

ISBN- 979-8-9924095-2-9

Model Context Protocol in the Enterprise

A Hands-On Implementation Guide for Data and Insights



Jothi Periasamy
Author



From Warehouses to Conversations

Shrink Your BI/ETL Footprint with
MCP-Powered Orchestration



Discover MCP for Enterprise Data
Management and Insights

Research & Innovation Team



Jothi Periasamy
Chief Data Scientist
Harvard University



Hrushikesh Anil Attarde
Indian University



Samyukthra Rajkumar S
University Of
California



Lokit Sushruth T
Vellore Institute Of
Technology



Pradyun Sathish
University Of
California



Sameer Navuduri
Amrita Vishwa
Vidhyapeetham



Table Of Contents

EXECUTIVE SUMMARY	1
KEY OBJECTIVE	2
PART 1: CONCEPTS, TERMINOLOGY, HIGH LEVEL UNDERSTANDING & PROCESS FLOW	3
1.1 WHAT IS MCP?	4
1.1.1 MCP Diagram	5
1.2 MCP ARCHITECTURE	6
• 1.2.1 MCP Components	6
• 1.2.2 MCP Layers	7
• 1.2.3 MCP Primitives	8
1.3 MCP VS AGENTIC APPROACH	10
• 1.3.1 Development and Integration	11
• 1.3.2 Maintenance and Scalability	11
• 1.3.3 Development Responsibility Shift	12
• 1.3.4 Potential Security Concerns	12
1.4 DATA MANAGEMENT AND INTEGRATION THROUGH MCP	13
• 1.4.1 How MCP Simplifies Data Management	13
• 1.4.2 How MCP Simplifies Data Integration	13
• 1.4.3 Reducing Dependency on SQL, ETL, Data Warehouses, and Data Lakes	14
1.5 BENEFITS OF MCP FOR DATA MANAGEMENT	16
• 1.5.1 Conversational Access to Structured Data	16
• 1.5.2 Deterministic and Auditable Query Execution	16
• 1.5.3 Modular Tool Integration and Scalability	17
• 1.5.4 MCP For Business	18
1.6 MCP PROCESS FLOW	21
• 1.6.1 High-Level MCP Workflow	21
• 1.6.2 Flowchart of MCP: User Input to Output	24
• 1.6.3 Detailed MCP Workflow	25
1.6.3.1 Steps	25
• 1.6.4 Entity Roles Summary	30
1.7 CUSTOM MCP CLIENT/SERVER ARCHITECTURE	31



EXECUTIVE SUMMARY

As organizations continue to accumulate vast amounts of structured data across customers, products, and transactions, the need for intuitive, efficient, and secure access has never been more pressing. Traditional database management relies on SQL fluency, specialized tools, and engineering support, creating friction for business users who simply want to ask for information or execute routine updates. Our research addresses this gap by pioneering conversational database management powered by the Model Context Protocol (MCP), so interacting with enterprise data becomes as easy as having a conversation.

Simplifying data management using MCP centers on bringing conversational intelligence to the forefront of database operations. Users can express intent in natural language to access, modify, and reason over structured data without writing SQL or navigating complex interfaces. Whether it's checking the status of a credit card application in finance or recording a multi-item sale in retail, the system translates plain-English requests into safe, auditable actions.

At the core is MCP, a structured, deterministic protocol that replaces unpredictable agentic behavior with reliable, tool-driven execution. Instead of improvising which tools to call and how to call them, each request is routed to known, vetted operations with clear guardrails. This creates a transparent path from user intent to database action that is traceable, explainable, and auditable, while preserving enterprise requirements for security and control.

We tackle this research in two parts. In *Part 1: Concepts, Terminology, and Process*, we establish the foundational ideas behind MCP and map the high-level process flow using a real-world business example: *"Check my credit card application status."* We show how a natural-language request is interpreted, matched to approved capabilities, executed against back-end systems, and returned as a clear, business-ready response. In *Part 2: Implementation*, we present a custom business scenario where a user can conversationally modify a database containing **Customer**, **Product**, and **Sales** tables (e.g., "Create customer John Smith, email johnsmith@gmail.com," or "Customer 1 buys 3 of product 1"). We then walk through how to implement an MCP framework to power this behavior, detailing both a **custom MCP client** and a **standard client** (Claude Desktop), with a concise summary of steps followed by a step-by-step implementation guidance for each client.

In effect, our platform reimagines how structured data is managed: it removes the SQL barrier, reduces reliance on technical teams, and delivers a **secure, explainable, and scalable** workflow grounded in MCP. By anchoring database operations in conversational intelligence and deterministic tooling, we make enterprise data management accessible to anyone—without compromising on trust, governance, or performance.

KEY OBJECTIVES

1

Enable Natural Language Database Interaction

Allow users to perform complex CRUD operations across multiple databases (MySQL, PostgreSQL) through simple, conversational prompts—eliminating the need to write or understand SQL.

2

Ensure Predictability Through Tool-Bound Execution

Replace autonomous, improvisational agent behavior with deterministic tool calls that map directly to defined database functions, ensuring clarity, safety, and repeatability.

3

Maintain Transparent Execution Flows

Design the system so that every step of an input-to-output path is traceable, enabling easier debugging, explainability, and compliance with enterprise data governance needs.

4

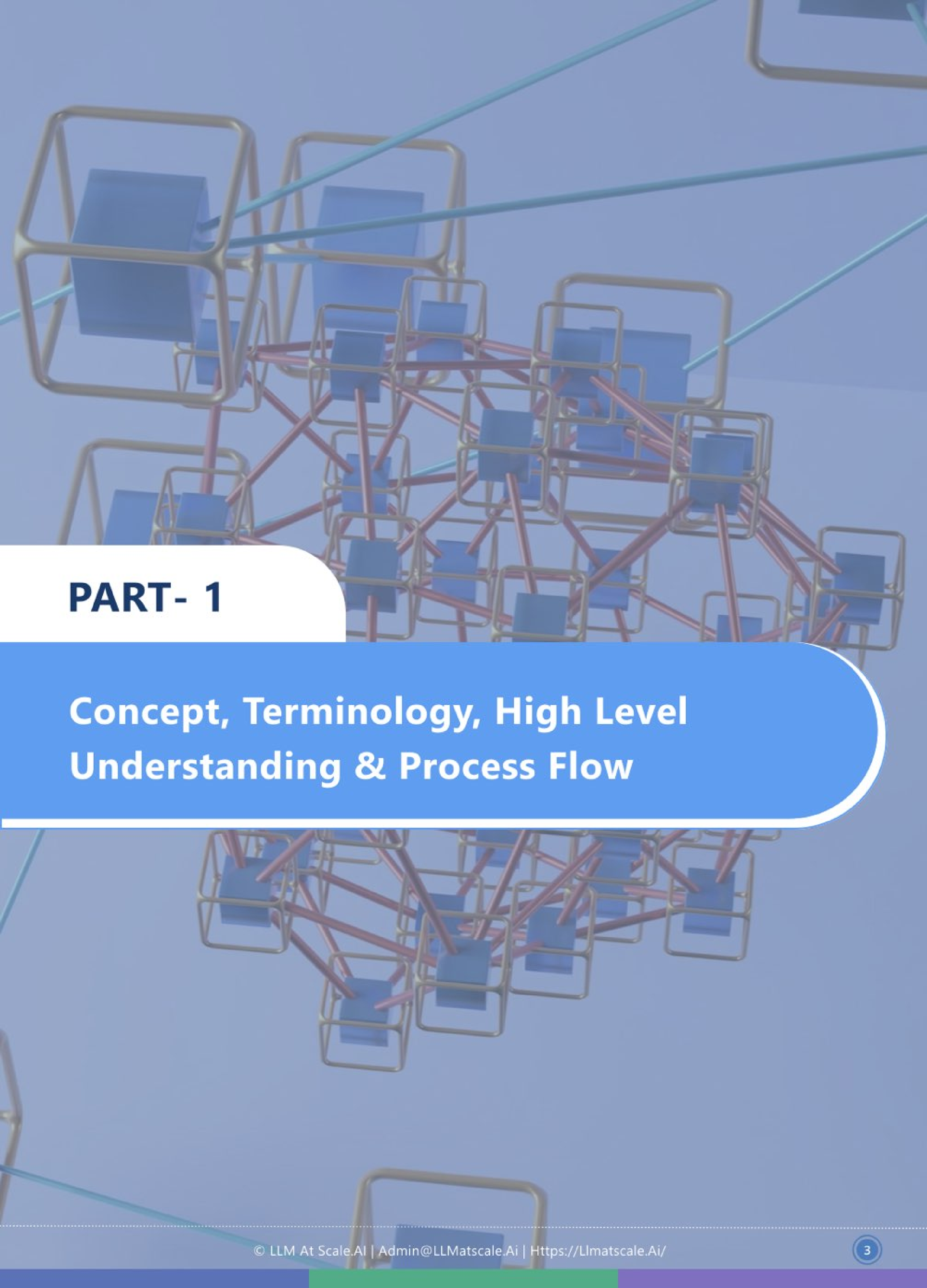
Scale Easily With Modular Tool Design

Build new database functions or add support for other structured systems simply by registering new tools—no need for retraining language models or re-engineering core logic.

5

Empower Non-Technical Teams

Provide a low-friction interface for operations, marketing, and business users to access and update live enterprise data without relying on engineering or BI teams.

The background of the slide is a light blue gradient. It features a complex, abstract structure composed of numerous blue cubes and red lines. The cubes are arranged in a way that suggests a 3D grid or a network, with some cubes appearing to be connected by red lines. The overall effect is a sense of depth and complexity, typical of a modern, tech-oriented presentation.

PART- 1

Concept, Terminology, High Level Understanding & Process Flow

1.1 What Is MCP?

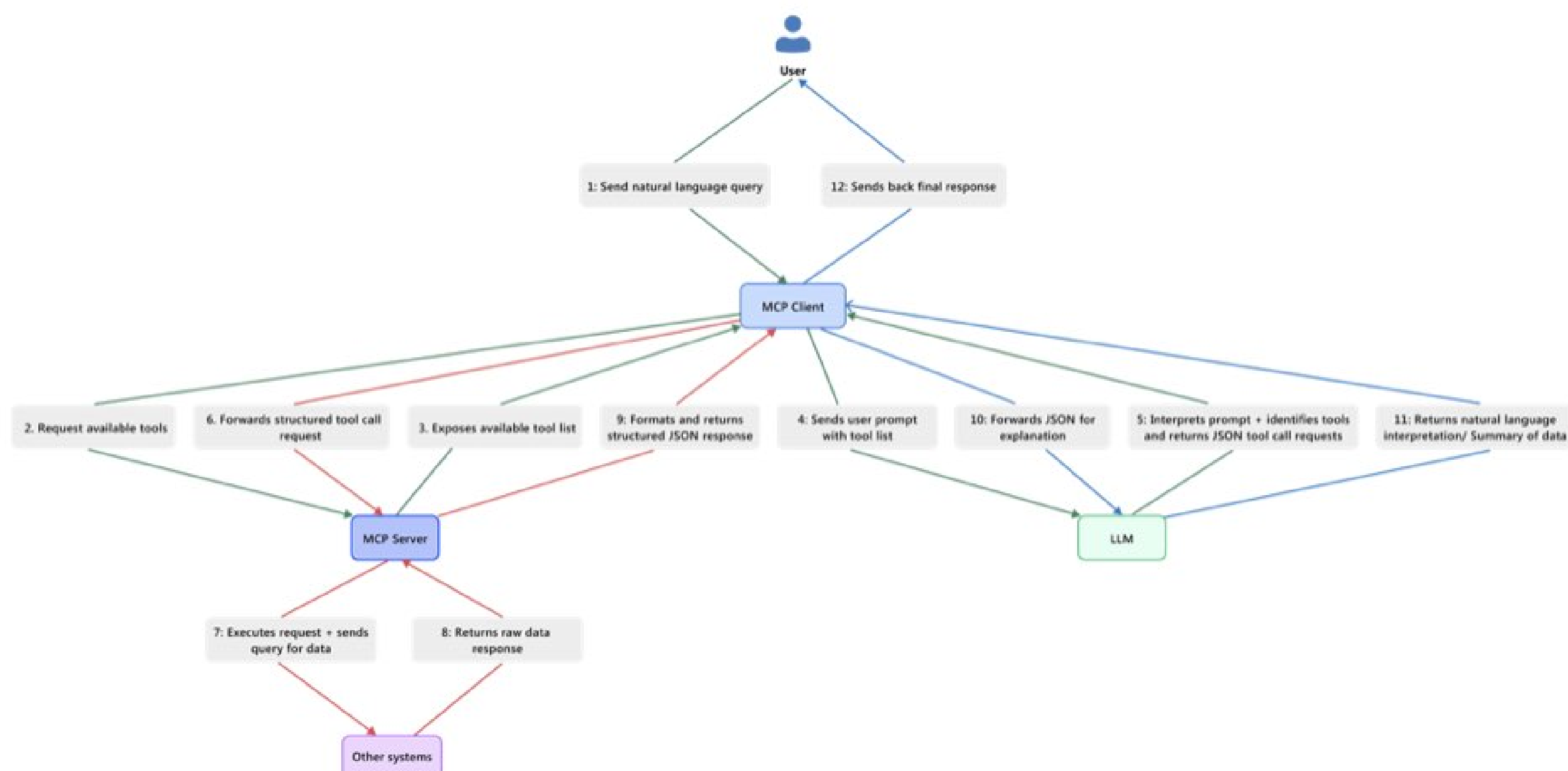
The Model Context Protocol (MCP) is an open-standard, open-source framework designed to connect large language models (LLMs) with external tools, data sources, and environments. First introduced by Anthropic in November 2024 and widely adopted after OpenAI implemented it in March 2025, MCP has quickly gained traction as a potential universal standard for AI system connectivity. By offering a standardized method for communication between AI systems and external services—such as APIs, file systems, and databases—MCP enables LLMs to move beyond traditional retrieval-augmented generation (RAG) or isolated inference. It supports advanced multi-tool agent workflows and complex reasoning tasks by giving LLMs structured access to external functions and contextual resources.

To understand MCP, it's essential to be familiar with a few key terms. An **MCP host** is the interface that a user interacts with. The host, which *houses the MCP client*, is typically an AI application, such as Claude Desktop, or some other interface, such as a Streamlit application. In many scenarios, the client itself can act as the host, allowing users to directly interact with the client. An **MCP client** is the central coordinating component responsible for managing connections and permissions. It controls access between the user and servers, handles user authorization, and manages LLM interactions. The **MCP server** is the tool-bearing component as it exposes specific capabilities, APIs, files, or prompts that the client can use. These servers are often modular, each dedicated to a specific function or domain. **Tools** in this context refer to any actionable capability exposed by a server (e.g., executing a command, retrieving a file, querying a database). **Functions** are typically the specific operations or callable endpoints offered via these tools. Together, these components allow AI agents to operate more like intelligent systems that can plan, coordinate, and act across multiple environments.

Architecturally, MCP is composed of three main components: *the host*, *the client*, and *the server*. It also consists of the LLM, which is either the client itself or a separate component. If the LLM is separate, it is accessed by the client through an API call. The host runs multiple client instances, maintaining authorization rules and coordinating the broader AI-agent workflow. Each client maintains a 1:1 connection with a single server and handles the exchange of messages using the JSON-RPC 2.0 format. For local servers, communication typically happens over standard input/output (stdio). Clients are responsible for capability negotiation and session management, establishing a single

stateful session with their paired server. Servers themselves—whether local or remote—offer isolated, domain-specific capabilities, never seeing the full context of a conversation or having access to other servers. This design ensures privacy, clarity, and composability.

MCP was built on several guiding design principles. First, servers are easy to build: they use simple interfaces and are focused on narrow, clearly defined capabilities, leaving complexity to the host. Second, servers are composable. Because each server performs a focused function, multiple servers can be easily combined to power complex workflows. Third, servers are isolated: they operate independently, receiving only the context necessary for their task, with the host managing all cross-server coordination and maintaining the full conversation history. Finally, MCP is designed for extensibility: new features and capabilities can be added incrementally without breaking existing functionality, allowing clients and servers to evolve independently. This modular, interoperable, and forward-compatible design makes MCP a powerful foundation for building sophisticated AI systems.



1.2

MCP ARCHITECTURE

1.2.1

MCP Components

The Model Context Protocol (MCP) operates on a client-server architecture designed to create a secure, predictable, and scalable workflow for connecting Large Language Models (LLMs) with external tools and data sources. The system consists of several key entities: MCP Host, MCP Client, MCP Server, User, and Other Systems. Each plays a distinct role while working together to ensure seamless orchestration, controlled permissions, and standardized integrations.

The **MCP Host** is the interface the user interacts with. It houses the MCP Client and provides the application environment where requests are initiated and results are displayed. Examples of MCP Hosts include AI applications like Claude Desktop, chat-based interfaces, Streamlit dashboards, or integrated developer tools. In some cases, the MCP Client itself can act as the Host, allowing users to interact with it directly without needing a separate hosting layer, as with the Claude Desktop App.

The **MCP Client** is the central coordinator of the MCP workflow. It manages connections, permissions, and LLM interactions while controlling access between the user and MCP Servers

Key Responsibilities

- Sends the user's query to the LLM.
- Retrieves a list of available tools from connected MCP Servers.
- Validates structured tool requests from the LLM.
- Forwards approved requests to the appropriate MCP Server.
- Receives structured responses and sends them to the LLM for summarization.
- Delivers the final result back to the user.

This separation ensures the LLM never directly interacts with sensitive systems, improving security, auditing, and control.

The **MCP Server** acts as the tool-bearing component in the MCP ecosystem. It exposes APIs, prompts, commands, files, or other actionable capabilities that the client can use. Each MCP Server is modular and typically dedicated to a specific domain (e.g., credit checks, document storage, analytics). When the MCP Client sends a tool execution

request, the MCP Server executes the requested operation, retrieves the necessary data from the appropriate source, formats the result into a clean, structured JSON response, and sends it back to the MCP Client. This structure abstracts away backend complexities like API authentication, versioning, and data formatting, making the system more scalable and secure.

“**Other Systems**” refers to external data sources and services that MCP tools connect to, such as databases, CRMs, APIs, document repositories, or third-party services. These systems remain isolated and are accessed exclusively through the MCP Server, ensuring controlled and auditable data flows.

1.2.2

MCP Layers (Protocol Architecture)

The Model Context Protocol is built on a layered architecture that separates concerns into two distinct levels: the *data layer* and the *transport layer*. Conceptually, the data layer forms the inner core of MCP, while the transport layer surrounds it, providing the channel for secure and reliable communication. This separation allows MCP to maintain consistency in how information is structured and interpreted while remaining flexible in how that information is delivered across different environments.

The ***data layer*** defines the *JSON-RPC 2.0* based protocol that governs communication between MCP clients and servers. It specifies the structure and semantics of the messages exchanged, ensuring a predictable and auditable workflow. Within this layer, ***lifecycle management*** handles connection initialization, capability negotiation, and termination, making sure both client and server are aligned on supported features. The data layer is organized into three groups of functionality: ***server features, client features, and utility features***. *Server features* allow MCP servers to expose core primitives such as tools for executing actions, resources for sharing contextual data, and prompts for reusable interaction templates. *Client features*, in turn, allow servers to request additional capabilities from the client, including sampling from the host LLM, eliciting input directly from the user, and logging messages for monitoring and debugging. Finally, *utility features* provide extended support such as notifications for real-time updates and progress tracking for long-running operations. Together, these groups of features make the data layer the central mechanism through which MCP enables standardized, structured, and controlled interactions.

The ***transport layer*** is responsible for carrying data-layer messages between MCP clients and servers, handling connection establishment, message framing, and secure

communication. It abstracts away the low-level details of transmission while ensuring that the JSON-RPC 2.0 protocol is applied consistently across different communication methods. MCP defines two transport mechanisms: ***stdio transport*** and ***streamable HTTP transport***. *Stdio transport* uses standard input and output streams for direct process-to-process communication on the same machine, offering maximum performance with no network overhead and making it ideal for local deployments. *Streamable HTTP transport*, on the other hand, enables remote communication by using HTTP POST for client-to-server messages and optional Server-Sent Events (SSE) for streaming responses back to the client. This transport supports standard HTTP authentication methods such as bearer tokens, API keys, and custom headers, with OAuth recommended for token management.

By clearly separating the transport layer from the data layer, MCP ensures that the same structured JSON-RPC messages can be exchanged reliably across both local and remote environments without modification.

1.2.3

MCP Primitives

MCP primitives are the fundamental building blocks of the data layer protocol. They define how MCP servers expose functionality and context to clients in a standardized, discoverable, and auditable way. By working through these primitives, clients and servers can interact predictably, enabling language models to access external actions, contextual data, and reusable guidance without relying on ad hoc integrations. Within MCP, the three primary server-side primitives are ***tools***, ***resources***, and ***prompts***.

Tools are executable functions that an AI application can invoke to perform specific actions. Examples include running database queries, executing file operations, or calling APIs. Tools are discoverable using the *tools/list* method, which enumerates the available options, and they can be executed through the *tools/call* method. In some cases, tools may also be inspected using *tools/get* for additional metadata. By structuring tools in this way, MCP allows clients to dynamically discover and call server functionality at run-time while ensuring that each action follows a predictable and controlled process.

Resources represent contextual data that clients can retrieve from servers to inform or enrich an interaction. Unlike tools, resources are not executable but instead provide static or dynamic data such as database schemas, file contents, or API responses. Resources can be enumerated with *resources/list* and retrieved with *resources/get*, enabling clients to access and incorporate external information into workflows. Because resources may also support updates or streaming, they allow language models to remain aware of changing contexts in real time.

Prompts are reusable templates designed to guide and standardize interactions with language models. These can include system-level instructions, few-shot learning examples, or parameterized templates that structure how a model approaches a task. Prompts are discoverable through *prompts/list* and retrievable with *prompts/get*, ensuring they can be reused across different interactions without requiring redefinition. By exposing prompts as a formal primitive, MCP helps ensure that model interactions remain consistent, explainable, and aligned with best practices.

