layer**
   a. Agent orchestration using RAG, MCP, and LLM integration
4. **Green Layer**
   a. Database query execution and result retrieval

Orchestration

The agent coordinates all workflow stage. Each subsystem operates within its assigned responsibility layer.

VII. Software Test and Quality

To ensure this system meets the client's expectations and rigid operational requirements, the team defined a complete, multi-tiered software quality plan. Every verification mechanism was directly mapped to a functional requirement and its corresponding non-functional requirement, ensuring the development approach met an enterprise-grade definition of done before delivery.

1. Overview of Quality Assurance Methodology

The team used a hybrid quality assurance strategy that balanced speed with comprehensive coverage:

- **Unit Testing:** Conducted continuously by developers to test individual helper functions, database parsers, and API endpoints in isolation.
- **Integration Testing:** Verified that the core LLM Orchestration layer communicates correctly with the MCP Connectors and the RAG Ingestion pipeline.
- **User Interface Testing:** Visual verification of layout stability, structured data tables, and dynamic chart rendering across screen sizes.
- **Automated Evaluation Metrics:** Mathematical benchmarks such as the Ragas framework were used to evaluate LLM outputs deterministically.
- **Code Reviews:** Pull requests were peer reviewed to ensure compliance with architectural patterns and security standards.

1.1 Team Roles & Mapping

To keep testing execution streamlined, the testing efforts were distributed as follows:

- **Front-end / Back-end (API) Developer:** Responsible for the Chat UI framework, API integrations, and layout rendering.
- **Agent Developer:** Responsible for LLM Orchestration, intent routing, context permanence, and query execution structures.

- **RAG Developer:** Responsible for the document ingestion pipeline, semantic chunking, and database indexing evaluation.
- **MCP Developer:** Responsible for Model Context Protocol compliance, database connections, schema handshakes, and access configurations.

2. Requirement-to-Test Mapping Matrix

Test ID	Functional Requirement (FR)	Associated Non-Functional Requirement (N-FR)	Quality Technique	Primary Owner
☑ **Q-FR:01**	Chat UI	N-FR:2 (Tables and Graph Visualizations)	Visual & Layout UI Testing	Front-end / Back-end Developer
☑ **Q-FR:02**	LLM Orchestration	N-FR:3 (Context Permanence), N-FR:5 (Read-Only Access), N-FR:6 (Performance Optimization)	Integration Testing & Program Analysis	Agent Developer
☑ **Q-FR:03**	MCP-Style Connectors	N-FR:1 (Multi-Database Access)	Integration & Connection Handshake Testing	MCP Developer
☑ **Q-FR:04**	RAG Ingestion Pipeline	N-FR:1 (Multi-Database Access)	Automated Evaluation Benchmark (Ragas)	RAG Developer
☑ **Q-FR:05**	Query Result	N-FR:8 (Chat Generation from Result)	Functional Validation & UI Verification	Back-end / Agent Developer
☑ **Q-FR:06**	Observability	N-FR:4 (SQL Code Trace), N-FR:7 (Security)	Dynamic Log Verification & Penetration Testing	Back-end Developer / MCP Developer

3. Detailed Test and Quality Specifications

Test Plan Q-FR:01: Chat UI & Visualization

A. **Associated Requirements:** FR: Chat UI, N-FR:2 (Tables and Graph Visualizations)
B. **Purpose:** To verify the chatbot interfaces seamlessly with the client-provided UI framework, ensuring natural language outputs, structured SQL tables, and dynamic graphical charts render perfectly across all standard device sizes without layout degradation or overflow.
C. **Description:** Automated browser-driven tests rendered mock responses containing diverse Markdown structures, complex tabular data, and high-density charts. Manual exploratory tests verified structural integrity on multiple viewport breakpoints. Note that per MVP specifications, separate session history features are omitted.
D. **Tools Used:** React 18 + TypeScript + Vite, Recharts, ReactMarkdown, FastAPI backend, manual browser testing via Chrome.
E. **Threshold for Acceptability:** Chat UI renders responses end-to-end; all four chart types display without error; tables support sort, filter, and CSV export; token stream animates in real time.
F. **Edge Cases:**
    1. Chart rendered with missing required fields (e.g. bar chart without xKey/yKeys)
    2. Table response with 0 rows returned
    3. Response containing multiple charts in a single answer
    4. MCP row limit hit, triggering truncated result warning
G. **Expected Result:** Agent responses stream token-by-token into the chat UI. SQL results render as interactive tables with sort, filter, pagination, and CSV export. Charts render correctly for bar, line, pie, and scatter types based on the agent's output. Clarifying questions display the "NEEDS CLARIFICATION" badge. Errors surface a retry button.
H. **Notes From Team:** PNG chart export uses SVG-to-canvas conversion and requires the chart to be fully rendered before download. Session history is intentionally not persisted across page refreshes per MVP specification.
I. **Result / Status:** Passed The chat UI correctly handles all SSE event types in live testing against the running backend. Bar charts render cleanly for real TBS Jira and deployment queries. Tables display with sort, filter, pagination , and CSV export all functioning. The NEEDS CLARIFICATION badge appears correctly on ambiguous responses. Token streaming animates in real time. Error states surface the retry button as expected. PNG download produces a valid image file.