1.3

MCP VS AGENTIC APPROACH

Agentic AI, also known as agentic systems or agent-based models, is an architectural approach where an AI model (typically a Large Language Model (LLM)) is given autonomy to make decisions and perform actions in pursuit of a specific goal. Unlike traditional query-response systems, an agentic setup treats the LLM as a central “agent” capable of reasoning about the user’s intent, planning the necessary steps to accomplish a task, and executing those steps by leveraging external tools. These tools can include APIs, databases, search engines, or private systems, and the agent decides dynamically when and how to use them. This ability to combine reasoning, planning, and tool execution enables agentic systems to operate in dynamic environments and handle more complex workflows than conventional AI systems that simply generate responses. However, giving agents the freedom to act also forces developers to continually build and maintain custom tools for every new API or database, adding significant overhead. Because agents choose which tools to invoke, developers are left building and maintaining a growing set of bespoke connectors and wrappers, each with its own quirks and edge cases. At scale, this approach becomes brittle and costly to govern, thus setting the stage for MCP’s uniform interface.

MCP is much better than an agentic approach because of its standardization. By creating a uniform method of accessing external tools and resources, MCP cuts down the time and effort it takes developers to integrate data access and gives LLMs the ability to interact with that data in a structured and efficient way to produce meaningful results.

Concern	MCP	Agentic Approach
Development and Integration	Tools are defined in standardized schemas and discovered dynamically by the LLM. This reduces custom code and accelerates integration.	Developers must build each tool manually, including handling APIs, authentication, and tool execution logic.
Maintenance and Scalability	MCP servers abstract away backend API changes, reducing maintenance and improving long-term scalability.	Each tool requires manual updates when APIs change, making the system harder to scale and maintain.

Development Responsibility	Tool providers can host and maintain MCP servers, allowing developers to plug in rather than build tools.	Developers must implement, host, and secure every tool, increasing operational overhead.
Security	Standardization and central tool hosting reduce data risk and enable clear auditing of tool behavior.	Tool fragmentation and inconsistent implementations can lead to security gaps and misrouted data.

1.3.1

Development and Integration

With an agentic approach, developers must custom-build each tool the AI agent uses, which involves writing logic to interface with APIs, manage authentication, validate inputs, and handle errors. For example, in the story- upload use case, a developer would need to manually create a function that accesses the Google Docs API, handles OAuth2 authentication, formats the story content, creates the document, and handles the response. This tool must then be integrated into the agent loop, requiring additional logic to parse LLM outputs and trigger tool execution at the right time. In contrast, MCP dramatically simplifies this by offering a standardized way for LLMs to discover and call tools through schemas. The client queries the MCP server for available tools, and the server responds with structured tool definitions. The LLM uses these definitions to decide which tool to invoke and how to format the call. The actual execution is then handled by the MCP server. This streamlined flow reduces custom code and accelerates development, allowing developers to focus more on high-level workflows over integration.

1.3.2

Maintenance and Scalability

Another major difference between MCP and the traditional agentic approach is maintenance. When building tools manually, developers are responsible not only for the initial implementation but also for maintaining compatibility as APIs evolve. If an external service updates its API, such as the Google Docs API, adding new authentication requirements or deprecating endpoints, developers must go back and update their custom tools to reflect those changes. This becomes especially burdensome when multiple tools are in use, each requiring ongoing maintenance to ensure the agentic system remains functional and secure. As the number of integrated tools grows,

maintaining consistency and compatibility across them becomes increasingly difficult and time-consuming. With MCP, the burden of maintenance is significantly reduced because tools are exposed through standardized interfaces managed by the MCP server. The server handles communication with the external API. As long as the MCP tool schema remains consistent, updates to the backend API do not require the developer to touch their LLM orchestration logic. This leads to better long-term scalability and reduces the risk of system failures caused by third-party updates.

1.3.3 Development Responsibility Shift

Another benefit of MCP is that the responsibility for implementing and hosting tools shifts away from the application developer. In an agentic setup, developers must not only write each tool from scratch but also host and secure the infrastructure behind it, such as managing APIs and other operational concerns. With MCP, tool providers can develop and host MCP servers themselves. These servers expose functionality in a standardized way, allowing developers to connect to them without needing to re-implement the logic. For example, instead of building your own Google Docs integration, you could connect to an MCP-compliant server maintained by Google that already supports the functions you need to use. This approach enables a more modular, plug-and-play ecosystem where developers can focus on orchestrating intelligent workflows rather than building and maintaining the underlying tools. It also ensures that tools are implemented securely and maintained by the organizations that know their APIs best.

1.3.4 Potential Security Concerns

Another key difference between MCP and an agentic approach lies in security and the risk of tool fragmentation. In an agentic system, where many developers independently build and host tools that essentially perform similar functions, there's a growing risk of sensitive information being misrouted. Without a unified standard, agents may inadvertently call the wrong version of a tool, or worse, send proprietary or personal data to an untrusted or misconfigured endpoint. This opens the door to serious security concerns, including data leaks of financial records, private health information, and confidential company materials. Since these tools are often written with inconsistent practices and hosted in fragmented environments, validating who built the tool, what it does, and where the data goes becomes difficult. MCP helps mitigate this risk by standardizing tool schemas, enabling tools to be hosted and maintained by trusted providers, and routing all tool usage through a centralized, observable server layer. This ultimately promotes a safer, more reliable environment for LLMs to interact with external systems, especially when dealing with sensitive or regulated data.

1.4

DATA MANAGEMENT AND INTEGRATION THROUGH MCP

The Model Context Protocol (MCP) revolutionizes data management and integration by offering a conversational, standardized, and secure approach. It simplifies complex database interactions and eliminates many traditional pain points associated with data handling.

1.4.1

How MCP Simplifies Data Management

MCP fundamentally simplifies data management by abstracting away the underlying technical complexities and enabling natural language interaction.

- **Conversational Access:** Users can interact with databases using everyday language, similar to having a conversation. This means business users can retrieve or update information without needing to learn SQL queries or navigate intricate database interfaces. For example, instead of writing `SELECT status FROM CreditCardApplications WHERE user_id = 'USR-548923'`, a user can simply ask, "What's the status of my credit card application?".
- **Reduced Dependency on Technical Teams:** With MCP, non-technical teams like operations, marketing, and business users gain direct access to live enterprise data. This significantly reduces their reliance on engineering or Business Intelligence (BI) teams for data retrieval or updates.
- **Predictable and Traceable Operations:** Unlike traditional agentic AI systems that might improvise actions, MCP ensures that every request is securely routed to a predefined and vetted operation. This tool-bound execution makes every database action traceable, explainable, and auditable, enhancing reliability and compliance.
- **Secure and Auditable Workflows:** MCP's standardization and centralized tool hosting minimize data risk and provide clear auditing of tool behavior. This prevents security gaps and misrouted data that can occur with fragmented, inconsistently implemented tools in agentic systems.

1.4.2

How MCP Simplifies Data Integration

MCP streamlines data integration by providing a standardized and modular framework

for connecting disparate systems.

- **Standardized Tool Schemas:** MCP defines tools using standardized schemas, allowing LLMs to dynamically discover and call them. This significantly reduces the custom code traditionally required for integrating various APIs and systems, accelerating development. Developers no longer need to manually build each tool, handling APIs, authentication, and execution logic.
- **Abstraction of Backend Complexity:** MCP servers abstract away changes in backend APIs. This means that if an external service updates its API (e.g., changes authentication requirements or deprecates endpoints), developers do not need to manually update their custom tools or LLM orchestration logic as long as the MCP tool schema remains consistent. This reduces maintenance burden and improves long-term scalability.
- **Shift in Development Responsibility:** The responsibility for implementing and hosting tools can shift from application developers to tool providers who can host and maintain MCP servers. This promotes a “plug-and-play” ecosystem where developers can focus on orchestrating intelligent workflows rather than building and maintaining foundational tools.
- **Modular Design for Scalability:** New database functions or support for other structured systems can be added by simply registering new tools. This modular design eliminates the need for retraining language models or re-engineering core logic, making the system highly scalable and adaptable to evolving data landscapes.

1.4.3

Reducing Dependency on SQL, ETL, Data Warehouses, and Data Lakes

MCP’s approach fundamentally redefines how organizations interact with and manage their data, potentially reducing or eliminating the need for several traditional data paradigms.

- **Elimination of SQL:** The core promise of MCP is to enable users to interact with structured databases using natural language, thereby eliminating the need to write or understand SQL. This democratizes data access, making it available to a much wider audience beyond technical professionals.
- **Reduced Need for ETL (Extract, Transform, Load):** By enabling direct, conversational interaction with live operational data, MCP can reduce the necessity

for extensive ETL processes for certain use cases. Instead of moving data into separate analytical systems, users can query and update data directly at its source through secure, tool-bound operations. This can lead to more real-time insights and actions.

- **Streamlined Data Warehousing:** While data warehouses serve crucial long-term analytical and historical purposes, MCP's ability to query and reason over live, structured data can reduce the need for constantly updating data warehouse extracts for operational reporting or immediate data needs. It provides a dynamic layer of access that complements, rather than fully replaces, data warehousing, by making operational data directly usable for immediate business decisions.
- **Simplified Access to Data Lakes:** Data lakes, often repositories of raw, diverse data, can be notoriously difficult to query without specialized technical skills and complex processing. MCP, through its structured tool registration, can provide a standardized and conversational interface to specific, curated datasets within a data lake, making them more accessible for business users without requiring them to navigate the raw, unstructured environment. This effectively acts as a "conversational front-end" to relevant data within a data lake, without needing to move or transform all of it into a more structured format beforehand.

1.5

BENEFITS OF MCP FOR DATA MANAGEMENT

1.5.1

Conversational Access to Structured Data

MCP allows non-technical users to access complex databases through natural language, removing the need to know SQL or understand table schemas. This opens data access to business teams, marketers, and analysts without requiring backend support. With MCP, users can retrieve information from tables like *Product*, *Customer*, and *Sales* just by asking plain-English questions.

Demonstration:

A sales analyst needs to know which customers bought more than \$500 worth of products last month.

Without MCP

The analyst must file a request with the data team and wait 2–3 days as someone needs to submit a SQL query to the internal database to retrieve the required information.

With MCP

The analyst simply asks: “Which customers spent more than \$500 on products in June?” and MCP is able to route this query using the appropriate pre-built function, allowing anyone to be able to access internal systems and databases.

1.5.2

Deterministic and Auditable Query Execution

MCP uses predefined tools with strict parameter definitions, ensuring that every interaction with the database is predictable and traceable. This deterministic design is essential for auditing and debugging, especially in data-sensitive environments. Unlike dynamic agents that may generate unpredictable SQL, MCP logs the exact tool used, input parameters, and outputs for every request.

Without MCP

Without MCP, they must either write a custom SQL query themselves or submit a request to the engineering team, wait for the report, and risk inconsistent results due to query variations. This process introduces delays, potential errors, and minimal audit tracking.

With MCP

The analyst simply asks: "What's the total revenue from Product ID 103 last quarter?" MCP automatically maps this question to a pre-registered tool, such as `calculate_product_sales(product_id, start_date, end_date)`. The tool executes deterministically, logs the input and output parameters, and returns a clean, traceable response in real time. This consistency improves security, compliance, and transparency while reducing operational overhead.

1.5.3

Modular Tool Integration and Scalability

With MCP, developers can independently define tools to handle operations on specific tables like *Product*, *Customer*, or *Sales*, without altering the underlying LLM. Each tool is registered with a name, parameter schema, and return format, allowing the system to grow modularly. New functionality—such as filtering customers by region or checking product stock—can be added in minutes.

Demonstration:

Your business team wants to start tracking customers by region.

Without MCP

This would involve updating multiple data pipelines and possibly retraining query models.

With MCP

A new tool `get_customers_by_region(region)` is defined and registered. Now, a user can type: "Show me all customers from the Pacific Northwest." and MCP can route the call, query the *Customer* table, and return a filtered list with no retraining and no downtime.

1.5.4

How Simplifying Database Management Using MCP Matters for Businesses

Simplifying database management with MCP translates into significant business value by addressing key operational challenges related to data access, efficiency, and decision-making.

By acting as a universal, standardized adapter for AI agents, MCP provides a structured and efficient framework that drastically reduces the complexity and cost of connecting large language models (LLMs) to enterprise data systems. This foundational strategic benefit enables a cascade of practical advantages for daily business operations. It shifts the burden of managing complex integrations away from developers and toward a modular, scalable, and secure protocol. This allows companies to empower their teams with direct, conversational access to mission-critical data, ultimately accelerating workflows and fostering more informed, real-time decision-making.

Empowers Non-Technical Users: The conversational access to data, made possible by MCP's standardized integration, allows business users (e.g., marketers, sales representatives, analysts) to access and manipulate data using natural language, without needing to learn SQL or other complex query languages. This speeds up workflows and reduces the reliance on a limited number of data analysts or engineers. By enabling conversational database management, MCP empowers teams to interact with the **sales, customer, and product tables** to get the information they need, when they need it, which speeds up workflows and reduces their reliance on a limited number of data analysts or engineers.

Scenario: A sales manager needs to find which customers have spent the most on products.

Without MCP

The manager would have to submit a request to the data team, which then writes and executes a complex SQL query joining the *sales* and *customer* tables, such as `SELECT c.FirstName, c.LastName, SUM(s.total_price) AS total_spent FROM Customers c JOIN Sales s ON c.Id = s.customer_id GROUP BY c.Id ORDER BY total_spent DESC`. This can take hours or even days.

With MCP

The manager simply asks the MCP-powered agent, "Show me the top 5 customers with the highest total spending." The agent uses a pre-defined tool to query the tables and provides a real-time, accurate response. This speeds up decision-making and reduces reliance on technical support.

Eliminating Operational Delays: Instead of waiting for technical teams to run reports or perform data updates, MCP removes the common friction points that slow down business processes. By providing a universal link to data, MCP effectively eliminates the operational delays often caused by technical teams being the sole gateway to information. This allows employees to get the data they need, when they need it, enabling them to make faster, more informed decisions and focus on higher-value tasks. In doing so, MCP empowers a more autonomous, self-sufficient approach to data management across the organization.

Scenario: A business wants to run a promotion and needs to increase the price of all "Tech-Brand" products by \$5.

Without MCP

The marketing team has to identify all products from the "Tech-Brand" brand, manually edit each product's price in a spreadsheet, and send it to the data team for a bulk update. This is a multi-step process prone to errors and delays.

With MCP

A marketing employee can use a simple conversational prompt like, "Increase the price of all products that have 'Tech-Brand' in their name by \$5." The system uses a registered MCP tool to perform the update directly and instantly across the `product` table. This automates a routine, tedious task and frees up both marketing and technical teams.

Enhanced Productivity: Automating routine data tasks through MCP-powered conversational agents reduces manual effort and the potential for human error. This streamlining of operations leads to increased efficiency and cost savings, which is a key outcome of MCP's ability to make scaling integrations cheaper and faster than the traditional, one-off approach.

Scenario: A business wants to launch a new product and needs to create records for it.

Without MCP

A data entry clerk has to manually log into a database interface and fill out multiple fields in the **product** table (e.g., product name, SKU, price, description, etc.). This is a slow and repetitive process.

With MCP

The clerk can simply say, "Create a new product: 'Smart Speaker Pro,' SKU 'SSP-001,' price \$199.99, description 'Voice-activated smart speaker with advanced sound quality.'" The agent uses a tool to correctly parse the request and add the new record to the **product** table, ensuring accuracy and saving valuable time. This streamlining of operations leads to increased efficiency and cost savings.

Future-Proofing: Using the MCP architecture gives the team flexibility in adding new AI/analytics tools that can increase the range of operations. Due to its standardized, simple syntax, the task of adding functionality becomes much faster. This saves a lot of time and debugging efforts for the development team. With the help of MCP, adopting newer technologies becomes easier and centralized, which was a daunting task for developers of the past.

Scenario: A company wants to adopt a graph database to support an AI-driven recommendation engine.

Without MCP

Developers must write new connectors, migrate parts of the data, and rework application logic—an expensive and time-consuming process.

With MCP

The graph database is simply plugged into the MCP layer, allowing the AI engine to query both existing SQL databases and the new graph database through the same conversational interface, saving development time and future-proofing operations.

1.6

MCP PROCESS FLOW

In this section, we walk through the step-by-step workflow of how an MCP-enabled system processes user input, from the moment a question is asked to the moment a response is returned.

To provide both a general and detailed understanding, we begin with a numbered summary of the key steps in the MCP workflow. Following that, we break down each step with deeper technical insight and rationale. To ground the explanation, we use a real-world business scenario:

You recently applied for a Bank of America (BoA) credit card. A month later, you want to know the status of your application—whether it's been approved, denied, or is still in progress. While the information exists in a secure, internal system that isn't directly available to customers, the organization uses an MCP-integrated LLM system that can interface with their private database through tools. This allows you to simply ask, "What's the status of my BoA credit card application?" and receive a response conversationally.

Using this scenario, we illustrate how user intent is translated into a query, how tools are selected, how data is securely accessed, and how the final output is crafted and returned—all powered by the MCP framework.

1.6.1

HIGH-LEVEL MCP WORKFLOW FROM USER TO OUTPUT

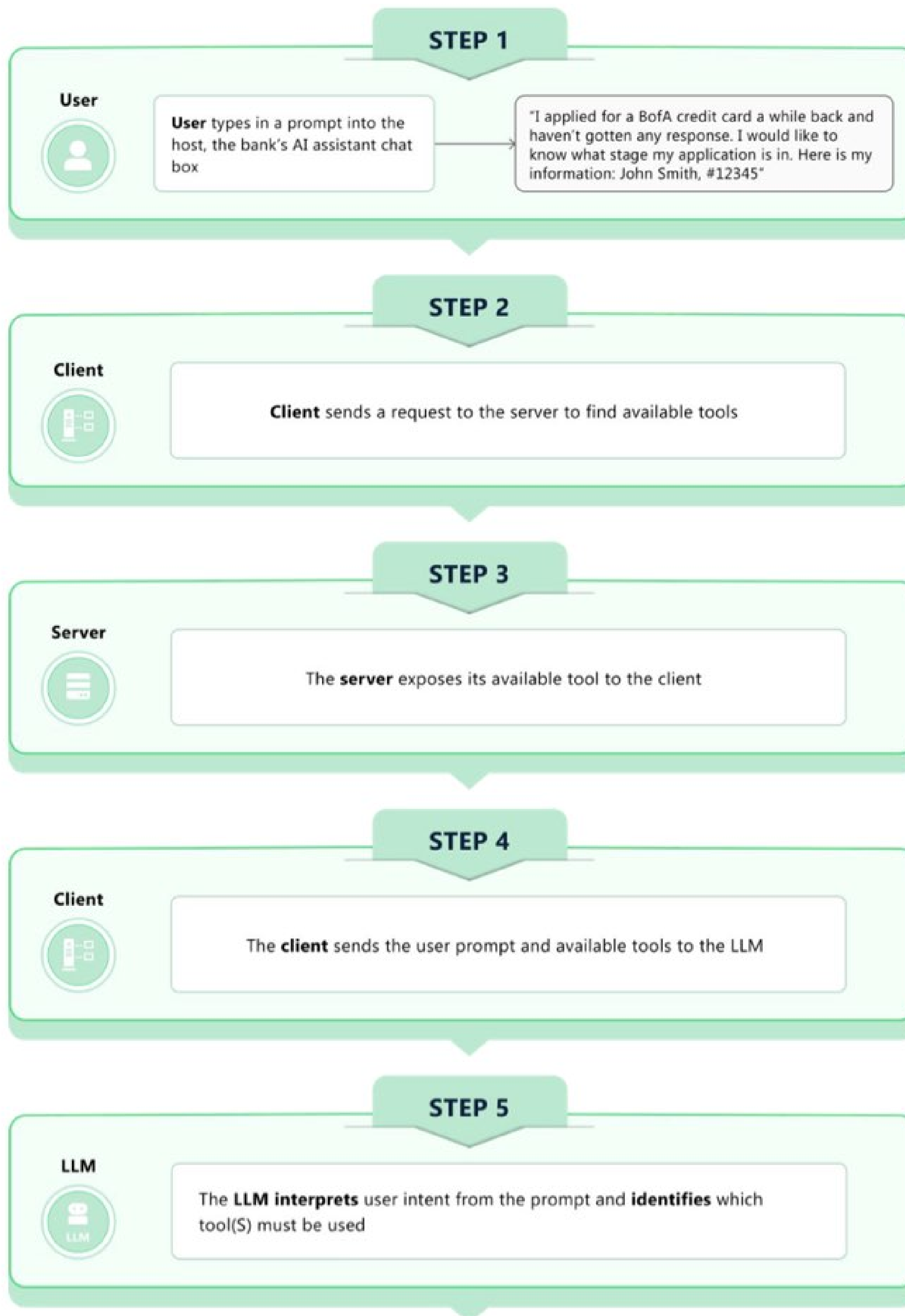
1. User enters a query into the custom UI interface.
2. Client backend receives the user input from the frontend.
3. Client connects to and queries the server for a list of available tools.
4. Client sends the user input + tool list to the LLM for intent interpretation.
5. LLM returns a JSON-formatted tool request with extracted parameters.
6. Client sends the tool request to the backend server for execution.
7. Server creates and executes secure SQL query to internal systems or APIs (e.g., Secure SAP FICO databases such as a BoA database which contains internal company data).
8. Internal system/database returns raw data (e.g., account application status, dates, etc.).
9. Server formats raw data into structured JSON.
10. Server sends structured JSON back to the client.
11. Client forwards JSON to the LLM for summarization.
12. LLM generates a natural language response from structured data.