Test Plan Q-FR:02: LLM Orchestration

A. **Associated Requirements:** FR: LLM Orchestration, N-FR:3 (Context Permanence), N-FR:5 (Read-Only Access), N-FR:6 (Performance Optimization)
B. **Purpose:** To verify the core Agent is able to process natural language inputs, accurately detect user intent, stay within contextual boundaries, infer tools dynamically, maintain stateful conversational memory, and guarantee that all inferred SQL queries are purely read-only and optimized.
C. **Description:**A suite of semantic regression prompts evaluated intent classification accuracy. The orchestration layer was run against long-running mock dialogs to test memory and performance under load.
D. **Tools Used:** LangChain, Qdrant, FastAPI, MCP tools
E. **Threshold for Acceptability:**
    a. Intent classification accuracy of 90% or higher across test prompts
    b. No destructive SQL generated across any test run
    c. Context retained correctly across at least 5 consecutive follow up turns
F. **Edge Cases:**
    i. Ambiguous query that maps to multiple databases
    ii. Follow up question referencing a result from 3+ turns ago
    iii. Query that looks like a SELECT but contains a subquery attempting a write
    iv. Empty result set returned from a valid query
G. **Expected Result:** Agent correctly classifies intent, selects the right tool, generates read only SQL, and

maintains conversational context across multi turn dialogs without degradation.

H. **Notes from Team:** N/A

I. **Result / Status:** Passed - The agent correctly classifies intent, enforces read only guardrails, generates valid SQL, and maintains context across 5+ follow up turns with no degradation. No destructive SQL was produced across any test run. Query accuracy for business terminology is currently limited by the RAG knowledge base state. Proprietary Qualcomm docs have not been ingested due to data governance constraints, so the team authored custom schema and glossary documentation in their place. The agent behaves correctly given what it has access to, and accuracy will increase proportionally once internal documents are onboarded.

Test Plan Q-FR:03: MCP-Style Connectors & Multi-DB Handshake

A. **Associated Requirements:** FR: MCP-style connectors, N-FR:1 (Multi-Database access)

B. **Purpose:** To verify the Model Context Protocol (MCP) engine successfully decouples the core orchestration layer from diverse, redundant target databases and document schemas, ensuring secure connections, correct translation of table semantics, and stable handshakes.

C. **Description:** Live handshakes were established across sandboxed target databases. The metadata discovery process was verified by having the agent query connection boundaries to discover schemas.

D. **Tools Used:** internal Qualcomm MCP db tools, genai MCP toolbox, langchain toolbox adapter, googleapis/mcp-toolbox

E. **Threshold for Acceptability:** Health and smoke tests pass; live database specific items are query-able such as open tickets.

F. Edge Cases:
   1. Additional Database Integration
   2. Non-standardized Database format Integration
   3. Agent confusion resulting from non-standardized database

G. **Expected Result:** The agent queries across multiple databases and infers the correct usages from schema. The retrieval from these separate databases should also be appropriate; i.e., querying for stations for a specific county should not retrieve results totaling from other sectors.

H. **Notes from Team:** Cross table (same database) joins are a main goal easily accomplished through proper usage of foreign keys. These joins should be intuitive if documented properly in database comments. Cross database joins are NOT considered, this operation is considered extraneous and not something the client is willing to divert resources for.

I. **Result / Status:** Passed - Agent is able to not only access all included databases (MySQL, SSMS, Oracle), but has the intuitive ability to jump across these databases to try and uncover the correct answer. Though its starting position seems obscure, the ability to recognize that certain portions of the database may not contain the correct or reflective information and then correspondingly changing its scope is satisfactory and impressive.

Test Plan Q-FR:04: RAG Ingestion Pipeline

A. **Associated Requirements:** FR: RAG ingestion pipeline, N-FR:1 (Multi-Database access)

B. **Purpose:** To evaluate the parsing, chunking, embedding, indexing, and subsequent retrieval stability of organization-sensitive data inside vector stores.

C. **Description:** A secure evaluation script parsed client-provided data, chunked the documents, embedded them, and ran standard retrieval queries. Automated checks applied the Ragas framework metrics using designated benchmark equations.