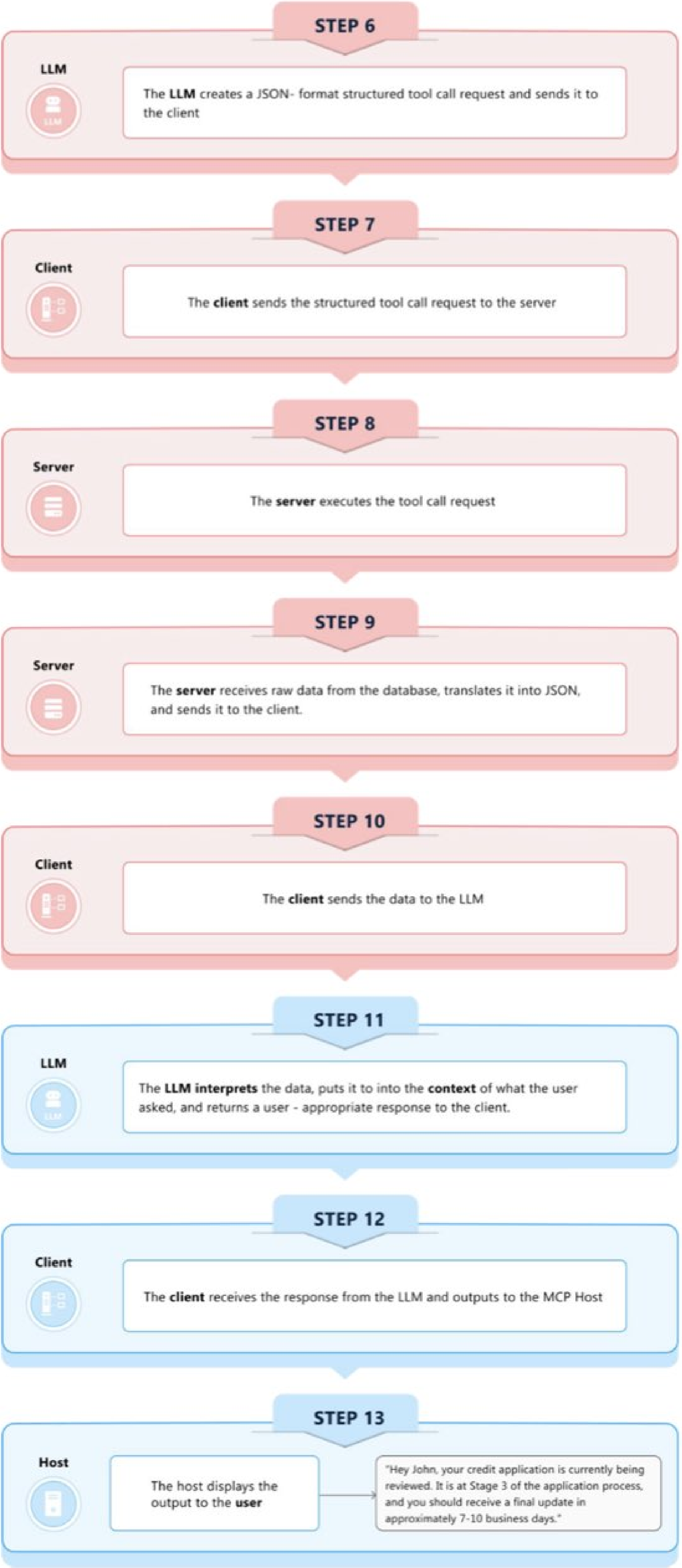
13. LLM returns the response to the client.

14. Client displays the final answer to the user in the UI.

Legend

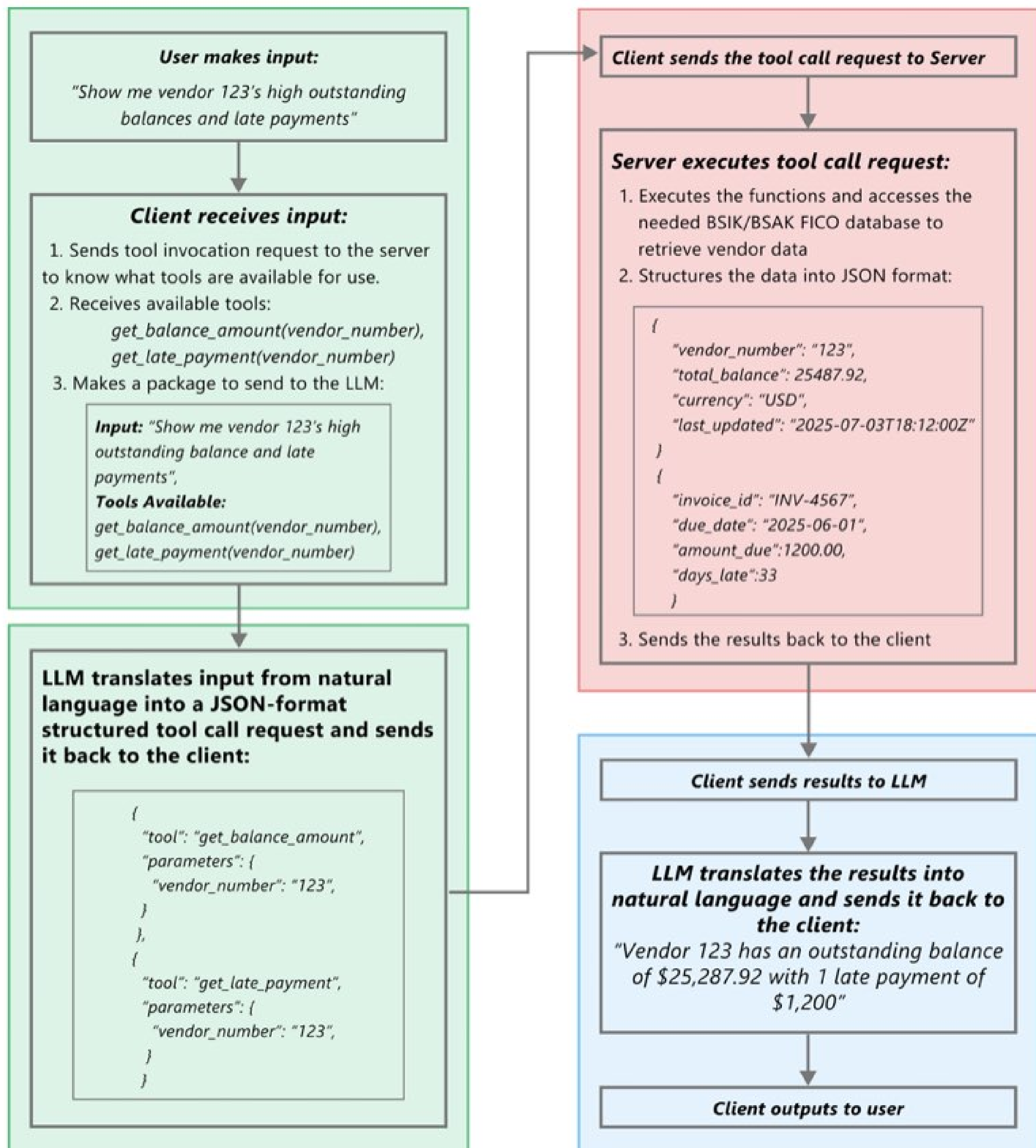
- Stage 1** - Processing user prompt, understanding what tools are needed
- Stage 2** - Executing functions to retrieve the data
- Stage 3** - Processing data and outputting results to the user





1.6.2

FLOWCHART OF MCP: USER INPUT TO OUTPUT



Legend

- Stage 1** - Processing user prompt, understanding what tools are needed
- Stage 2** - Executing functions to retrieve the data
- Stage 3** - Processing data and outputting results to the user

1.6.3

Detailed steps of MCP workflow for data management

Scenario:

You, the user, applied for a Bank of America (BoA) credit card. A month later, you're curious about the application status (approved/denied/in-progress). The status info exists in a private, internal system/database, not directly accessible to you. But the organization uses an LLM-centered MCP system that interfaces with internal tools.

Detailed Workflow: *"What's the status of my BoA credit card application?"*

Step 1

Location: User

Action: The user opens a chatbot or web app and asks:

"Can you check the status of my Bank of America (BoA) credit card application?"

Why: The user wants to retrieve private, dynamic data. This query initiates a process that must bridge human language and secure systems.

How: The user's input is captured by the app's frontend, which sends it as a secure HTTPS request to the client backend API.

Step 2

Location: Client (User Interface)

Action: The client captures input and has to find out what tools are available for use

Why: User input must be forwarded to a logic system that knows how to interpret natural language and decide whether to take action. **The LLM itself doesn't know what tools there are, so the client must find that out** for the LLM.

How: The client sends an authenticated API request to the server's tool registry endpoint to retrieve the list of available tools before invoking any actions.

Step 3

Location: Client → LLM

Action: The client constructs a content package including:

- The original user input

- A list of the available tools

This is sent to the LLM.

Why: The LLM needs to understand the user's intent and suggest the appropriate tool. Since it can't see what tools are there or call the tool itself, it needs this information to create instructions for the next step.

How: The client sends a JSON payload containing the user query and tool metadata via an authenticated HTTPS or gRPC request to the LLM's API endpoint for processing.

Step 4

Location: LLM

Action: The LLM **interprets the user's message** and **uses the list of available tools** to generate a structured tool call request since it cannot call tools directly. It includes the necessary parameters, which it derives from the natural language input, such as:

- bank: Bank of America
- user_id: USR-548923
- date_range: 2025-06-01 to 2025-07-08"

And looks like this:

```
{
  "tool": "get_credit_card_status",
  "parameters": {
    "bank": "Bank of America",
    "user_id": "USR-548923",
    "date_range": "2025-06-01 to 2025-07-08"
  }
}
```

Why: The LLM is used here only for semantic understanding and action suggestion. Because it can not access or call tools directly, it outputs instructions in a format the client can understand.

How: The LLM processes the combined input and tool list, then outputs a JSON-formatted tool call with extracted parameters, returning it via the LLM API response to the client.

Step 5

Location: LLM → Client

Action: The LLM sends the structured tool call request to the client.

Why: The LLM cannot access any tools and thus passes the tool request on to the client, which orchestrates the rest of the action.

How: The LLM returns the JSON tool call in its API response payload, which the client receives and parses for execution.

Step 6

Location: Client → Server

Action: The client sends the structured API tool request, which contains the necessary tools and parameters needed, to the server/backend

Why: The actual execution must be handled by a secure infrastructure with access to private user data. Sending this to a trusted server allows for backend execution of the request.

How: The client sends an authenticated, encrypted API request over HTTPS or gRPC to the server's MCP endpoint, including the tool name and parameters in JSON format.

Step 7

Location: Server → Other Systems

Action: The server receives the request and executes the tool, either by calling a BoA API or querying a secure internal database. The server calls the actual secure database system (e.g., *CardStatusSystem*) with the provided parameters.

Example HTTP API Call:

GET /status-check user_id=USR-548923&bank=boa

Authorization: Bearer user_access_token

Or SQL DB query:

*SELECT * FROM CreditCardApplications*

WHERE user_id = 'USR-548923' AND bank = 'Bank of America';

Why: The server has access to protected systems and can authenticate requests. It is the only layer trusted to retrieve sensitive information securely.

How: The server authenticates the request, then makes secure API calls or executes parameterized SQL queries against internal systems using stored credentials or tokens.

Step 8

Location: Other Systems (BoA backend)

Action: The system looks up the application in its records and returns raw data in rows and columns:

```
application_id  APP-74821
status          In Progress
submission_date 2025-06-01
last_updated    2025-07-08
```

Why: This is the source of truth. Only the database holds accurate, up-to-date info on the application status.

How: The database executes the query and returns the raw result set to the server in its native format (e.g., tabular data or a database-specific response).

Step 9

Location: Server

Action: The server receives the raw data response and formats the it into a structured JSON response:

```
{
  "Application_id": "APP-74821", "Status": "In
  Progress", "Submission_date":
  "2025-06-01", "Last_updated":
  "2025-07-08"
}
```

Why: Structured data ensures that the LLM or client can correctly process and present the results to the user. Raw backend data may be inconsistent or in more words than needed.

How: The server uses its database driver to fetch query results, maps the columns to keys, and serializes the data into JSON format using standard libraries before sending it.

Step 10

Location: Server → Client

Action: The structured JSON response is sent back to the client that requested it.

Why: The client knows where to send the JSON response and orchestrates the communication between the server and the LLM.

How: The server sends the JSON response over a secure, authenticated HTTPS or gRPC connection back to the client's MCP listener endpoint.

Step 11

Location: Client → LLM

Action: The client forwards the structured JSON response to the LLM.

Why: The LLM can't directly interact with the database or the server, but it can translate JSON data into something that explains the results clearly and naturally.

How: The client includes the JSON response in an API request or internal call to the LLM's endpoint for natural language processing.

Step 12

Location: LLM

Action: The LLM processes the JSON data and generates a final response: "Your Bank of America credit card application, submitted on June 1st, is still in progress as of July 8th."

Why: The LLM translates machine data into user-friendly language, making the system accessible and useful for non-technical users.

How: The LLM uses prompt templates or fine-tuned instructions to convert the JSON input into a clear, natural language summary before returning it to the client.

Step 13

Location: LLM → Client

Action: The LLM forwards the natural language output to the client.

Why: The LLM can't communicate with the user; only the client can orchestrate communication between the two.

Step 14

Location: Client → User

Action: The client receives the LLM's natural language response message and displays it in the original UI.

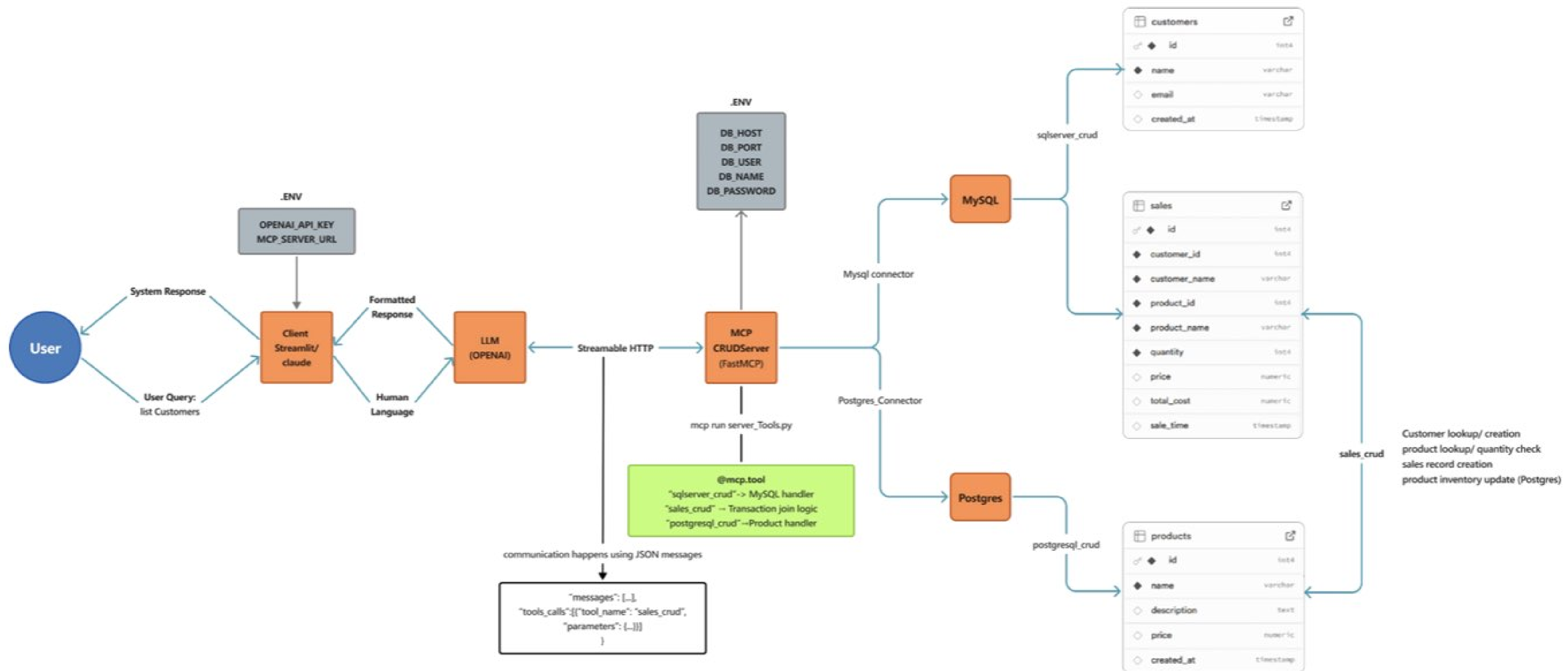
Why: The final output must be delivered to the user in the interface they started with.

1.6.4 ENTITY ROLES SUMMARY

Entity	Function
User	Makes natural language request
Client (MCP)	Sends prompts, interprets LLM output, manages tool execution
LLM	Interprets user intent and formats instructions or summaries in text
Server	Executes the requested tool/API function
Other Systems	Contains the actual credit application data

1.7

CUSTOM MCP CLIENT/ SERVER ARCHITECTURE





PART- 2

IMPLEMENTATION: MCP FOR ENTERPRISE DATA AND INSIGHTS

This section focuses on the implementation side of MCP, specifically how to build both standard and custom MCP clients that interact with databases through natural language. Here, we guide you through both a high-level summary and a detailed breakdown of the implementation steps, showing you what's required to create a working MCP client from scratch or using existing components.

We center this explanation around a practical business use case, and what our business model is based on:

2.1 SCENARIO

Your business maintains a database with tables containing **customer**, **product**, and **sales** data. The customer table includes fields for an auto incremented customer ID number, their first, last and full names, their email, and a timestamp for when they were created. The product table includes fields for an auto incremented product ID number, product name, price, and description. The sales table includes fields for an auto incremented sales ID number, the customer who bought it, the product that was bought, the quantity of the product bought, the unit price of the product, the total sale price, and a timestamp for when the sale was made. You want users, especially non-technical ones, to interact with this database using natural language instead of SQL so that they can manage your business's customers, product and sales.

For example, let's simply take a look at how a new customer can be added. Rather than writing out **INSERT INTO Customers VALUES ('John Smith')**, a user can simply say, *"Create customer John Smith email johnsmith@gmail.com"* and the LLM, with the help of MCP, converts this into an executable SQL query and routes it to the appropriate database, creating a new customer with the name John Smith and assigning him an email address of *johnsmith@gmail.com*. Thus, you are able to leverage conversational intelligence to easily and efficiently manage your database.

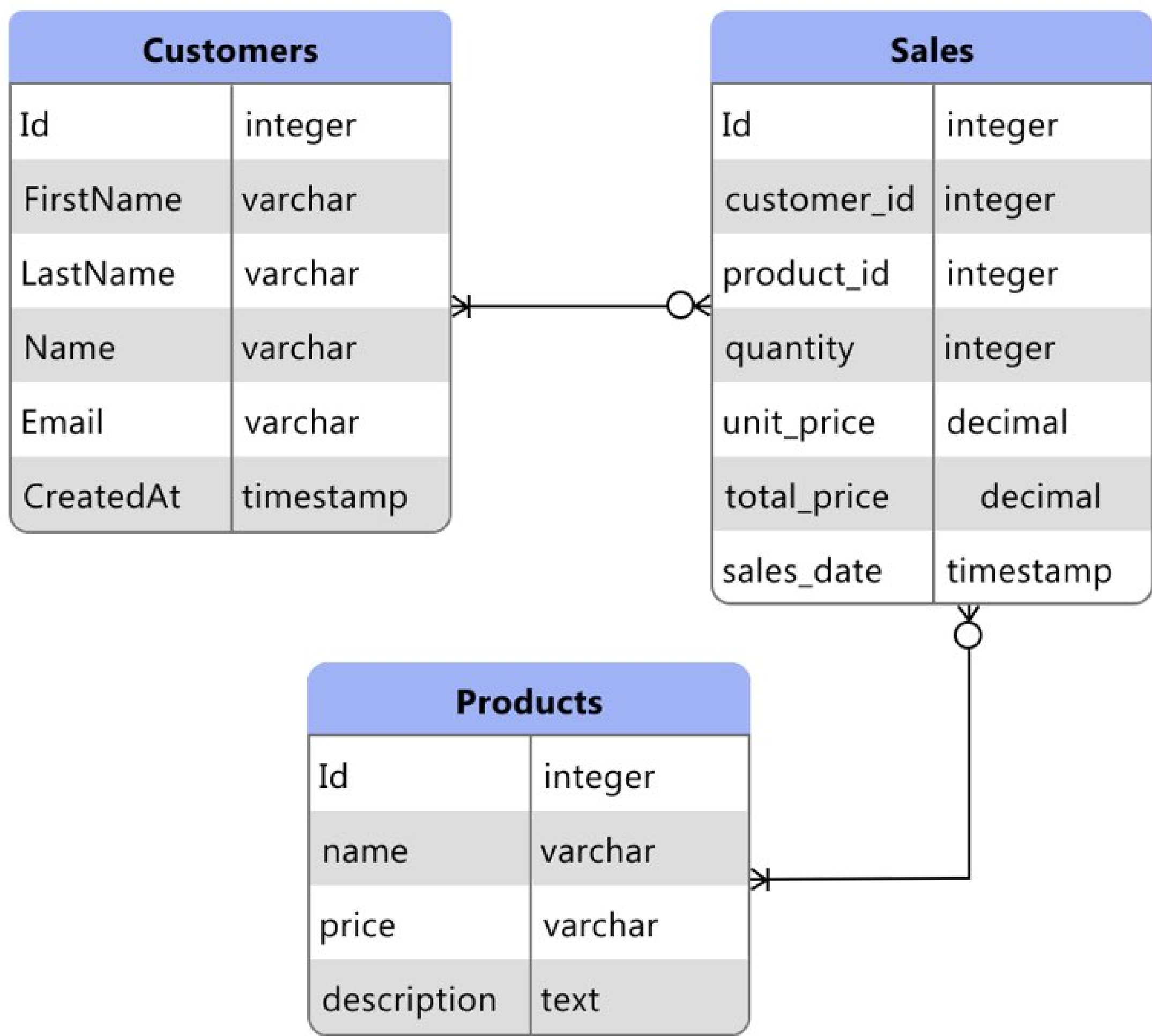
The implementation involves exposing core database management operations such as insert, update, and delete, allowing users to modify data with conversational commands. In addition, we integrate a set of ETL (Extract, Transform, Load) functions—essential for cleaning and transforming raw data into usable formats. We include:

1. String concatenation: to combine fields like first and last names.
2. Date conversion: to reformat timestamps (e.g., converting 2025-08-03T14:52:00Z to Aug 3, 2025, 2:52 PM).