D. **Tools Used:** Pytest, Pytest-asyncio, Qdrant , BAAI/bge-small-en-v1.5 embeddings

E. **Threshold for Acceptability:** Similarity of returned documents to use query is at least 0.7, K limit respected, data from all databases must be indexed, results returned in decreasing score order (highest first)

F. **Edge Cases:**
   1. MCP Toolbox unreachable at ingest time or empty “rag_data/” directory - pipeline must complete gracefully using only existing data
   2. Reingest cycle - collection must be deleted and then rebuilt and then the retriever ref must

be queryable immediately after reingest cycle

3. Broken retriever at query time - Retrieve context must return an empty string if it is broken

G. **Expected Result:** All RAG tests must pass in an isolated temp store with a minimum score of 70% with a set of required keywords that must be present.

H. **Notes from Team:** Schema ingestion depends on MCP Toolbox being reachable, cold build flag must distinguish when data reingestion has been run.

I. **Result / Status:** Passed. All five retrieval checks completed successfully: persistent collection existence and non-emptiness after ingestion, cross-database coverage confirming both MySQL and MSSQL content was indexed, chunk content validity with no empty documents stored, retriever functionality confirming results are returned within the k limit, and keyword presence in top-k results for representative domain queries. Detailed scores are withheld per client data governance requirements.

Test Plan Q-FR:05: Query Result Formatting

A. **Associated Requirements:** FR: Query result, N-FR:8 (Chat Generation from result)

B. **Purpose:** To guarantee that the chatbot returns a highly structured, readable, and deterministic response layout (Natural Language Summary, SQL Table, and SQL Graph visualization) and is capable of iterating on those results.

C. **Description:** Functional prompts were run against the query engine. The output text was structurally parsed to ensure all three mandatory elements (Summary, Table, and Graph) are present in the response body. Follow-up queries were made to test iterative query generation.

D. **Tools Used:** FastAPI backend, deepagents/LangGraph agent, SSE stream inspection, audit.log for tool call sequence verification.

E. **Threshold for Acceptability:** Every data-returning response includes a natural language summary, a populated result table, and a chart where visualization is appropriate. Follow-up queries referencing prior results produce contextually correct output without requiring the user to re-explain the original question.

F. **Edge Cases:**

1. Query that returns no rows, table event omitted, summary explains empty result
2. Ambiguous query requiring clarification, only token events fired, no table or chart
3. Multi-step query requiring multiple SQL executions before a result is returned
4. Follow-up query referencing a prior result ("show me the top 5 of those")
5. Query where visualization is not appropriate, chart event correctly omitted

G. **Expected Result:** Agent returns all three output components for data-returning queries: a natural language summary via token stream, a structured table via the table event, and a chart spec via the chart event. The Pipeline view correctly shows each SQL query executed in sequence. Follow-up queries within the same session correctly reference prior context without requiring the user to re-state the original question.

H. **Notes From Team**: Chart generation is driven by the agent inferring visualization type from the result shape. the user does not specify chart type. If the agent determines visualization is not appropriate, the chart event is omitted entirely and this is expected behavior, not a failure. The thread_id scopes all conversation memory, so follow-up query correctness depends on the session remaining open.

I. **Result / Status:** Passed. Data returning queries consistently produce all three output components: a natural language summary via token stream, a structured table via the table event, and a chart spec via the chart event. The Pipeline view correctly surfaces each SQL execution in sequence. Follow up queries referencing prior results resolve correctly within the same session without requiring the user to restate the original question, scoped by thread ID. Chart omission behaves as expected. When the agent determines visualization is not appropriate for the result shape, the chart event is correctly skipped and no error is raised.

Test Plan Q-FR:06: Observability, SQL Tracing, and Security

A. **Associated Requirements:** FR: Observability, N-FR:4 (SQL Code Trace), N-FR:7 (Security)

B. **Purpose:** To verify the chatbot logs all end-to-end execution telemetry (including exact SQL codes executed) into an immutable, searchable database while handling malicious queries safely.
C. **Description:** A variety of benign queries were executed and the generated telemetry logs were inspected to ensure they capture exact timestamped parameters, transaction states, and SQL traces without logs being exposed to data leakage risks.
D. **Tools Used:** OTel, genai MCP toolbox
E. **Threshold for Acceptability:** Honest and observable logs can be retrieved and parsed. Every agent action leaves a traceable mark through thought or code. Timestamped session data including time, session ID, and sources queried is recorded.
F. **Edge Cases:**
    1. Error or failed requests
    2. User stopped request
    3. Timeout
G. **Expected Result:** Timestamped session logs, sources queried, and SQL traces are recorded in the audit log for every transaction.
H. **Notes From Team**: Each section of the system has some form of auditability. A tool kit created from OTel API, codename "Jäger ", encompasses the general auditing kit that the team works off of.
I. **Result / Status:** Passed - Otel implemented and both logs and audit trail are provided for further development and testing procedures. The internal telemetry toolkit "Jäger" captures complete audit trails across test runs and acts as the MCP logging wrapper.