- 3. Decimal rounding: to clean up price values (e.g., from 36.97777 to 36.97).
- 4. Null value handling: to replace or flag missing data entries.

By the end of this section, readers will understand how to set up MCP clients that translate everyday language into precise, database-ready actions, enabling smarter, easier access to structured data.

Here is an **Entity Relationship Diagram** to help visualize how the **Customer**, **Sales**, and **Product** tables in the database are related:



2.2

MCP CLIENT CONFIGURATION AND SETUP

This section outlines how to set up two types of MCP clients: a standard one using Claude Desktop and a custom one with a Streamlit App.

The custom MCP client and server code that has been developed by us can be found here:

GitHub repo link: https://github.com/LLM-AI-INDIA/LLM_MCP_Project

2.2.1

Section 1: Summary of steps

2.2.1.1 – STANDARD CLIENT - CLAUDE DESKTOP

Step 1 Install and sign-in

- **Purpose:** Acquire the official desktop client.
- **Objective:** Have Claude Desktop running under your account.
- **Action:** Download from Anthropic, install, and log in.

Step 2 Install MCP Endpoint

- **Purpose:** Load your server's tool manifest into Claude Desktop.
- **Objective:** Register the SQL Server and PostgreSQL tools from main.py.
- **Action:** Run mcp install main.py (with Claude Desktop open) to auto-install the endpoint.

Step 3 Activate Tools

- **Purpose:** Expose DB tools in the assistant.
- **Objective:** Enable SQL Server and PostgreSQL groups.
- **Action:** Toggle both tools sourced from `Server_Tools.py`.

Step 4 Confirm Connectivity

- **Purpose:** Verify database access.
- **Objective:** Rely on env-var credentials already set.
- **Action:** Proceed if no connection errors appear.

Step 5 Run CRUD Prompts

- **Purpose:** Perform natural-language operations.
- **Objective:** Create, read, update, delete sample data.
- **Action:** Issue prompts like "Create customer Charlie..."

Before proceeding, ensure you have set up the MCP server by following the steps in 2.3.1.1 Hosting Server – render.com.

2.2.1.2 – CUSTOM CLIENT (Referencing 2.3.1.1 Hosting Server – render.com)

BUILD

1. Install dependencies

- **Purpose** – Provision required libraries.
- **Objective** – Ensure `client.py` imports in `client1.py` resolve.
- **Action** – `pip install -r requirements.txt`

2. Verify local launch

- **Purpose** – Catch Runtime errors early.
- **Objective** – Confirm the interface renders.
- **Action** – `streamlit run client.py`

HOST

3. Push code to GitHub

- **Purpose** – Make the project deploy-able

- **Objective** – Store `client.py`, `requirements.txt`
- **Action** – Commit and push to a public repo.

4. Deploy the app

- **Purpose** - Spin up a managed instance
- **Objective** – Let Streamlit Cloud build the image.
- **Action** – *New app -> choose repo -> Deploy*

5. Add Secrets

- **Purpose** – Safeguard credentials.
- **Objective** – Supply `OpenAI_API_KEY` and `MCP_SERVER_URL` expected by `client.py`
- **Action** – *Settings -> Secrets -> paste key-value pair*

RUN

6. Open URL

- **Purpose** – Access the live URL
- **Objective** – Reach **`https://<your-app-name>.streamlit.app`**.
- **Action** – Visit the link in any browser

7. Submit Prompts

- **Purpose** – Drive CRUD via natural language
- **Objective** – Sends request to MCP server through the text box
- **Action** – Enter commands like "Create a customer named Charlie ...".

2.2.2

Section 2: Detailed Implementation Steps

2.2.2.1 – STANDARD CLIENT - CLAUDE DESKTOP

Step 1 Install and Sign-In

- **Purpose** – Acquire the official Claude Desktop Client
- **Objective** – Have Claude Desktop installed and running under your own Anthropic account.
- **Action** –
 1. Navigate to the official [Anthropic Claude Desktop](#) download page.
 2. Download the desktop application for your OS (Windows/macOS).
 3. Install the application following the on-screen instruction.
 4. Launch Claude Desktop and sign in using your Anthropic account credentials.
 5. Keep the client open – this is required for later steps like endpoint registration.

Step 2 Install MCP Endpoint

- **Purpose** – Load your local MCP server tool definitions into Claude
- **Objective** – Register the SQL server and PostgreSQL tool interfaces from your code.
- **Action** –
 1. Ensure that the MCP server file (`Server_Tools.py`) is fully working and follows the MCP tools creation structure
 2. Open a terminal in the directory where your server code resides.
 3. Run the following command `mcp install Server_Tools.py`

```

• (SQL MCP Server) PS C:\Users\hrush\OneDrive\Desktop\SQL\SQL MCP Server> mcp install main.py
[07/03/25 21:18:30] INFO      Added server 'MultiDBCRUD' to Claude config
                        INFO      Successfully installed MultiDBCRUD in Claude app
○ (SQL MCP Server) PS C:\Users\hrush\OneDrive\Desktop\SQL\SQL MCP Server>

```

4. This command uses introspection to extract your tool definitions and sends them to Claude Desktop, which should already be running.

Step 3 Activate the Tools

- **Purpose** – Enable the registered tools for use inside Claude’s assistant window
- **Objective** – Make SQL Server and PostgreSQL tools groups visible and callable via Claude’s interface
- **Action** –
 1. In Claude Desktop, navigate to the tools tab or sidebar
 2. Locate the tools that were just registered (e.g., sqlserver_crud, postgresql_crud, sales_crud)
 3. Manually toggle or enable each of the required tools
 4. Confirm that the tools are visible and active for the Claude assistant session.

Step 4 Confirm Connectivity

- **Purpose** – Validate connections to the underlying databases through the active tools.
- **Objective** – Ensure that Claude can successfully invoke tools without any connection issues.
- **Action** –
 1. Verify that all required environment variables are already set before the server was started
 - a. SQL Server: SQL_SERVER, SQL_USER, SQL_PASSWORD, DB_NAME etc.
 - b. Postgresql: PG_HOST, PG_PORT, PG_USER, PG_PASSWORD, PG_DB
 2. Start the MCP server if it’s not already running

```
mcp install Server_Tools.py
```
 3. Back in Claude Desktop, perform a quick test (e.g., try listing customers or products)
 4. If no errors appear in the Claude interface and server logs show correct handling, connectivity is confirmed.

Step 5 Run CRUD Prompts

- **Purpose** - Test the full natural language interface backed by your custom tools.
- **Objective** – Create, read, update, and delete records using conversational prompts
- **Action** –
 1. In Claude Desktop, enter prompts in natural language
"Create a customer named Charlie with email charlie@example.com"
 2. Read customers:
"Show me up to five customers."
 3. Delete a customer:
"Delete customer with ID 2".

Create Product table in Postgres Database

H can you create product table

I'll create a product table for you. Let me use the PostgreSQL database to set this up.

M postgresql_crud

Let me check what operations are available by trying to read the existing products:

M postgresql_crud

Great! It looks like a products table already exists in the PostgreSQL database with the following structure:

- **id:** Product identifier
- **name:** Product name
- **price:** Product price
- **description:** Product description

The table currently contains 2 products:

1. Widget (\$9.99) - A standard widget
2. Gadget (\$14.99) - A useful gadget

Add a new record headphone in product table

show me in table format

Product Records Table

Interactive artifact

I've created a formatted table showing all your product records. The table displays:

- **ID:** Unique Identifier for each product
- **Product Name:** Name of the product
- **Price:** Product price in USD
- **Description:** Brief description of the product

Currently, you have 3 products in the database:

1. Widget (\$9.99)
2. Gadget (\$14.99)
3. Headphone (\$119.99) - the one we just added

Would you like me to add more products, update existing ones, or perform any other operations on the product data?

Product Records

ID	PRODUCT NAME	PRICE	DESCRIPTION
1	Widget	\$9.99	A standard widget.
2	Gadget	\$14.99	A useful gadget.
3	Headphone	\$119.99	High-quality headphones.

Total products: 3

Create Customer table in SQL Server

can you create customer table

I'll create a customer table for you. Let me use the PostgreSQL database to set this up.

M postgresql_crud

Let me check what operations are available by trying to read the existing products:

M postgresql_crud

Great! It looks like a products table already exists in the PostgreSQL database with the following structure:

- **Id:** Customer identifier
- **name:** Customer name
- **price:** Customer price
- **description:** Customer description

The table currently contains 2 customers:

Add a new record in customer table

SQL Customer Table Creation

Add a record name Hrushikesh with email id hrushikesh@gmail.com

I'll add a new customer record for Hrushikesh with the email hrushikesh@gmail.com:

id sqlserver_crud

Perfect! I've successfully added Hrushikesh to the customer table. Let me update the table display to show all records including the new one:

id sqlserver_crud

Customer Records Table

Interactive artifact - Version 2

Excellent! I've successfully added Hrushikesh to the customer database. The new record has been created with:

- **ID:** 3 (automatically assigned)
- **Name:** Hrushikesh
- **Email:** hrushikesh@gmail.com
- **Created Date:** July 4, 2025 at 9:46 PM

Customer Records

ID	NAME	EMAIL	CREATED DATE
1	Alice	alice@example.com	July 4, 2025 9:27 PM
2	Bob	bob@example.com	July 4, 2025 9:27 PM
3	Hrushikesh	hrushikesh@gmail.com	July 4, 2025 9:46 PM

Total records: 3

Before you begin with the detailed steps for the custom client, ensure you have completed the server setup outlined in 2.3.2.1 Hosting Server– render.com.

2.2.2.2 – CUSTOM CLIENT – STREAMLIT APP (Referencing 2.3.2.1 Hosting Server– render.com)

BUILD

Step 1 Install dependencies Locally

- **Action** – Provision required Python Libraries
- **Objective** – Ensure that all the imports in client.py resolve properly
- **Action** –
 1. Ensure your requirements.txt includes all the necessary packages streamlit, openai, requests, factmcp
 2. In your project directory, open a terminal and run:

```
pip install -r requirements.txt
```

Step 2 Verify the Local Launch

- **Purpose** – Catch errors before deployment
- **Objective** – Confirm that the Streamlit app launches and renders correctly
- **Action** –
 1. Run the Streamlit app locally using:

```
streamlit run client.py
```
 2. A browser window should open automatically with your UI. If not manually visit the website <https://localhost:8501>
 3. Confirm that
 - a. The UI loads without errors.
 - b. You can input prompts
 - c. MCP server connection.

HOST

Step 3 Push the code to GitHub

- **Purpose** – Make the project deployable from the cloud.
- **Objective** – Upload your application code and metadata.
- **Action** –
 1. Ensure these files are included:
 - a. `Client.py`
 - b. `Requirements.txt`
 2. Initialize and push to GitHub. Commands are as follows

```
git init
```

```
git remote add origin
```

```
https://github.com/yourusername/mcp-streamlit-client
```

```
git add .
```

```
git commit -m "Initial client commit"
```

```
git push -u origin main
```

Step 4 Deploy the App

- **Purpose** – Launch the app on Streamlit Cloud.
- **Objective** – Let Streamlit manage your infrastructure and run the app continuously.
- **Action** –
 1. Go to <https://streamlit.io/cloud> and sign in.
 2. Click **"New app"**.
 3. Choose the GitHub repository and branch containing your client code.
 4. Specify the file to run (e.g., `client.py`).
 5. Click **"Deploy"**.

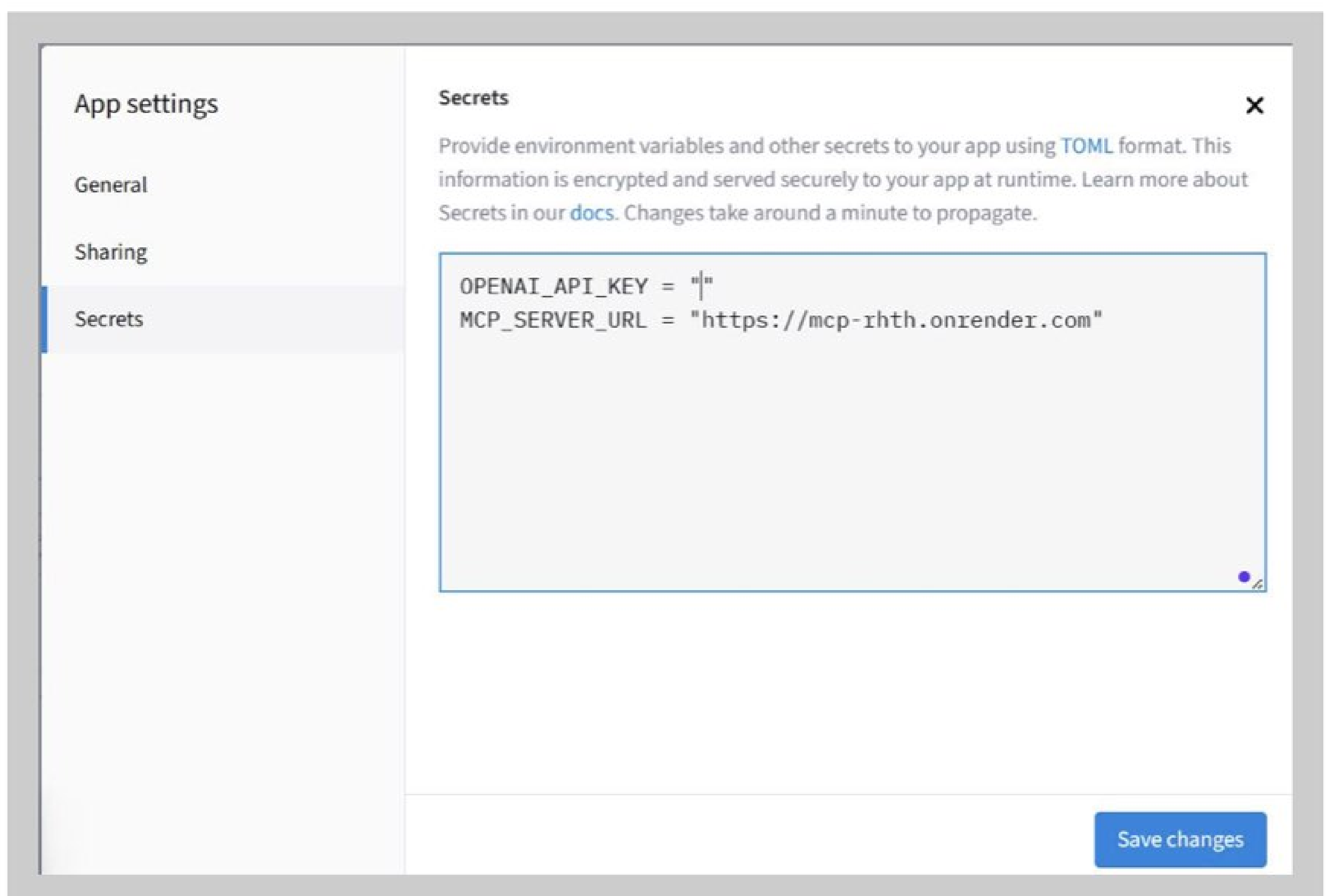
6. Streamlit Cloud will automatically:

- Clone your repo
- Install dependencies via `requirements.txt`
- Run the app using `streamlit run client.py`

Step 5 Add Secrets

- **Purpose** - Securely manage credentials
- **Objective** – Provide runtime values like your API keys and MCP server URL.
- **Action** –
 1. In the streamlit Cloud app dashboard, go to settings -> Secrets/
 2. Paste the following key-value format

```
OPENAI_API_KEY = "your_openai_key_here"
MCP_SERVER_URL = https://your-mcp-server.onrender.com (the server link hosted on render.com)
```



RUN

Step 6 Open the URL

- Purpose – Access the deployed UI from the web
- Objective – Reach the hosted version of the app.
- Action –
 1. Once the deployment is complete, Streamlit provides a public URL like <https://your-app-name.streamlit.app>
 2. Open this link in any browser to access your live UI.

Step 7 Submit prompts

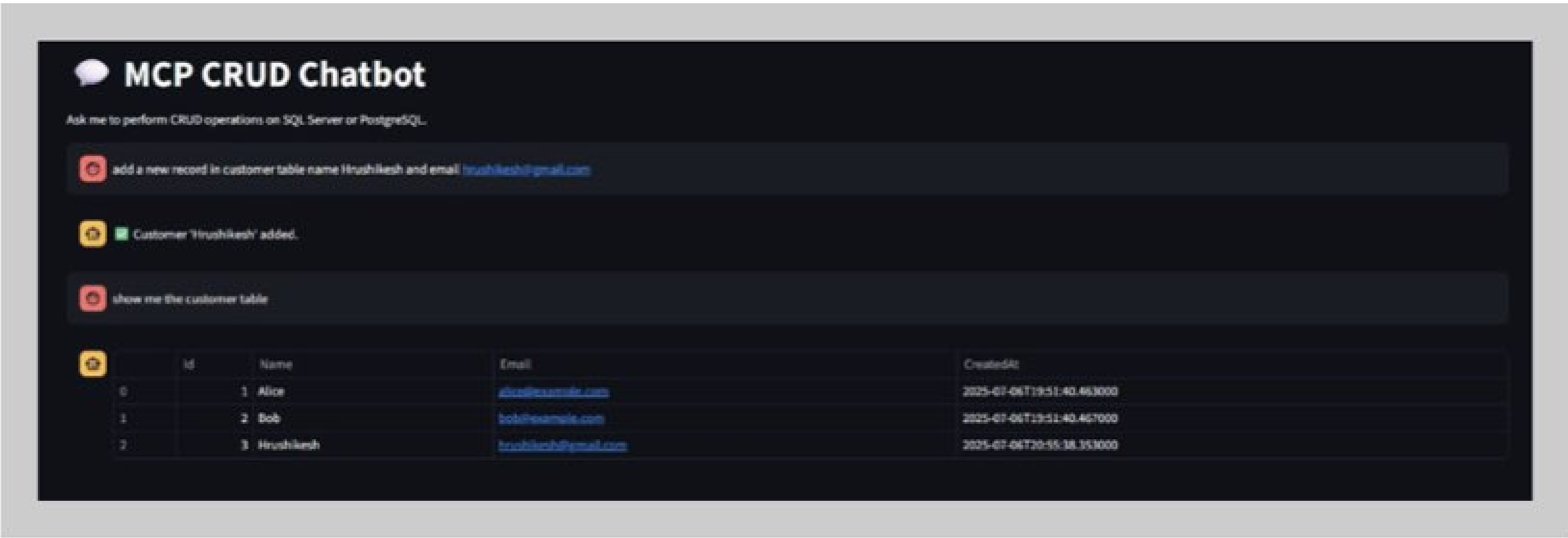
- Purpose – Interact with the backend MCP server via natural language.
- Objective – Allow the user to perform CRUD operations through the client interface.
- Action –
 1. In the app, use the text box to enter the commands like:

Create a customer named Charlie with email charlie@example.com

Show me five products

Update the customer ID 1 to have email charlie@newdomain.com

Add a new record in customer table with streamlit app as a client



Add a new record in product table with streamlit app as a client

add a new record name headphones with price 199.99 in the product table

Product 'headphones' added.

show me the product table

	id	name	price	description
0	1	Widget	9.9900	A standard widget.
1	2	Gadget	14.9900	A useful gadget.
2	3	headphones	199.9900	

2.3

MCP SERVER CONFIGURATION AND SETUP

This section provides a detailed guide on how to configure and deploy the MCP server.

2.3.1

Section 1: Summary of steps

2.3.1.1 – HOSTING SERVER-RENDER.COM

Step 1 Prepare Project for Deployment

- **Purpose:** Organize code for cloud compatibility
- **Objective:** Ensure necessary files (`Server_Tool.py`, `requirements.txt`) are in the repo
- **Action:** Push project to GitHub with all server-related files

Step 2 Configure Environment Variables on Render

- **Purpose:** User must enter your own database credentials.
- **Objective:** Define SQL Server and PostgreSQL parameters via environment variables.
- **Action:** Add secrets via Render's Environment tab.

Step 3 Define Build and Start commands

- **Purpose:** Allow render to install and run the servers
- **Objective:** Use Render's build system to execute the correct startup script
- **Action:** Set build command to `pip install -r requirements.txt` and start command to `python Server_Tools.py`

Step 4 Deploy and Retrieve Server URL

- **Purpose:** Get publicly accessible MCP endpoint
- **Objective:** Enable clients to interact with the hosted server
- **Action:** Deploy and copy Render URL to use in your Streamlit app.

2.3.2

Section 2: Detailed Implementation Steps

2.3.2.1 – HOSTING SERVER -RENDER.COM

Step 1 Prepare Project for Deployment

- **Purpose:** Organize code for cloud compatibility.
- **Objective:** Ensure necessary project files are available in a public or private GitHub repository.
- **Action:**
 1. Ensure the following files exist in your project root:
 - `Server_Tools.py` – contains your MCP server and tool definitions.
 - `requirements.txt` – lists all dependencies (including `fastmcp`, `mcp[cli]`, `pyodbc`, `psycopg2-binary`, `openai`, `streamlit`).
 2. Create a new GitHub repository (public or private).
 3. Commit all project files and push them to the GitHub repo:

Commands are as follows:

```
git init
```

```
git remote add origin
```

```
https://github.com/yourusername/mcp-server
```

```
git add .
```

```
git commit -m "Initial commit"
```

```
git push -u origin main
```

Step 2 Configure Environment Variables on Render

- **Purpose:** Set up secure access to your databases and LLM API.