VIII. Project Ethical Considerations

This section examines the ethical dimensions of building an AI system that queries sensitive enterprise data. It identifies the principles most relevant to the project, the ones most at risk, the tests applied to reason about them, and the safeguards built into the system.

Pertinent Principles:

- **Contribute to Society and Human Well-being**: Qualcomm will be able to make better data driven decisions.
- **Know and Respect Existing Laws:** The chatbot must comply with data governance policies, access control requirements, and audit trail regulations established by Qualcomm.
- **Ensure Transparency**: Users should understand how the chatbot answers questions, the data sources being queried, and the confidence level of the results.

Most Vulnerable Principles:

- **Avoid Harm:** A security breach exposing sensitive information or undetected hallucinations could lead to incorrect strategic decisions which would cause harm to Qualcomm.
- **Provide Service of High Quality:** This is vulnerable if accuracy safeguards are insufficient. Stale data could negatively impact decision-making.

Michael Davis Tests Applied:

- **Public Disclosure Test**: Comprehensive audit trails of queries, LLM responses, and SQL queries were maintained so that decisions made by the system can be justified through the audit trail if publicly disclosed.
- **Reversibility Tes**t: The chatbot is read-only by design so it can only query databases. This ensures that incorrect outputs cannot cause data corruption. This was mitigated by implementing confidence scoring to flag uncertain results and adding verification steps that prompt the LLM to validate its reasoning.

Software Quality Plan Impact:

- **Insufficient Testing:** Without rigorous testing of confidence scoring and hallucination, we could deploy a system that appears reliable but actually produces inaccurate results.
- **Inadequate Access Control Testing:** Without rigorous testing of the access control system to make sure that only certain internal Qualcomm employees have access to this chatbot, unwanted users could expose sensitive data.
- **Incomplete Audit Trail Implementation:** Without comprehensive logging of all SQL queries and statements and responses, there may not be enough information to investigate security incidents.
- **Inadequate SQL Injection Prevention Testing:** Without thorough testing that the LLM cannot bypass safeguards, we risk compromising sensitive data.

Security considerations:

- <u>Data & Privacy</u>
    - **Source transparency:** Users should be able to know where the information is coming from.
    - **Data sensitivity:** We must ensure proper access controls so that only internal authorized Qualcomm employees can access the chatbot since sensitive data will be shown to the user.
- <u>Accuracy & Hallucination</u>
    - **Confidence scores:** The LLM generates a confidence score for each user query that is not shown to the user but is used to identify potential accuracy problems.

- **Stale data:** In order to ensure up to date and accurate data, the chatbot retrieves new data everyday.
- **LLM Hallucination:** LLMs can still make up things even when they have good data. This is mitigated by a verification step that encourages the LLM to ask the user for clarification when uncertain.

- Misuse Prevention
    - **SQL Injection:** The user will not be able to run SQL queries but the LLM will be able to run SQL queries in order to get the context that it needs in order to answer the question. The tool that runs the SQL queries only allows SELECT statements and always uses LIMIT. The databases are also read only in order to prevent modification of the databases.
    - Audit trail: Every user query also logs everything that happened including all SQL queries that were run as well as the user prompts and final LLM response in order to identify potential threats.

IX. Project Completion Status

This section summarizes what was completed, what was left incomplete, and how each area was tested, organized by the four development tracks: backend and frontend, agent, RAG, and MCP.

Backend / Front End Development

Completed Items

- Architecture Design (**MVP**)
  - Defined the full system architecture and produced the architecture diagram and component breakdown that served as the shared reference for all team members throughout development.
- API Contract (**MVP**)
  - Specified all endpoints, request and response shapes, SSE event types, and error behavior before implementation began. Updated incrementally as scope evolved.
- FastAPI Implementation with SSE Streaming (**MVP**)
  - Implemented all four endpoints. The chat endpoint streams token, sql, table, chart, done, and error events progressively via SSE as they are produced by the agent.
- Frontend - SSE Client, ChatInterface, TableViewer, ChartRenderer (**MVP**)
  - Built the full React 18 + TypeScript frontend including real-time token rendering, paginated and sortable table view with truncation warnings, and chart rendering for bar, line, pie, and scatter types from agent-generated specifications.
- Excel Export for TableViewer (**In-Scope**)
  - Users can export any query result to an .xlsx file directly from the table view.
- Codebase Refactor and Project Organization (**In-Scope**)
  - Restructured the repository into clearly separated layers and standardized naming conventions and module boundaries across the backend.
- Dockerization (**Post-MVP**)
  - Containerized the FastAPI application and frontend alongside the existing MCP Toolbox Docker setup, allowing the full stack to be stood up with a single compose command.