- **Objective:** Provide runtime credentials through Render's Environment settings.
- **Action:**
 1. Go to <https://dashboard.render.com>.
 2. Create a new **Web Service** and link it to your GitHub repository.
 3. Once the service is created, go to the **Environment** tab in Render.
 4. Add the following key-value pairs:

Key	Value		
MYSQL_DB		👁	🗑
MYSQL_HOST		👁	🗑
MYSQL_PASSWORD		👁	🗑
MYSQL_PORT		👁	🗑
MYSQL_USER		👁	🗑
OPENAI_API_KEY		👁	🗑
PG_DB		👁	🗑
PG_HOST		👁	🗑
PG_PASSWORD		👁	🗑
PG_PORT		👁	🗑
PG_SSLMODE		👁	🗑
PG_USER		👁	🗑

Step 3 Define Build and Start commands

- **Purpose** - Enable Render to automatically install dependencies and run your server.
- **Objective** - Leverage Render's build lifecycle for a smooth deployment.
- **Action** -

1. In the Build Command field enter:

```
pip install -r requirements.txt
```

2. In the Start Command field enter:

```
python Server_Tools.py
```

3. Make sure the code includes:

```
if __name__ == "__main__":  
    mcp.run(transport = "streamable-http")
```

Step 4 Deploy and Retrieve Server URL

- **Purpose:** Launch your MCP server on a live, publicly accessible endpoint.
- **Objective:** Allow external clients (e.g. Claude or Streamlit) to connect to your hosted server.
- **Action:**
 1. Click **Deploy** on the Render dashboard.
 2. Wait a few minutes for the build and startup processes to complete.
 3. Once deployed, you'll see a **live URL** in the top bar of the Render service dashboard (e.g., <https://mcp-server.onrender.com>).
 4. Copy this URL for use in your clients:
 - In Claude: register the endpoint
 - In Streamlit: set it in `MCP_SERVER_URL` inside `secrets.toml` or environment config

2.4 ETL FUNCTIONS DEMO OF MCP

UI

Interface showcasing application, protocol, LLM and server tool selection for ETL- free data exchange and cloud deployment.

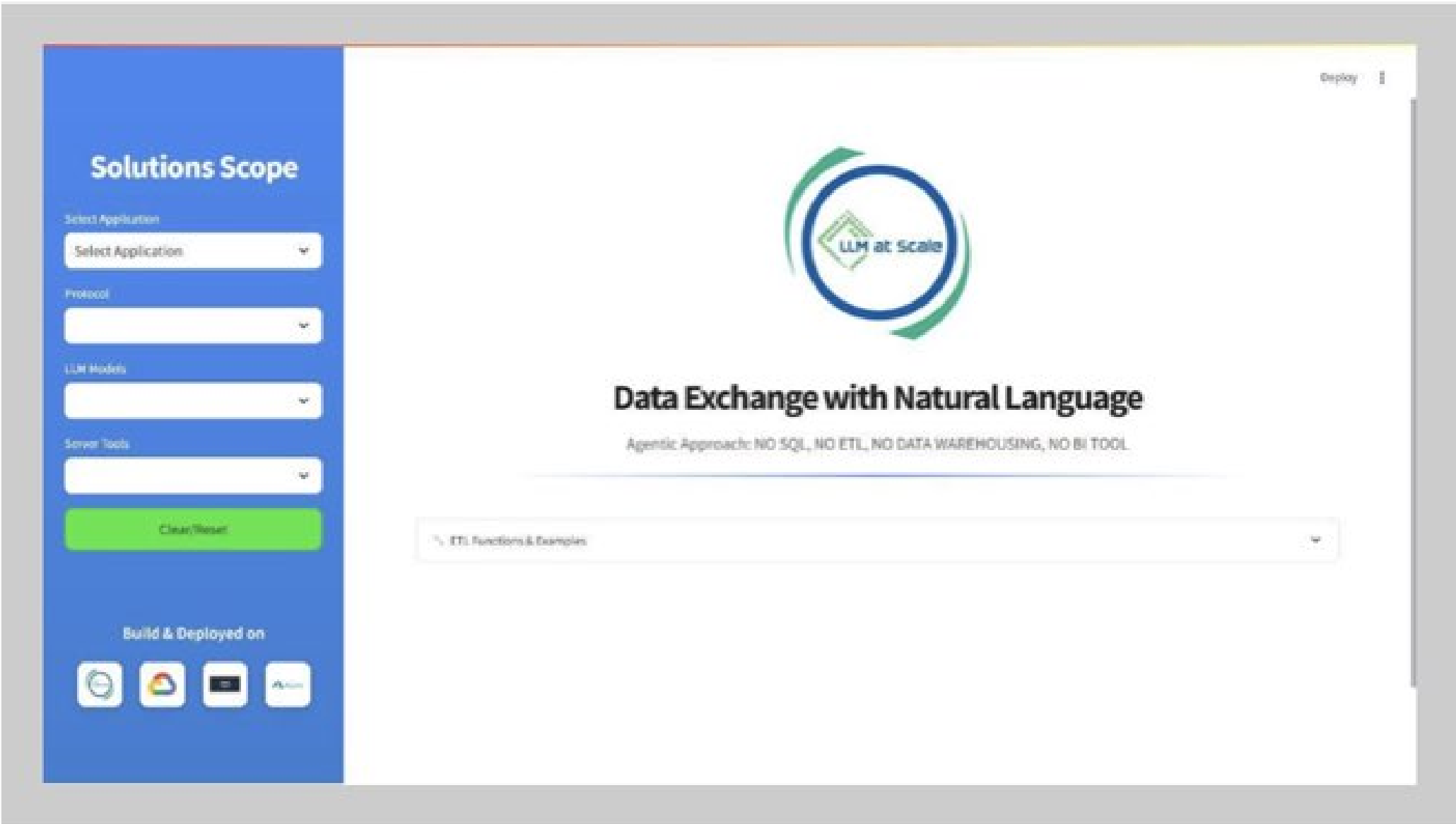


Figure 2.3.1: Application Interface

READ

A read query was executed on the database, successfully retrieving three product records. Each record contained detailed attributes including the product name, price, and description.

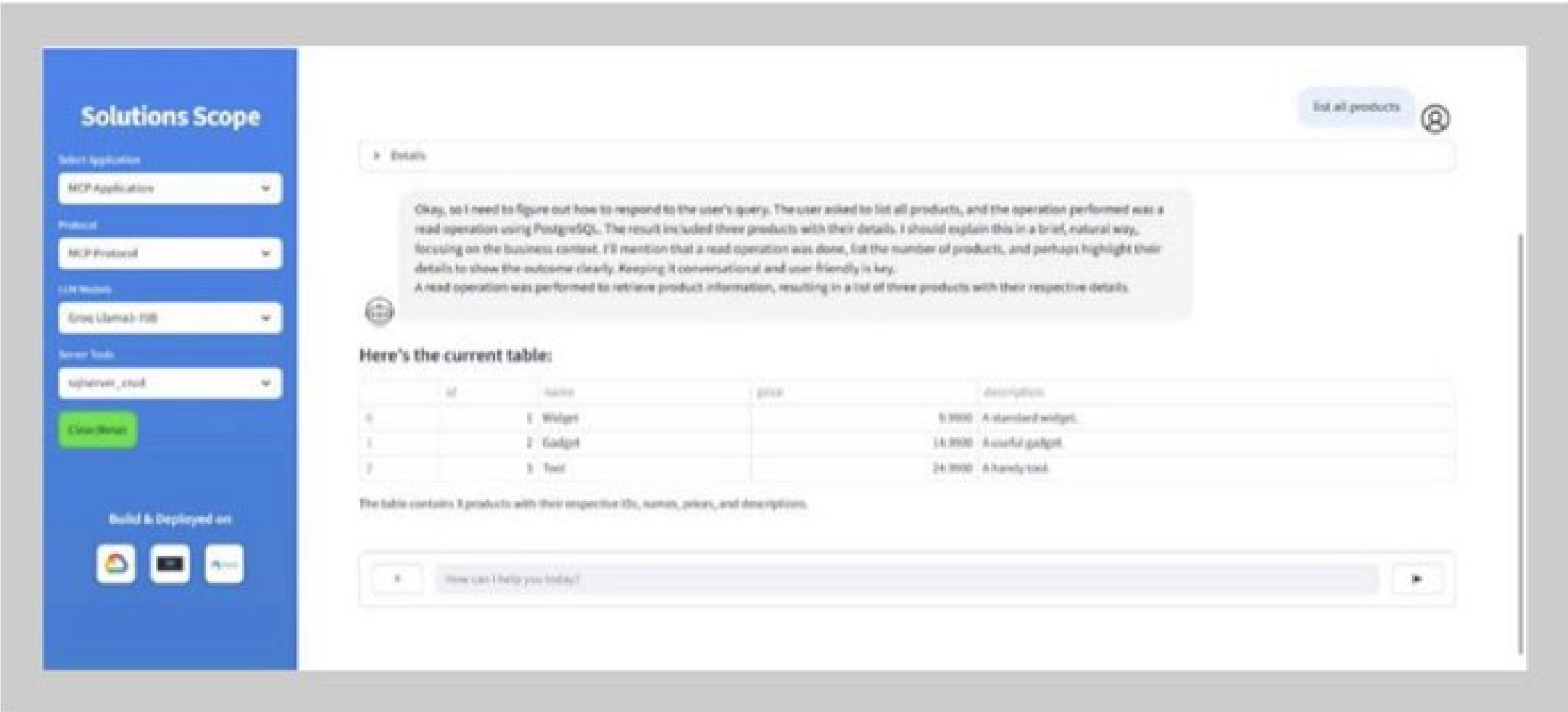


Figure 2.3.2: Product Records Retrieval Via SQL Query

CREATE

A new customer entry was inserted into the customer database, adding **John Doe** with the email address john@mail.com. Following this insertion, the total number of customers increased from **three to four**, confirming the successful creation of the record.

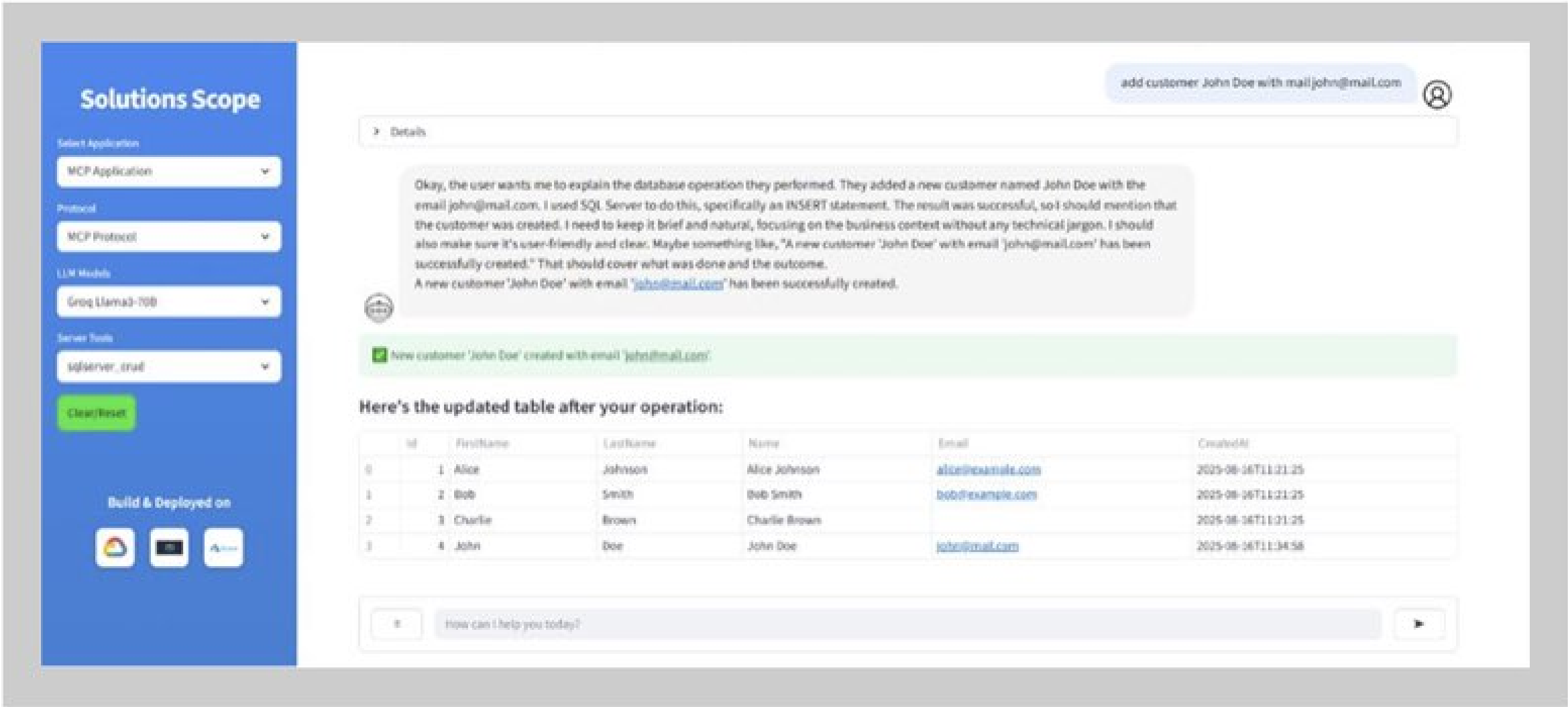


Figure 2.3.3: Customer Insertion With Record Count Update

UPDATE

The email address of **Alice Johnson** was updated to alice@hotmail.com. The modification was applied seamlessly without affecting the overall customer count, which remains at **four**.

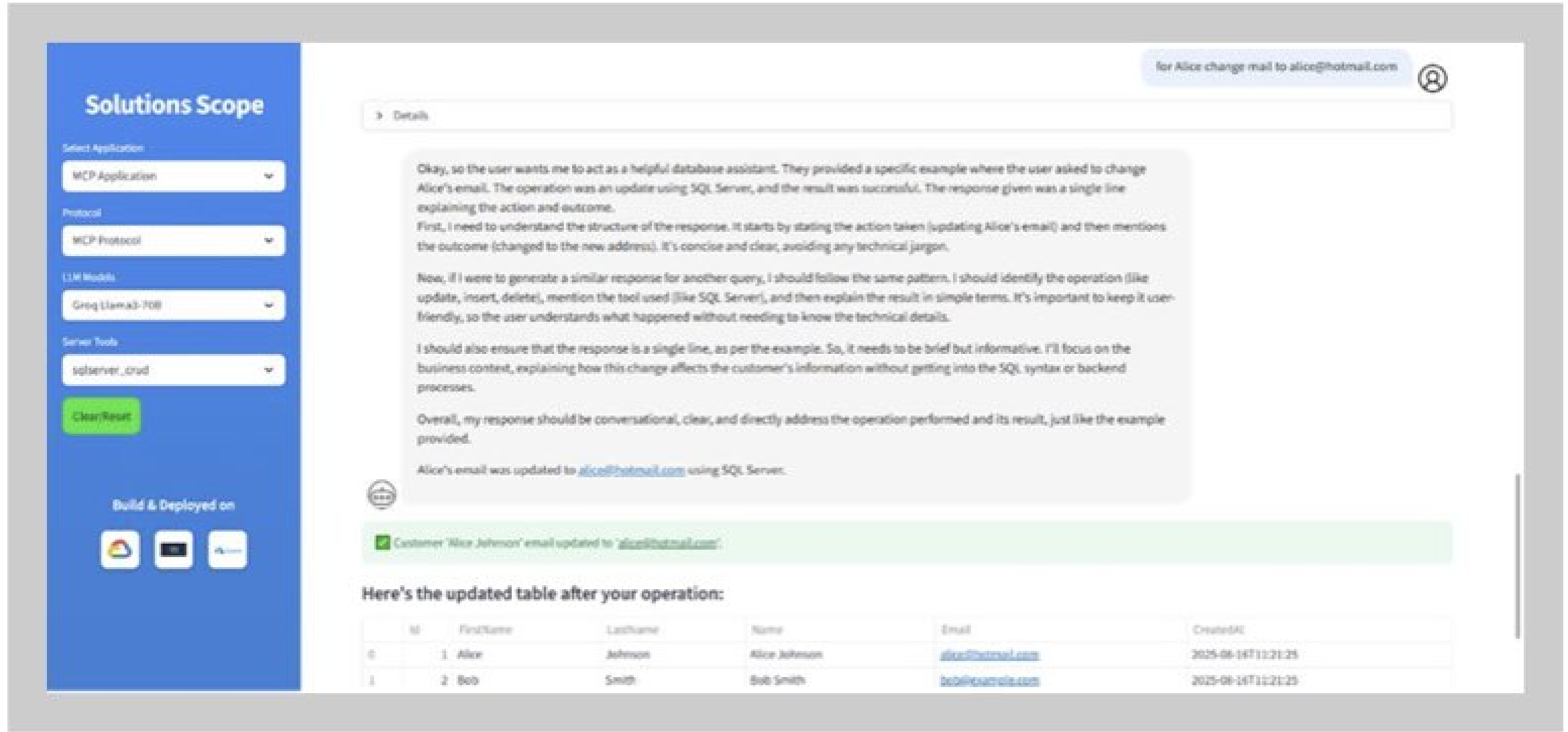


Figure 2.3.4: Email Update In Customer Dataset

DELETE

The product with **ID = 1** (Widget) was removed from the database. This deletion reduced the total number of products from **three to two**, ensuring the database reflects accurate inventory status.

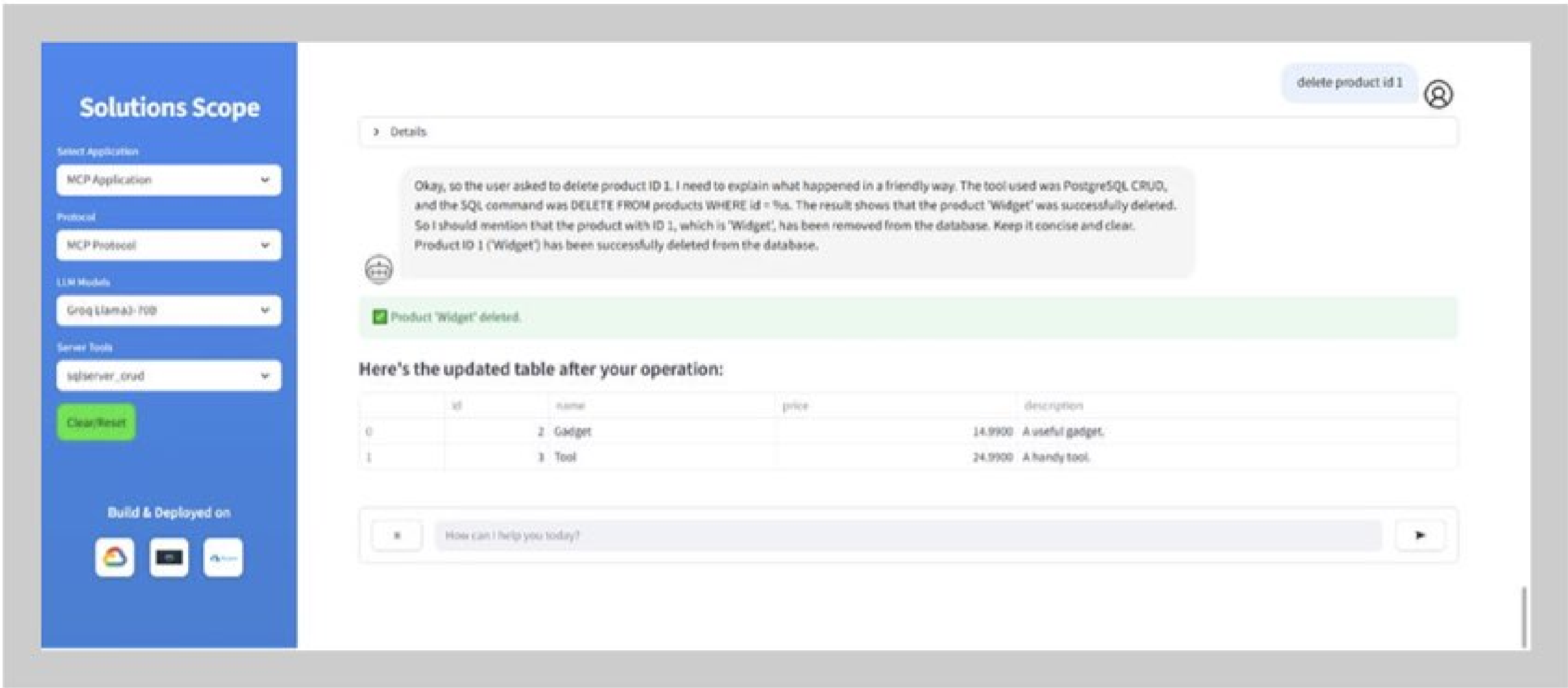


Figure 2.3.5: Product Deletion Reflecting Inventory Change

ETL Data Format Conversion

Sales data was extracted and reformatted to highlight critical information such as customer names, product names, quantities, unit prices, total prices, transaction dates, and customer email addresses. A standardized date format was applied for improved readability and consistency.

The sales data was generated by performing a **join operation** across the **Sales**, **Customers**, and **Products** tables, effectively consolidating customer and product details into a unified view of each sales transaction

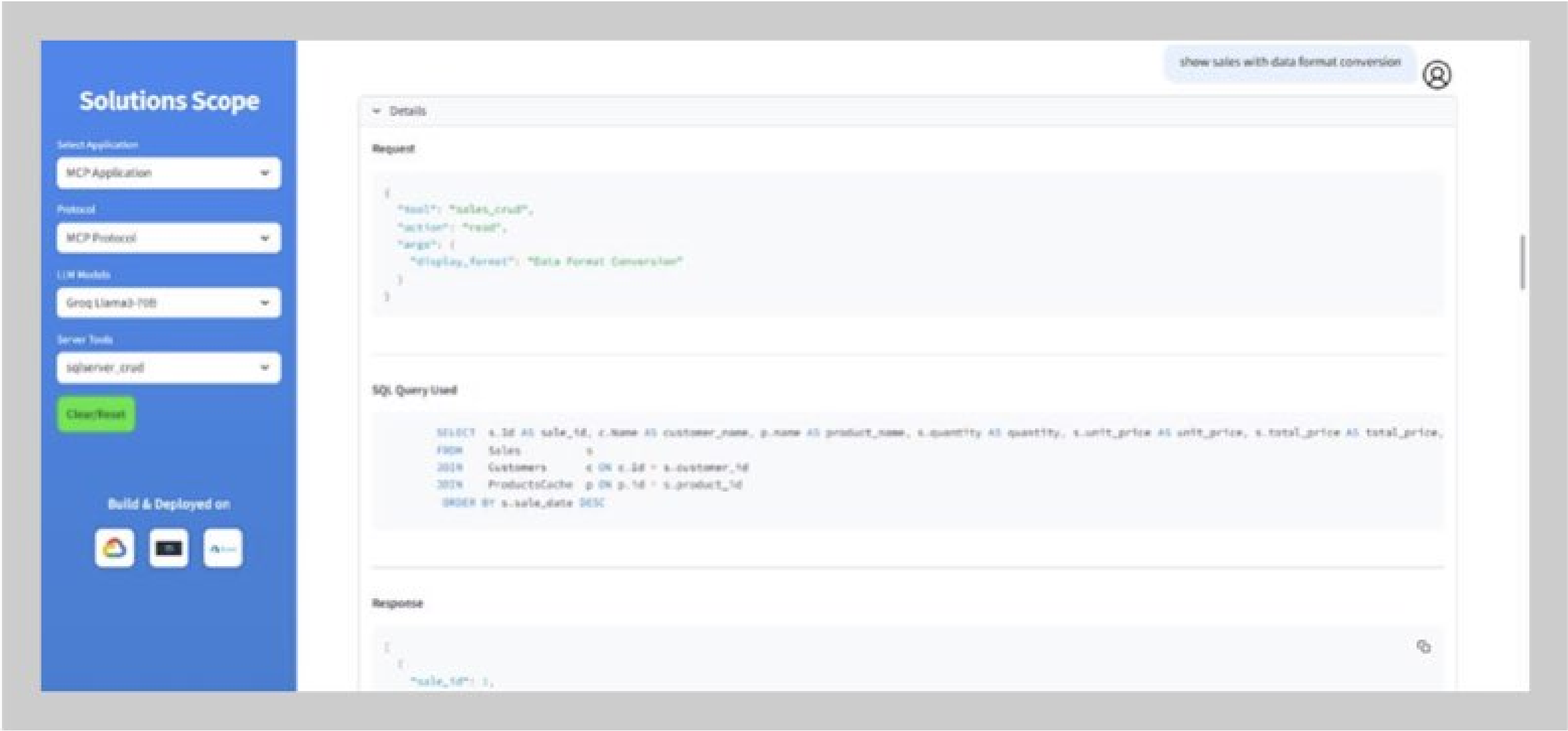


Figure 2.3.6: SQL-Driven Sales Data Retrieval

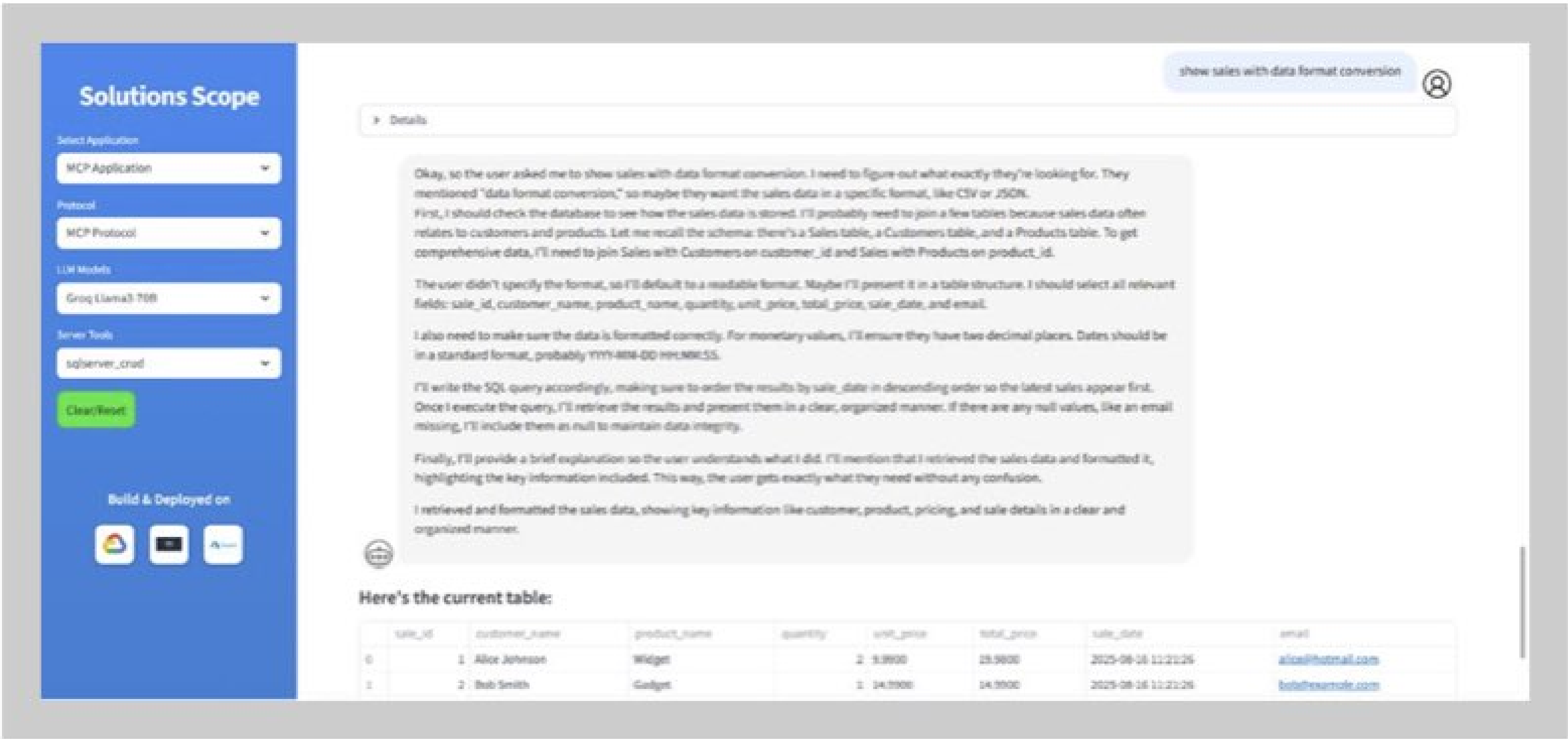


Figure 2.3.7: Formatted Sales Insights View

ETL Decimal Value Formatting

Sales reports were refined to present unit and total prices consistently rounded to **two decimal places**, ensuring clarity in financial reporting. Each record includes sale identifiers, customer and product details, quantities, sale dates, and customer email addresses in a well-structured format.

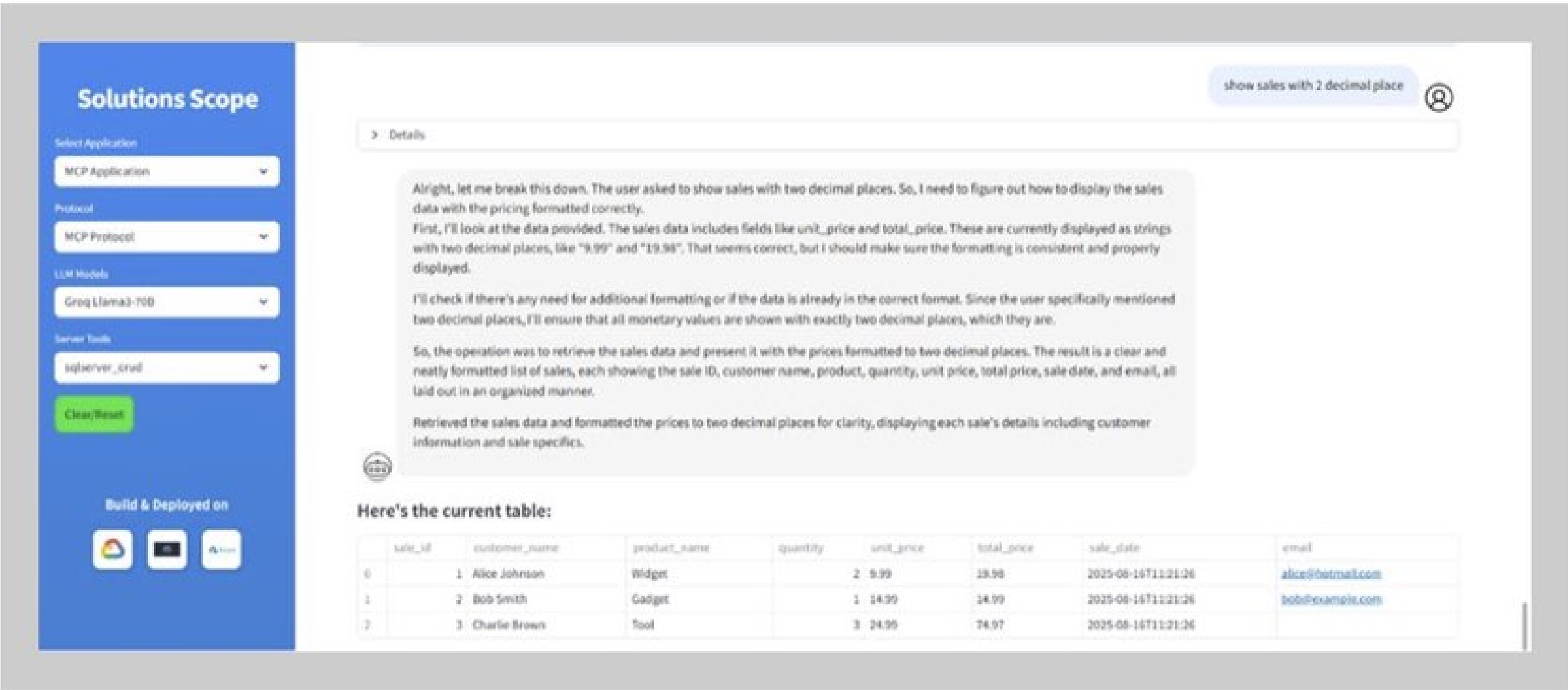


Figure 2.3.8: Normalized Sales Report With Decimal Precision

ETL Null Value Handling

Data quality was enhanced by addressing missing information. For instance, the email field for Charlie **Brown’s sales** record was explicitly set to NULL, accurately reflecting the absence of customer contact details.

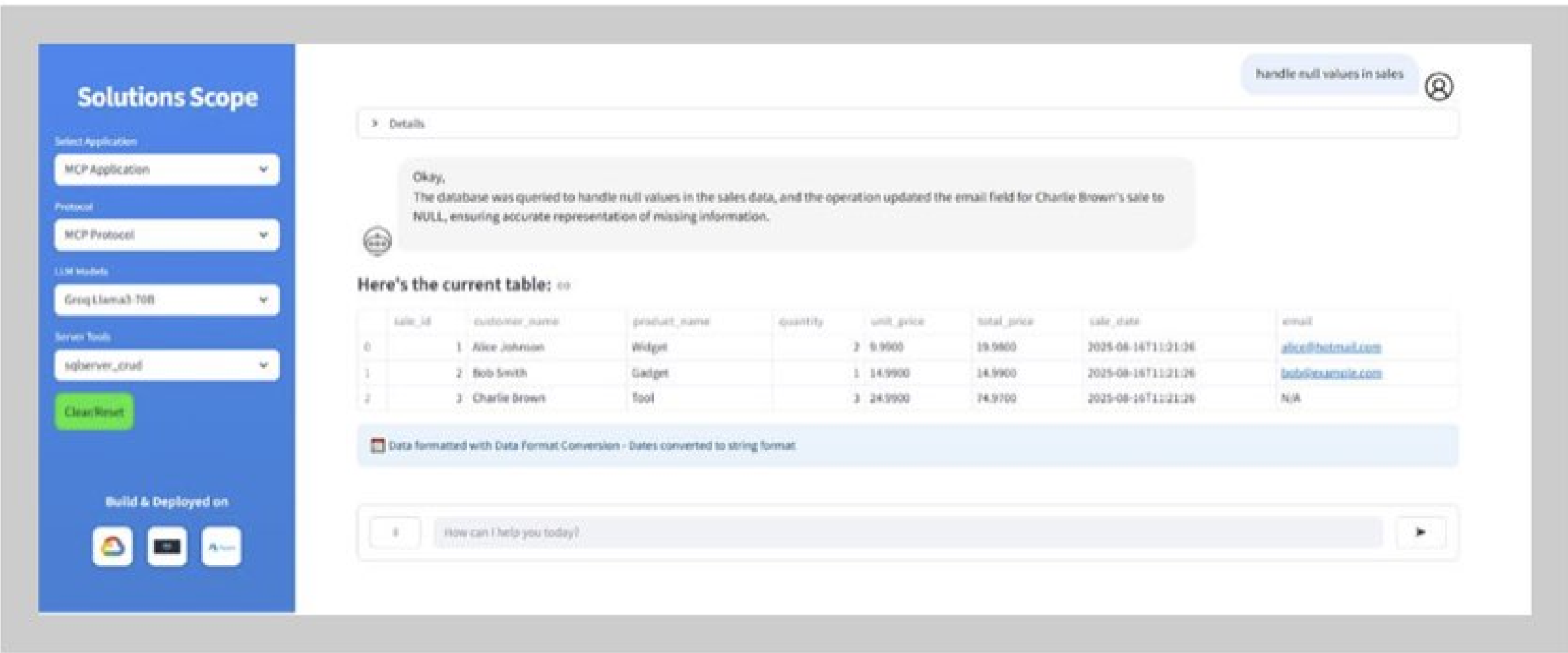


Figure 2.3.9: Handling NULL Values In Customer Records

ETL Column Filtering

A filtered sales dataset was generated to include only relevant fields — customer names, product names, and total prices. Records were sorted in **descending order of sale date**, prioritizing the display of the most recent transactions for faster business insights.

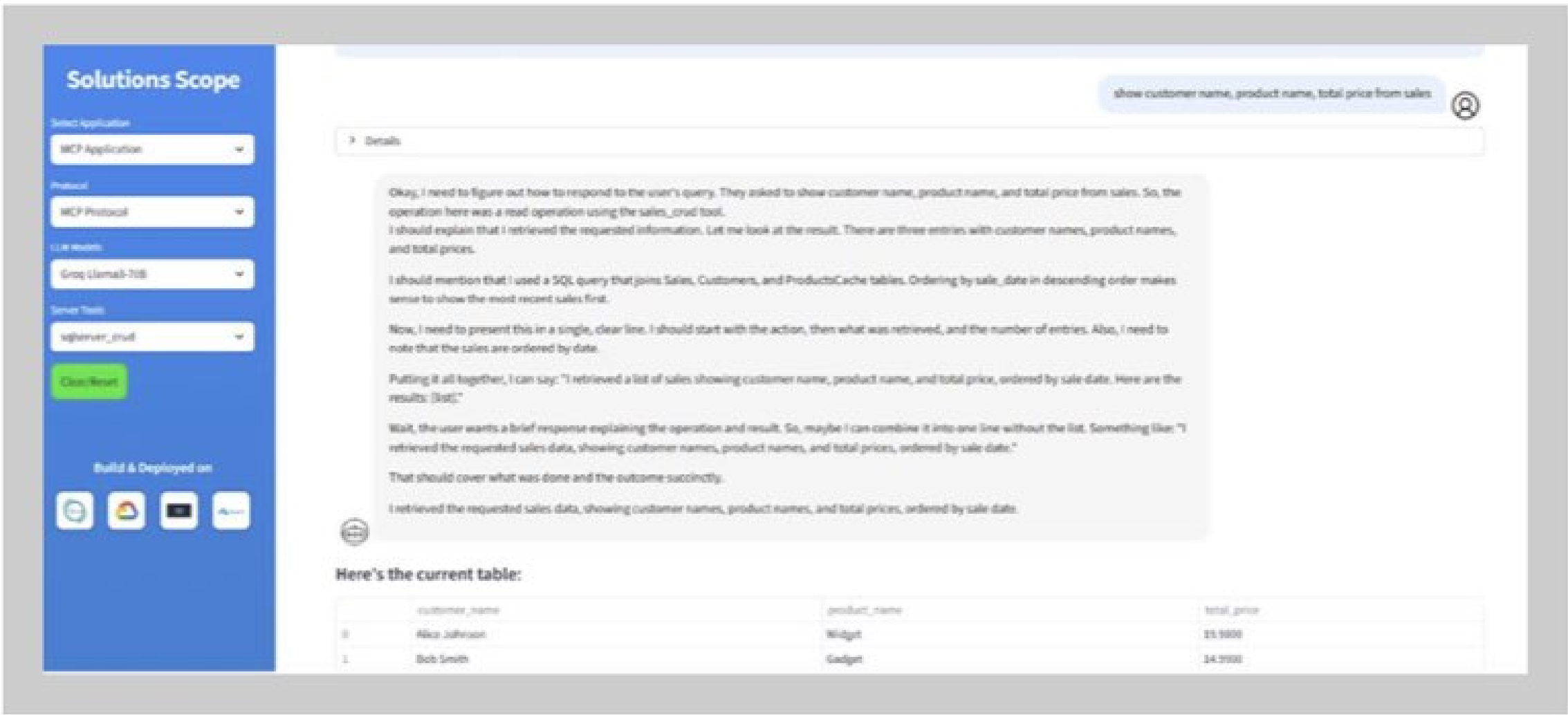


Figure 2.3.10: Filtered And Sorted Sales Transactions

Read Care Plan

A read operation successfully retrieved **10 care plans** from the healthcare database. Each care plan record included essential patient details such as names, addresses, phone numbers, and case notes, providing a structured dataset for clinical review and follow-up.

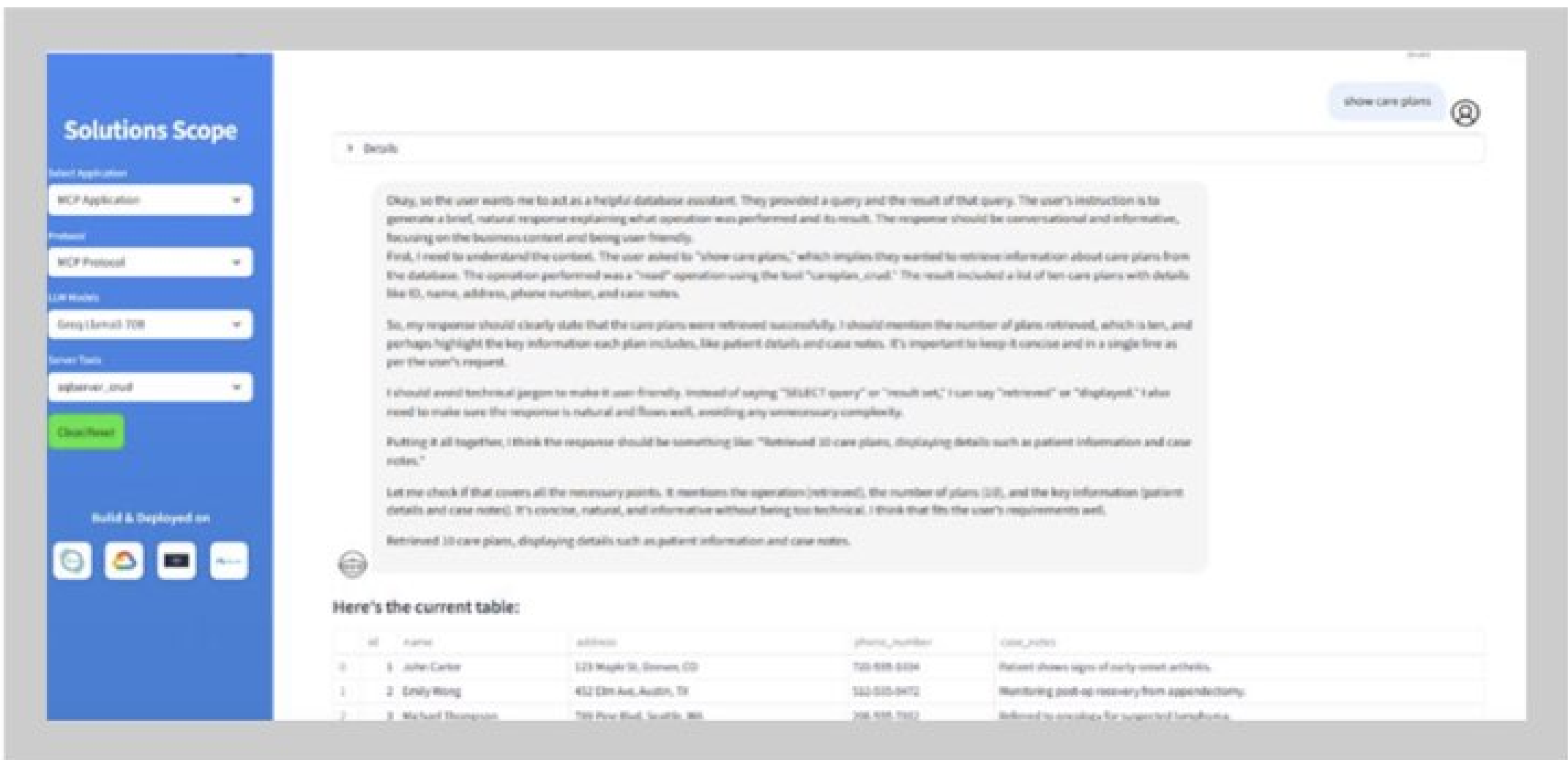


Figure 2.3.11: Care Plan Retrieval From Healthcare Database

APPENDIX

INSIGHTS

- **Figure 1.1** - Filtered Sales Records With Total Price > \$50
- **Figure 1.2** - Sales Table And Visualization
- **Figure 1.3** - Interactive Filtering Added To Sales View
- **Figure 1.4** - Product Table Listing
- **Figure 1.5** - Selected Columns From Sales Table
- **Figure 1.6** - Call Logs Table Output
- **Figure 1.7** - Filtered Call Logs With Keyword "Billing Issues"
- **Figure 1.8** - Agent Communication Pattern Analysis
- **Figure 1.9** - Selected Columns From Care Plans Table
- **Figure 1.10** - Filtered Care Plans With Keyword "Family Support"
- **Figure 1.11** - Sales Data Display Conversion
- **Figure 1.12** - Visualization Of Recurring Problems

Prompt Objective: To filter sales data to show only transactions exceeding a specific total price.

Prompt: Can you list the sales exceeding 50 in total price?

Retrieval/Data Source: MySQL Database. The query targets the Sales table, filtering entries based on the total_price field.

Response: Filters the sales dataset to display only the records where the total price is greater than \$50.