Incomplete Items

- Model Selection (**Out-of-Scope**)
  - Evaluating available QGenie model configurations to select an optimal production model based on SQL dialect adherence, tool sequence correctness, and response latency was not completed within the project timeline.
- Dockerization (**Post-MVP**)

- The MCP Toolbox runs in Docker but the FastAPI application and frontend have not been containerized. Deprioritized in favor of completing functional scope within the project timeline. Currently being implemented.

- Embeddable Widget and External API Access (**Future Work**)

  - The client has expressed interest in embedding the chatbot inside existing internal portals and exposing the agent programmatically. The current SSE contract supports both, but the integration approach has not been defined or implemented.

Testing

- Frontend

  - SSE client event parsing for all event types, database utility functions, and TableViewer pagination, sorting, filtering, and truncation behavior. Validated in Chrome and Edge with no compatibility issues identified.

- Backend Unit Tests

  - Endpoint request validation, chart repair logic, and telemetry utilities. All unit tests run offline via a conftest.py fixture that mocks load_agent_deps without requiring Docker or live database connections.

Agent Development

Completed Items

- Intent Classification & Skill Routing (**MVP**)
  - Correctly classifies user intent and routes to the appropriate SQL skill and tools across all supported dialects
- Multi-Dialect SQL Generation (**MVP**)
  - MySQL, SQL Server, Oracle Database
- Multi-Database Session Support (**MVP**)
  - Agent can query multiple databases within the same session
- Summarization Middleware & Context Management (**In-Scope**)
  - Offloads conversation history by summarizing older turns rather than accumulating raw tokens, keeping the context window compact across extended multi-turn sessions
- Guardrails & Safety (**In-Scope**)
  - Agent refuses harmful prompts and notifies the user of out of scope prompts not relating to the agent's function, enforces read only access on every query, and prevents destructive SQL generation
- Structured Response Formatting (**In-Scope**)
  - Every data returning response includes a natural language summary, SQL table, and chart where visualization is appropriate

Incomplete Items

- Sub-Agent Orchestration (**Post-MVP**)

    - Partially implemented with a one line code switch to enable. Specialized workers for schema retrieval and query generation are designed but not production ready. This is the intended long term fix for tool over-invocation and query latency on complex inputs

- Agentic Workflows (**Post-MVP**)

    - External task automation such as ticket creation was evaluated and intentionally excluded. The agent is scoped to data retrieval and analytics only

Testing

- a. Integration Tests
    - MCP connectivity and tool invocation across all three database connectors
    - RAG retrieval and context injection into agent prompt
    - Quantitative performance improvements were observed across development iterations. Schema lookup context was reduced from approximately 20,000 tokens to 500 tokens per query following the discover_schema fix, which eliminated the grep cascade caused by the default schema dump returning the full database rather than the requested table. This directly reduced tool call count and end to end response time. SQL accuracy against business terminology remains limited by the RAG knowledge base state, as proprietary Qualcomm documentation has not been ingested due to data governance constraints. The agent performs correctly given available context and accuracy will increase proportionally as internal documents are onboarded.
- b. Guardrail Tests
    - Out of scope prompt rejection (e.g. "what is the weather in San Diego")
    - Destructive query prevention (e.g. DROP, DELETE, UPDATE attempts)
- c. Functional Tests
    - Schema awareness (e.g. "how many tables are in X database")
    - Multi turn follow up query correctness
    - Empty result set handling
    - Ambiguous query clarification flow

RAG Development

Completed Items

- Data Ingestion (**MVP**)

    - MySQL database schema and glossary

    - SSMS (SQL Server) database schema and glossary

    - Input documents were iteratively refined into structured, readable text files containing table-level descriptions and column-level documentation. Each iteration was tested to confirm measurable improvements in retrieval relevance before proceeding.

- HyDE Retrieval (**MVP**)

    - Implemented Hypothetical Document Embedding (HyDE) to improve retrieval accuracy. Rather than embedding the user's raw question directly, the system first generates a synthetic schema passage describing what an answer would look like, then uses that passage as the search query. This better

matches the semantic shape of the indexed documents. If generation fails, the system falls back to embedding the raw query automatically.

- RAG testing (**MVP**)
    - The retrieval pipeline was tested after each iteration to verify that accuracy was improving. Testing covered:
        - Persistent collection existence and non-emptiness after ingestion
        - Cross-database coverage confirming both MySQL and MSSQL content was indexed.
        - Chunk content validity ensuring no empty documents were stored
        - Retriever functionality confirming the retriever returns results and respects the k limit
        - Keyword presence in top-k results for a representative set of domain-specific queries
        - Similarity score thresholds requiring a minimum cosine similarity of 0.7 for retrieval to be considered confident.

Incomplete Items

- Unstructured Data Sources (**Out-of-scope**)
    - The initial scope was deliberately limited to database schema and local glossary files to establish a working prototype. The client has expressed interest in extending ingestion to additional formats such as Microsoft Excel documents, which would broaden the context available to the agent for business-specific questions.
- MRR Evaluation (**Out-of-scope**)
    - The client suggested exploring Mean Reciprocal Rank as a complementary retrieval evaluation technique. This was not completed within the project timeline. Documentation has been provided so the client can take this over directly.

Quantitative Performance testing

- RAG reingestion
    - Reingestion timing was measured across five runs using a dedicated shell script. Because reingestion runs automatically at midnight every day, the requirement was that it complete in under five minutes. The average measured time was under 3.5 minutes, satisfying this requirement with significant margin.

MCP Development

Completed Items

- Multi-Database Integration (**MVP**)
    - MySQL
    - SSMS (SQL Server)
    - Oracle Database

- Modular Database Integration (**MVP**)
    - Consistent and documented format for adding new databases to the system without modifications or further refactoring of source code. This item is requested by the client so that more resources can be onboarded easily.
- Data Sanitization | Guardrails (**In-Scope**)
    - Data handling both to and from databases to support modular database ingestion. Utilizes built in MCP features for a lightweight and seamless clean working suite.
- Database Toolbox Endpoint (**In-Scope**)
    - Unified all connections to one toolbox endpoint allowing full access to resources through connection. This abstraction is implemented to assist with other development layers' and their need of access to database information without formally connecting to the database itself.
- Database Connectivity (**In-Scope**)
    - TCP connections to databases leverage JSON-RPC 2.0's native large-scale data payload to account for inflated context windows while maintaining a strong and reliable connection. Using both elements to ensure a satisfactory and efficient session with users each and every time.

Incomplete Items

- User-level database integration (**out-of-scope**)
    - Rejected as client requested that users shouldn't be able to add databases themselves, the process should be formally handled through engineers hence the additional complexity.
- Agent auto pickup (**out-of-scope**)
    - Incomplete as the task to reformat the agent and redirect RAG ingestion was deemed far too complicated and dangerous with little payoff. Documents regarding these exclusions are detailed in client files.
- TDV database integration (**formerly in-scope**)
    - Incomplete as clients will take on the integration themselves as they both do not have the time nor resources to onboard the team for TDV database access and want to test the process of integration themselves.

Quantitative Performance testing

- Smoke Testing
    - Health checks across multiple different databases using simple calls on tools. Failure of any is considered a failure on all as loose connectivity to any database may result in inaccurate / misguided answers.
- Integration Testing
    - End-to-end testing confirmed the agent successfully called MCP toolbox items, including Execute, Discover, and Health checks.

X. Future Work

Additional Database Integration

Adding more databases was intentionally out of scope. The main intention of this project is to provide a stable foundation for the client to add their own systems. Scalability is the core goal and the required documentation and procedures were delivered to the client via formalized files.

Sub-Agent Orchestration

Sub-agent orchestration is the primary future work item for the agent. A manager agent delegating to five specialized workers, schema linking, query writing, validation, and execution, is partially implemented and ready to be completed with minimal effort given the existing one line code switch. This architecture will directly address current latency and tool over-invocation issues by distributing work across dedicated workers rather than a single agent handling everything sequentially. The validator layer in particular adds a self correction step before any query reaches the database, improving accuracy and reducing failed execution retries.

RAG Improvements

The most impactful near-term improvement is expanding glossary coverage based on evidence from production. When the agent retries a query or asks for clarification on a question that should be answerable, the audit log captures what context was actually returned. If the retrieved passage is generic schema content rather than the specific table the question requires, that is the gap the glossary should fill. Adding content speculatively risks diluting retrieval quality for concepts that already work well. Additions should be driven by observed failures.

Ingesting unstructured data sources such as Microsoft Excel documents would give the agent access to business context that does not exist in the schema layer, improving accuracy on questions about thresholds, targets, and domain-specific definitions. Implementing MRR evaluation would complement the existing similarity score tests by measuring whether the most relevant document appears near the top of results, not just somewhere in the top-k.

The client should continue refining the glossary to distinguish similar terms, clarify column semantics, and explicitly document differences between databases for the same concept. These targeted, evidence-driven glossary edits have historically had a larger impact on agent accuracy than changes to the retrieval algorithm itself.