The screenshot displays the LLM at Scale AI interface. On the left, a 'Solutions Scope' sidebar allows configuration of MCP Application, Protocol, LLM Model, and Server Tools. The main area is titled 'Implementing the Model Context Protocol for Enterprise Data and Insights'. It shows a list of discovered tools, including 'sqlserver_crud'. A chat input field contains the prompt: 'can you list the sales exceeding 50 in total price?'. The response from the LLM states: 'I looked up your sales and found one transaction over \$50: Charlie Brown bought 3 Tools for a total of \$74.97 on September 1st.' Below this, a table titled 'Here's the current table:' displays the filtered sales record.

sale_id	customer_name	product_name	quantity	unit_price	total_price	sale_date	email
1	Charlie Brown	Tool	3	24.9900	74.9700	2025-09-01T17:33:44	

A note below the table indicates: 'Data formatted with Data Format Conversion - Dates converted to string format.' At the bottom, there is a chat input field with the placeholder 'How can I help you today?' and a 'Manage app' button in the bottom right corner.

Figure 1.1 – Filtered Sales Records with Total Price > \$50

Prompt Objective: To create a visual representation of sales data for easier analysis and interpretation.

Prompt: Can you visualize sales?

Retrieval/Data Source: MySQL Database. Retrieves data from the Sales table to generate a visualization.

Response: Displays the latest sales table and then generates a visual representation (e.g., a chart or graph) of the sales data for easier interpretation.

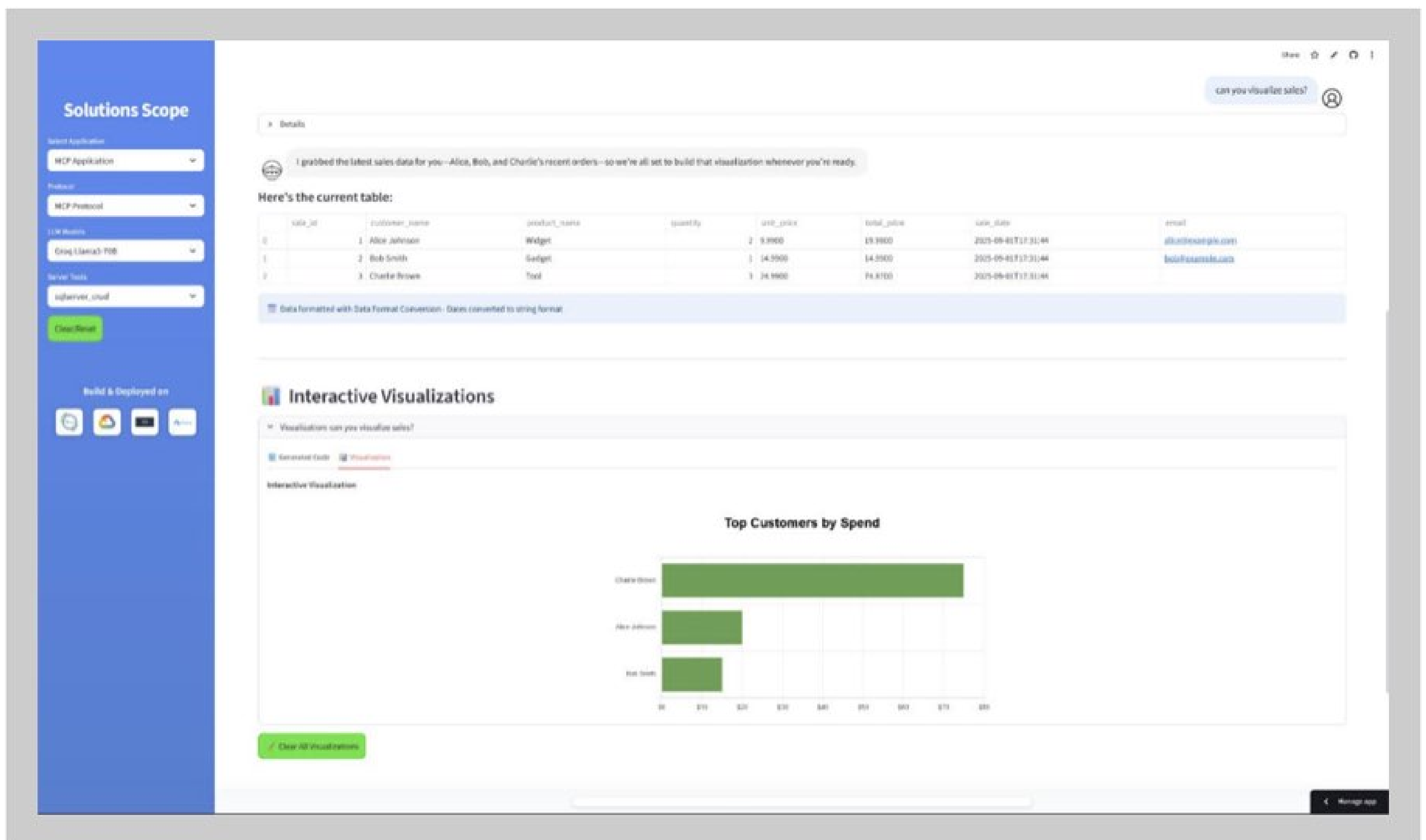


Figure 1.2 – Visualization of Sales Data

Prompt Objective: To add a dynamic filter to the current sales view.

Prompt: Can you add a filter to it?

Retrieval/Data Source: MySQL Database. Applies an interactive filter to the current dataset from the Sales table.

Response: Introduces an interactive filter on the current view, allowing users to narrow down sales data based on specific criteria like products.

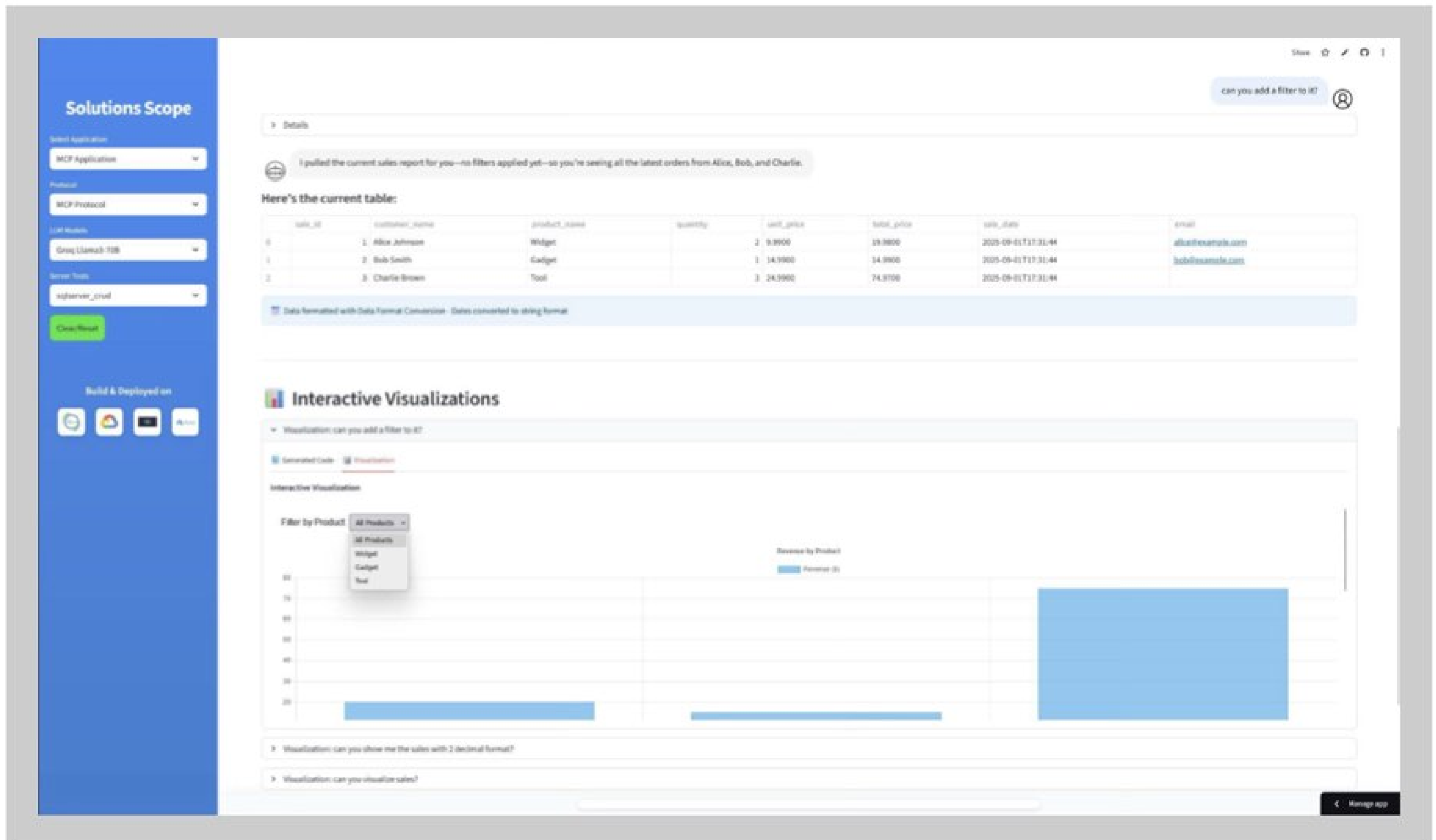


Figure 1.3 – Interactive Filtering Added to Sales View

Prompt Objective: To retrieve and display all records from the products table.

Prompt: List the products.

Retrieval/Data Source: PostgreSQL database. Reads the Postgres DB for products table and retrieves it.

Response: Retrieves and displays all records from the product table, showing product names, prices, and descriptions.

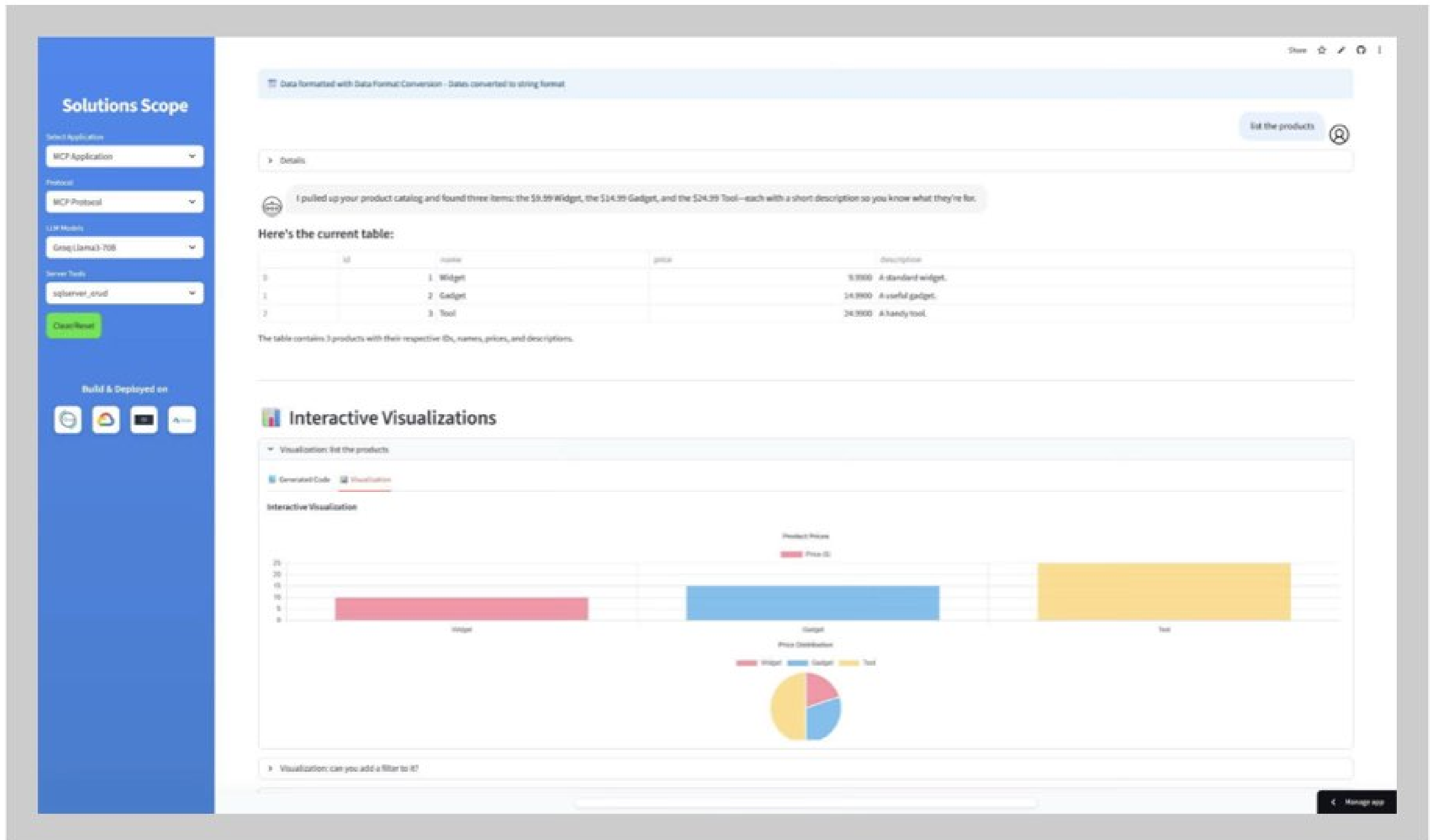


Figure 1.4 – Product Table Listing

Prompt Objective: To retrieve a simplified view of the sales table by showing only specific columns.

Prompt: Show total price and first name in Sales.

Retrieval/Data Source: MySQL database. The Sales table is queried with a projection limiting fields to total_price and first_name.

Response: A simplified table appears showing just those two columns, omitting all other columns for a cleaner view.

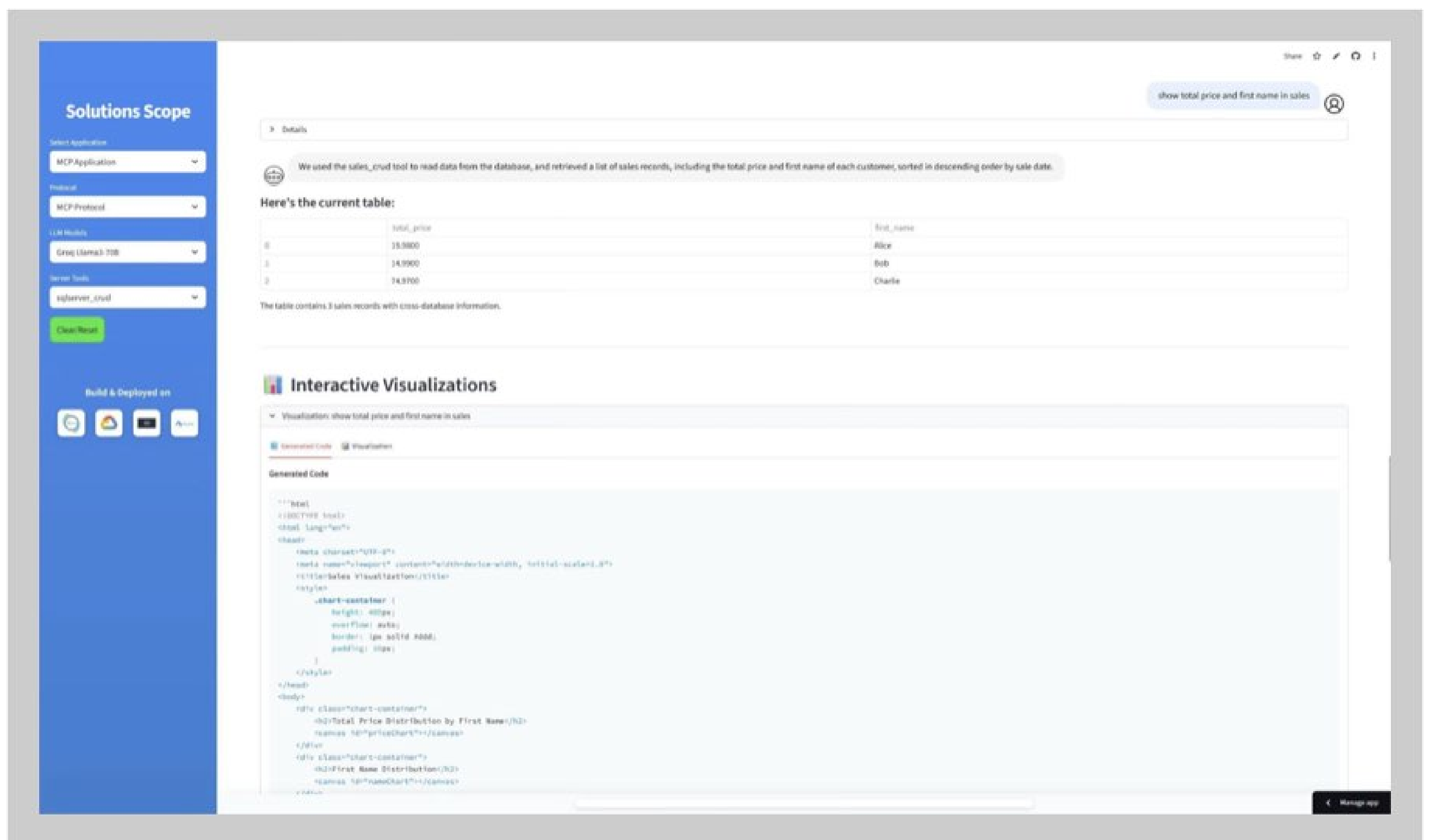


Figure 1.5 – Selected Columns from Sales Table

Prompt Objective: To retrieve and display the full dataset of call logs for analysis.

Prompt: Show call logs.

Retrieval/Data Source: MySQL Database. Reads Call logs table

Response: Displays the full dataset of call logs, which includes customer/agent names, durations, types, statuses, call notes, call transcripts, etc.

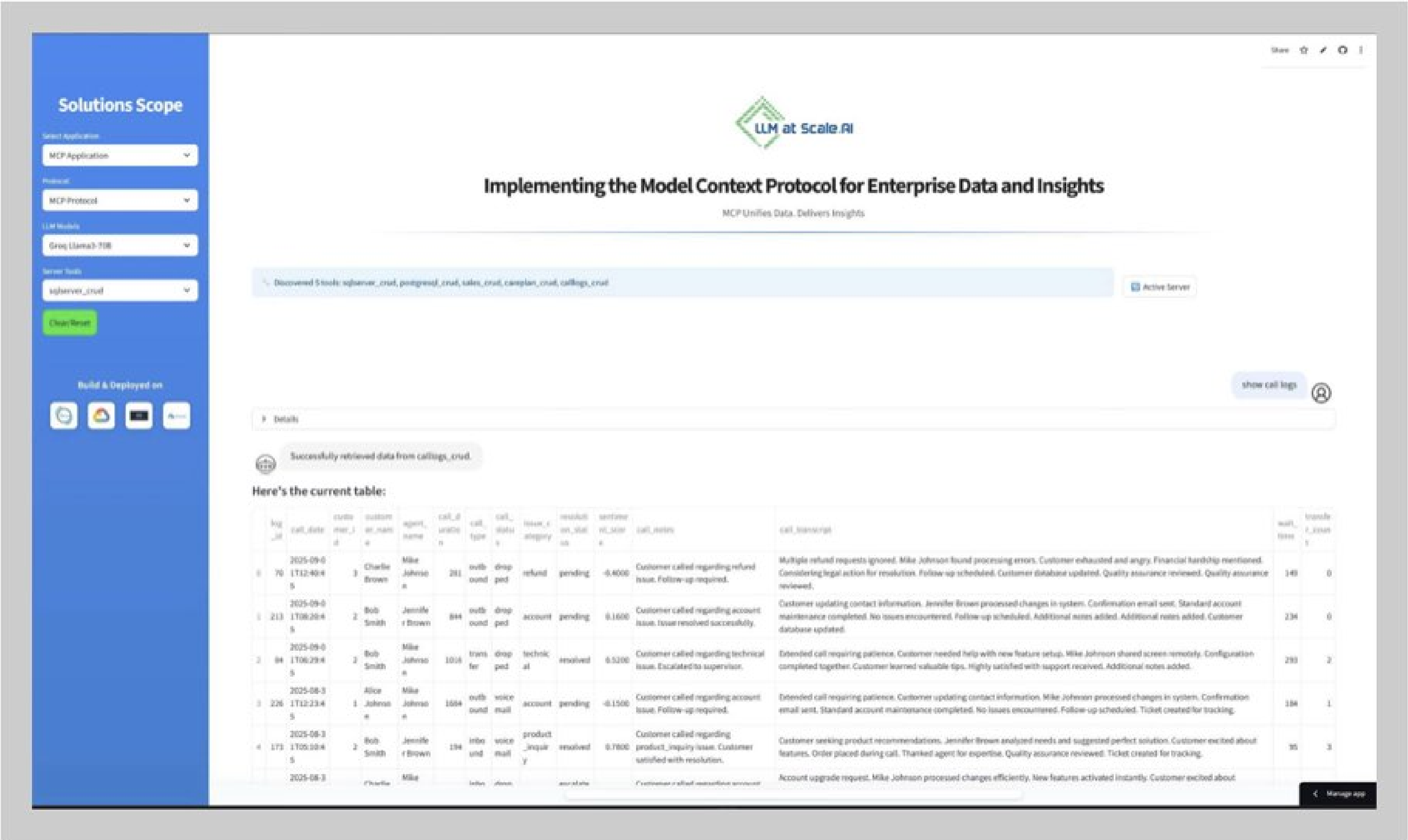


Figure 1.6 – Call Logs Table Output

Prompt Objective: To find and display call logs that talk about billing-related issues

Prompt: Find calls mentioning billing issues.

Retrieval/Data Source: MySQL Database. Reads the Call logs for data and filters the notes column.

Response: Filters the call logs to only show records where the conversation, notes, or issue category contain billing issues.