Back-End / Front-End

The most directly actionable near-term item is defining the integration approach for the embeddable widget and external API access. The existing SSE contract and token-based authentication are already sufficient to support both use cases without changes to the agent or MCP layers. The embeddable widget would allow the chatbot to be surfaced inside existing internal portals without redirecting users to a standalone application. External API access would allow scripts and internal tools to query the agent programmatically and consume structured results directly. Both paths share the same backend endpoint and differ only in how the client is packaged and distributed.

Model selection was not completed within the project timeline. Evaluating available QGenie model configurations against representative production queries, measuring SQL dialect adherence, tool sequence correctness, chart specification quality, and response latency, would establish a documented basis for the production model choice and provide a repeatable benchmark for evaluating future model upgrades as they become available through the gateway.

XI. Lessons Learned

Backend / Front End Development

Design the API contract before writing any code

Having an agreed upon specification for endpoints, event types, and payload shapes before either side was implemented eliminated an entire class of integration bugs. When scope changed, updating the document first and then updating both sides kept the system coherent. Treating the contract as a living document rather than a one-time deliverable made it a practical tool throughout the project rather than an artifact that drifted out of sync.

Server-Sent Events do not need to be WebSockets

The interaction model is one request followed by one streamed response, which is exactly what SSE is designed for. Reaching for WebSockets would have added bidirectional connection management, more complex CORS configuration, and additional frontend state handling without providing anything the application actually needed. Matching the transport mechanism to the actual communication pattern simplified both the backend and the frontend significantly.

LLM-generated output cannot be passed directly to the frontend

Agent-generated chart specifications frequently referenced column names that differed by case or did not exist in the actual query result. Catching and repairing these mismatches on the server before emitting the chart event eliminated an entire category of frontend rendering failures. Any structured output that originates from an LLM and drives UI behavior needs a validation layer between generation and consumption.

A working test suite before a refactor is not optional

The codebase refactor touched import structure, shared utilities, and module boundaries across both the agent and API layers simultaneously. Having unit tests in place before that work began made it possible to verify that behavior was preserved after each change. Attempting a refactor of that scope without existing coverage would have made regressions invisible until they surfaced in integration testing.

Agent Development

Token Efficiency at Scale

Token usage is important to optimize even when cost is not the primary concern. For a company the size of Qualcomm, API costs are negligible, but inefficient token usage directly impacts latency and user experience. Once a product like this is deployed at scale, prompt size, context window management, and tool call frequency all need to be treated as performance requirements.

Single Agent First

Building a stable single agent before moving to sub agents was the right call. It gave the team a working foundation to test against, debug easily, and understand the system holistically. Jumping straight to multi agent orchestration would have made isolating issues significantly harder and slowed overall progress.

Knowledge Quality Determines RAG Quality

The gap between placeholder documentation and real company docs made it clear that the agent is only as accurate as what gets ingested into the vector store. The architecture is correct but accuracy on business specific questions will not be fully realized until proprietary documentation is onboarded.

Prompt Engineering is Real Engineering

Small changes to the system prompt had outsized effects on agent behavior, tool invocation frequency, and response quality. Prompt changes deserve the same level of iteration, review, and version control as any other part of the codebase.

Latency Compounds in Agentic Systems

Each tool call adds time, and sequential invocations on complex queries stack up fast. Designing with parallelism in mind from the start, rather than retrofitting it later, would have reduced the latency issues the team encountered on multi step queries.

LangGraph Self Correction Proved Its Value Early

Routing failed queries back to the LLM with structured error context instead of surfacing raw database errors to the user was one of the highest value architectural decisions made. It made the agent feel reliable and polished from the user's perspective even when queries failed on the first attempt.

RAG Development

We learned that the index is the bottleneck, not the algorithm. Retrieval accuracy is bounded by what is in the index. The embedding model and HyDE retrieval both performed well, but the agent's ability to answer business-specific questions accurately will not be fully realized until proprietary documentation is onboarded. Architectural correctness does not substitute for content coverage.

We also learned that similarity scores are signals, not verdicts. Mid-range similarity scores do not always indicate a problem. Some concepts are genuinely distributed across many documents and will naturally produce a flat score distribution across the top-k results. The meaningful diagnostic question is whether the agent produced a correct answer, not whether retrieval looked clean. Production retry counts and clarification rates from the audit log are more reliable indicators of real RAG failure than score thresholds alone.

We learned that testing the RAG in isolation pays off. Running the tests independently of the full agent pipeline made it straightforward to determine whether a failure originated in retrieval, SQL generation, or execution. Keeping these layers testable in isolation prevented debugging sessions from becoming guesswork and made each iteration's improvements attributable to a specific change.

MCP Development

We learned the limitations of LLMs regarding connections and utilizing external databases. When we tried using database drivers, we saw how inefficient it was to keep an open socket connection. Working around it, learning about these tools and how to manage stable and secure connections while still holding a large payload, such as an iterated context window, was challenging and a valuable skill to develop.

XII. Acknowledgments

We are incredibly grateful and thankful to the entirety of Qualcomm for the wonderful and insightful opportunities presented to us as a team. We have been eager to work on this project and have dedicated significant time and effort in ensuring that our product to Qualcomm was not only satisfactory, but also representative of our gratitude.

We'd like to formally acknowledge Edward Sunarto, our direct point of contact and main client, for dedicating great effort in assisting us as a team throughout this project. His foresight and immense preparation had been invaluable as the process from onboarding to development was smooth and without issues. We'd also like to highlight his intuition and guidance, key ideals that helped our team gain direction and confidence in our project. Lastly, we'd also like to thank Edward for his commitment to being available with our team; frequent meetings and quick responses made sure that our team never strayed too far from our goals nor developed too long within unclear conditions

Additionally, we'd like to highlight Shankar Krishnamoorthy for his great insight and intuition in our research with artificial intelligence. He provided detailed and impactful resources and tools as the team learned how to develop with large language models. Similarly, we'd like to thank Zhikang Hao for his assistance and efforts with our documentation and data aggregation. His expertise as a data analyst provided great insight for our team whilst developing with real and live data that we were unfamiliar with.

We as a team would also like to thank Donna Bodeau for her mentorship and meetings with the team. Her guidance gave the team much to consider and iterate upon as work progressed across overlapping stages in parallel with CSM course requirements.

And finally, a sincere thank you to Kathleen Kelly and the Colorado School of Mines for organizing this collaborative session. Their efforts and sights to combine student ambition with client needs created a wonderful opportunity for all parties involved and benefited both careers and business development for respective parties involved.

XIII. Team Profile

Ryker Phelps
Computer Science
Hometown: Aurora, CO
Work Experience: Pool Cleaning, Web Development
Hobbies: Rubik's Cubes, Eating, Car Longevity

"I am very excited to work on this project because I love data, I want to learn about using LLMs in applications, and I think this is a product that will help people with their work."

Thai Tran

Computer Science - Computer Engineering

Hometown: Broomfield, CO

Work Experience: FAST Enterprises Software Implementation Consultant, Employment Development Department Software Engineer (EDD, State of California), Chicken Farmer (Colorado)

Hobbies: Arts and Craft, 3D Modeling, Cooking, Hiking, Pickleball, Tennis

"This project seems both incredibly interesting and challenging. I'm hoping that as I near the end of this work, I'll have gained tremendous industry and career skills that can help shape my and hopefully the world's future in a positive light!"

Sang Bui

Computer Science - Data Science

Hometown: Thornton, CO

Work Experience: National Center for Atmospheric Research, Autonomy, Robotics, & Intelligent Algorithms (ARIA) Lab, CSCI 200 teaching assistant.

Hobbies: Concerts, watching sports, traveling

"I am very excited about working on this project, as it explores topics that I currently have no experience in, especially tools to harness LLMs. I look forward to learning as much as I can and contributing to something that can potentially go into production!"

Kevin Bamwisho
Computer Science - Data Science
Hometown: Denver, CO

Work Experience: Undergraduate Researcher in Operations Research, RIOT Sports Intern

Clubs: National Society of Black Engineers (NSBE), Club Basketball

Hobbies: Basketball, Bowling, Snow boarding, Lifting

"I am very excited to work on a project involving artificial intelligence, especially given where the world is today, and to create something meaningful that can positively impact the everyday lives of Qualcomm employees."

References

Googleapis. *MCP Toolbox for Databases*. GitHub, https://github.com/googleapis/mcp-toolbox.

Gutmans, Andi. "Google Cloud Databases Supercharge the AI Developer Experience." *Google Cloud Blog*, 9 Apr. 2025, https://cloud.google.com/blog/products/databases/whats-new-for-google-cloud-databases-at-next25.

Appendix A – Key Terms

Term	**Definition**
TBS	Test Base Station (Qualcomm's internal cellular test infrastructure)
NL	Natural Language
SQL	Structured Query Language (for querying relational databases)
RAG	Retrieval Augmented Generation (gives context to an LLM to improve accuracy)
LLM	Large Language Model (AI model that interprets questions)
MCP	Model Context Protocol (a standard interface for exposing tools and data to an LLM agent
RBAC	Role-Based Access Control (restricts data access based on user role)
AC	Acceptance Criteria (conditions a feature must meet to be considered complete)
UAT	User Acceptance Testing (end user validation before final delivery)
DOD	Definition of Done (the checklist that determines when the project is complete)
SSO	Single Sign On (authentication used in company systems)
Vector DB	Vector Database (stores embeddings used for semantic search in the RAG pipeline

See also Section I for in-context definitions