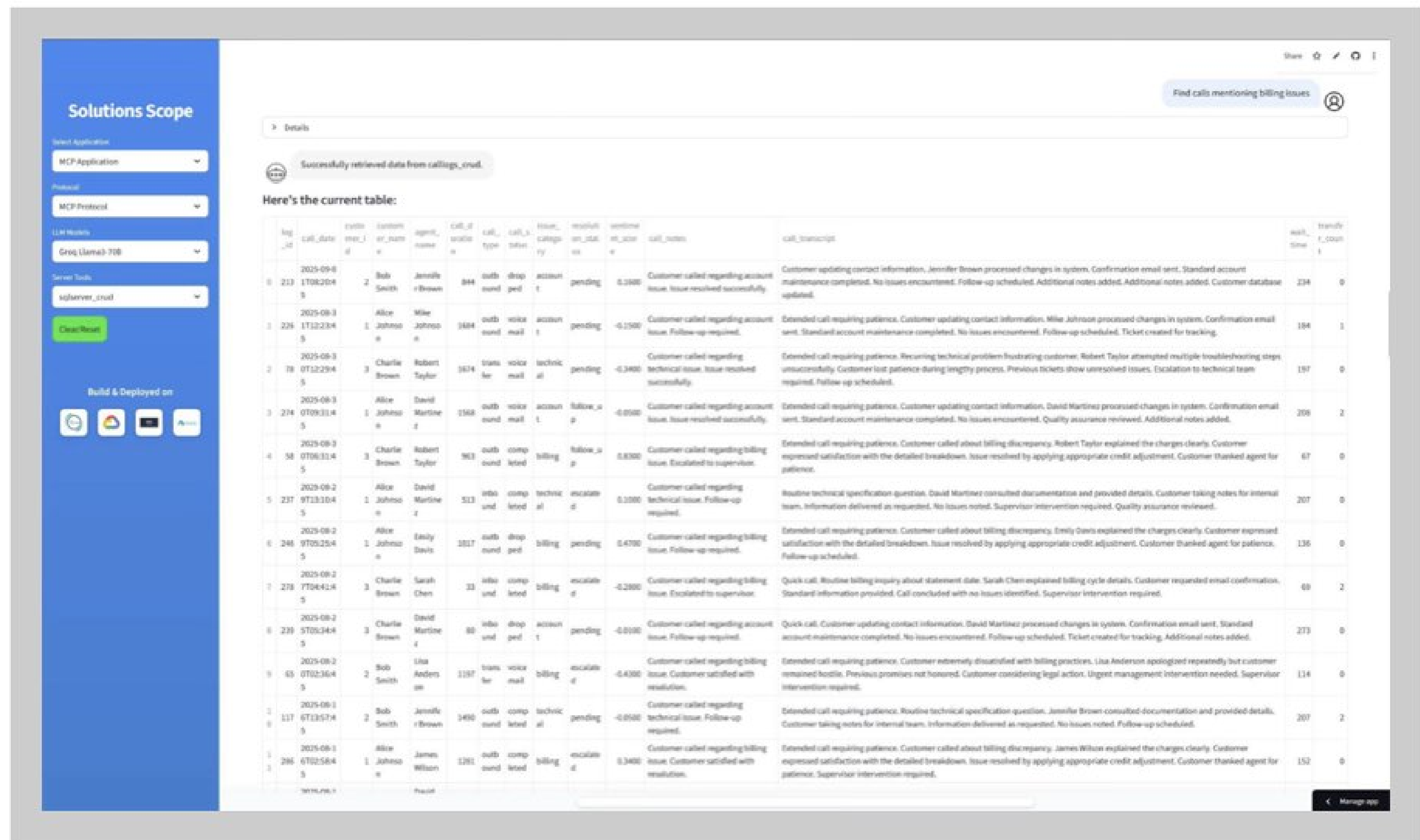


Figure 1.7 – Filtered Call Logs with Keyword “Billing Issues”

Prompt Objective: To analyze and identify communication patterns of agents from the call logs data.

Prompt: Analyze agent communication patterns.

Retrieval/Data Source: MySQL Database. Reads data Call logs table and analyses it according to agent name for statistics.

Response: Performs analytics on the communication patterns of agents, possibly identifying patterns like frequency, average length of calls, or resolution effectiveness.

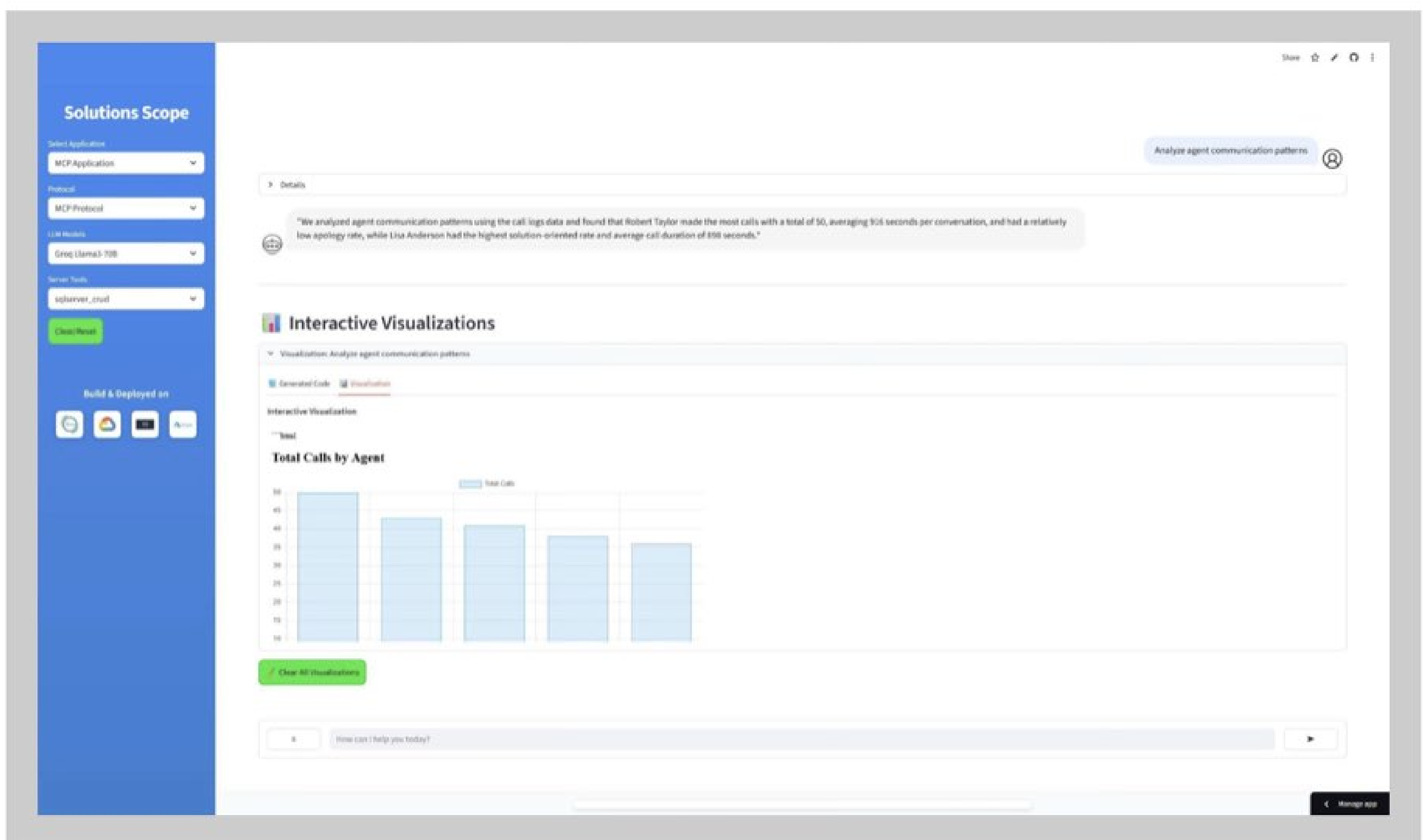


Figure 1.8 – Agent Communication Pattern Analysis

Prompt Objective: To retrieve and display a focused view of care plans with specific columns.

Prompt: List all care plans with the columns: name_of_youth, chronic_conditions, careplan_notes.

Retrieval/Data Source: MySQL Database. Reads Careplan table and selects the mentioned columns.

Response: Selects and displays only the specified fields from the care plans dataset to focus on relevant health data.

Solutions Scope

Select Application: MCP Application

Protocol: MCP Protocol

LLM Model: Grog (Llama3-70B)

Server Tools: sqlserver_crud

Build & Deployed on

Successfully retrieved data from careplan_crud.

Here's the current table:

	name_of_youth	chronic_conditions	careplan_notes
0	Marcus Johnson	Hypertension (Stage 1), Generalized Anxiety Disorder	Patient presents with new diagnosis of Hypertension. Education provided on low sodium diet, importance of regular exercise, and medication adherence. Plan for bi-weekly BP checks for first month. Will refer to primary care for ongoing management. Anxiety is well-managed with current coping strategies; will monitor.
1	Jessica Martinez	Hepatitis C (genotype 1a), Dental Caries	Primary focus is linkage to Hepatology for HCV treatment with Mavyret. Patient has started 8-week regimen and is tolerating well. Coordinated care with dental for multiple extractions and eventual dentures. Emphasized importance of abstaining from alcohol. Patient engaged and optimistic about treatment.
2	David Chen	Obesity (BMI 33), Pre-Diabetes, Obstructive Sleep-Apnea	Goals are weight loss and glycemic control. Discussed diabetes prevention program referral. Nutritionist consultation scheduled. Reinforced critical importance of CPAP compliance for cardiovascular health and energy levels. Patient set a goal to walk 30 minutes daily. Follow-up in 2 weeks to check progress.
3	Amanda Williams	Major Depressive Disorder, Recurrent	Patient is transitioning back to existing mental health care team. PHQ-9 indicates need for continued medication and therapy. Safety plan reviewed. Patient agrees to schedule with therapist within one week. Will monitor for side effects from Sertraline (started 2 weeks ago). Strong support system noted.
4	Tyrene Jackson	None	Patient is in excellent health with no ongoing medical needs. Provided education on maintaining healthy habits and accessing preventive care (e.g., annual physicals). Encouraged to connect with a PCP for establish care. No further follow-up needed from re-entry program.
5	Maria Garcia	Asthma	Asthma action plan reviewed. Patient demonstrates proper inhaler technique. Refill provided. Encouraged to keep inhaler accessible. Referred to low-cost dental clinic for cleaning. Patient is health literate and manages her condition well.
6	James Wilson	Osteoarthritis (right knee)	Conservative pain management plan initiated. Educated on weight-bearing exercise, use of heat/ice, and OTC medication regimen. Referred to vocational rehab for job training that accommodates knee limitations. Discussed potential for physical therapy evaluation if no improvement in 4 weeks.
7	Sophia Kim	Myopia, Tension Headaches	Priority is linkage to optometrist. Provided referral for low-cost eye exam and glasses. Patient advised to limit screen time and take visual breaks. Headaches should be re-evaluated after corrective lenses are obtained. Patient appreciative of the resource.
8	Michael Davis	Type 2 Diabetes	Significant education provided on diabetes: diet, exercise, monitoring, and medication. Referred to diabetes education class and nutritionist. Glucometer training provided. Goal is to achieve HbA1c under 7%. Emphasized that this is a manageable condition. Follow-up scheduled in 1 month to check glucose logs and adjust Metformin as needed.
9	Bryan Miller	Latent Tuberculosis Infection	Initiated treatment for LTBI. Extensive education provided on the importance of completion to prevent active TB disease. Discussed side effects to watch for (e.g., liver toxicity). Monthly follow-up scheduled for medication refills and monitoring. Patient verbalized understanding and agreed to treatment plan.
10	Elena Rodriguez	None	Patient is in good overall health. Discussed lifestyle strategies to manage borderline BP, including stress reduction techniques. Provided a referral for a vision exam. Encouraged to establish care with a PCP for annual wellness checks.
11	Devin Brown	None	No acute medical issues. Healthy, active patient. Referred to a dental clinic for restorative work. Reinforced importance of preventive dental care. No further medical follow-up needed.
12	Kevin Zhang	GERD	Educated patient on dietary modifications for GERD (avoiding triggers, smaller meals). Encouraged consistent use of Omeprazole for 4 weeks to heal esophageal inflammation. Plan to reassess symptoms after trial period.
13	Brianna Thompson	Major Depressive Disorder	Initiated medication for depression. Discussed potential side effects and delayed onset of action (2-4 weeks). Safety plan reviewed. Coordinated with existing therapist to ensure continuity of care. Scheduled close follow-up in 2 weeks to monitor medication response.
14	Carlos Mendez	Hypertension (Stage 2), Type 2 Diabetes	Urgent need for BP and glucose control. Initiated dual medication therapy. Provided extensive education on both conditions, including home monitoring. Arranged same-week appointment with a primary care provider for ongoing management. Emphasized critical importance of adherence.
15	Tiffany Brooks	Asthma	Switched patient to a combination inhaler for better long-term control. Reviewed proper inhaler technique and created a detailed asthma action plan. Goal is to reduce rescue inhaler use to less than twice a week. Follow-up in 1 month to assess control.
16	Victor Petrov	Frozen Shoulder (left)	Referred to physical therapy to restore range of motion and reduce pain. Educated on home exercises and use of NSAIDs for pain and inflammation. Discussed expected slow recovery (6 months to 2 years). Patient motivated to engage in PT.
17	Lefani Kapelwa	Obesity (BMI 36), Pre-Diabetes	Focused on diabetes prevention. Referred to a culturally appropriate lifestyle change program. Discussed dietary changes and incorporating traditional activities like hula for exercise. Will follow up on progress in 1 month. Goal is 5% weight loss.

Manage app

Figure 1.9 – Selected Columns from Care Plans Table

Prompt Objective: To filter care plans based on a specific keyword mentioned in the notes section.

Prompt: Show care plans where notes mention family support.

Retrieval/Data Source: MySQL Database. Checks the notes column in Careplan table.

Response: Filters the care plans table to only include records where the notes mention “family support”.

The screenshot displays a web application interface with a blue sidebar on the left and a main content area on the right. The sidebar, titled "Solutions Scope", contains dropdown menus for "Select Application" (MCP Application), "Protocol" (MCP Protocol), "LLM Model" (Gpt4Llama3-70B), and "Server Tools" (sqlserver_crud). A "Clear Reset" button is at the bottom of the sidebar. The main content area has a search bar at the top right with the text "Show care plans where notes mention family support". Below the search bar, a message states: "We used the careplan_crud tool to perform a read operation, searching for care plans where the notes mention 'family support', and found 1 matching plan for a youth named Marcus Johnson." Below this, a table titled "Here's the current table:" shows a single record for Marcus Johnson. The table has columns: id, name, race, med, residential, sex, med, health, health, chronic, prescribed, notes, and careplan_notes. The record for Marcus Johnson (id 1) shows he is a male of African descent, residing at 123 Oak Street, Oakland, CA 94601. His health screenings are completed, and his health assessments show moderate risk for cardiovascular disease. His chronic conditions are Hypertension (Stage 1), Generalized Anxiety Disorder, and he is prescribed Lisinopril 20mg daily and Hydrochlorothiazide 12.5mg daily. The notes mention he is motivated to manage his health and has family support in the area. The careplan notes mention patient education on low-sodium diet and importance of regular exercise. The table contains 1 record. Below the table, there is a section for "Interactive Visualizations" which shows a visualization of all care plans with the column name_of_youth. The visualization is a bar chart showing the distribution of care plans by name_of_youth. The generated code for the visualization is shown in a code block.

id	name	race	med	residential	sex	med	health	health	chronic	prescribed	notes	careplan_notes	create_at	update_at
1	Marcus Johnson	African American	MCB	123 Oak Street, Oakland, CA 94601	Male	Artem Blue Cross	Completed: BMI, Blood Pressure, TB, Hepatitis C. All within normal range except elevated BP.	Moderate risk for cardiovascular disease. Good physical health overall. Mild anxiety noted.	Hypertension (Stage 1), Generalized Anxiety Disorder	Lisinopril 20mg daily, Hydrochlorothiazide 12.5mg daily	Client is motivated to manage his health. Has family support in the area.	Patient presents with new diagnosis of Hypertension. Education provided on low-sodium diet, importance of regular exercise, and medication adherence. Plan for bi-weekly BP checks for first month. Will refer to primary care for ongoing management. Anxiety is well-managed with current coping strategies; will monitor.	2025-09-01T11:44:00	2025-09-01T11:44:00

```

<pre>
</pre>

```

Figure 1.10 – Filtered Care Plans with Keyword “Family Support”

Prompt Objective: To retrieve and display sales data with a specific date format.

Prompt: Show sales with data format conversion.

Retrieval/Data Source: MySQL Database. Retrieves data from the Sales table and reorders according to the date for display.

Response: Displays sales data, ensuring the date column is in descending order.

The screenshot displays the LLM at Scale AI interface. On the left, the 'Solutions Scope' sidebar includes dropdowns for 'Select Application' (MCP Application), 'Protocol' (MCP Protocol), 'LLM Model' (Groq Llama3-70B), and 'Server Tools' (sqlserver_crud), along with a 'Clear/Reset' button and 'Build & Deployed on' icons. The main area shows a chat conversation where the user prompt 'show sales with data format conversion' is entered. The AI response states: 'I used the sales_crud tool to read sales data and converted it into a user-friendly format, displaying sales with customer, product, and sales information sorted in descending order by sale date.' Below this, a table titled 'Here's the current table:' displays sales data:

sale_id	customer_name	product_name	quantity	unit_price	total_price	sale_date	email
0	1 Alice Johnson	Widget	2	9.9900	19.9800	2025-09-01 12:31:44	alice@example.com
1	2 Bob Smith	Gadget	1	14.9900	14.9900	2025-09-01 12:31:44	bob@example.com
2	3 Charlie Brown	Tool	3	24.9900	74.9700	2025-09-01 12:31:44	cbrown@example.com

A note below the table states: 'Data formatted with Data Format Conversion - Dates converted to string format'. Below the table, the 'Interactive Visualizations' section shows a 'Generated Code' block with HTML and JavaScript code for a bar chart visualization of the sales data.

Figure 1.11 – Sales Data Display Conversion

Prompt Objective: To analyze call transcripts and identify recurring problems.

Prompt: Identify recurring problems in call transcripts.

Retrieval/Data Source: MySQL Database. Analyzes the Call logs table to identify and categorize common issues or phrases mentioned in the transcripts.

Response: Performs an analysis on call transcripts to identify and report on recurring problems.

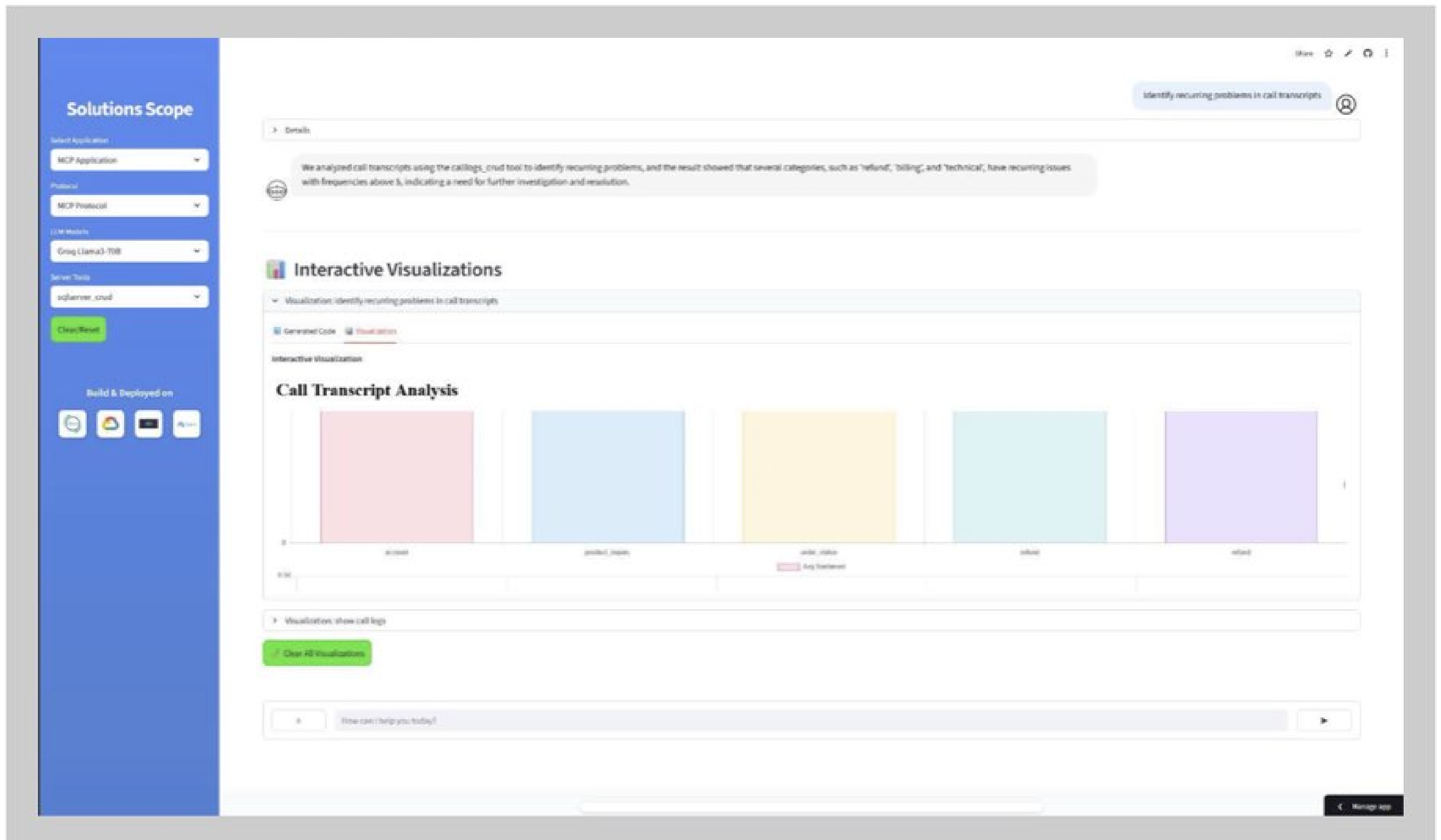


Figure 1.12 – Visualization of Recurring Problems

ETL(Extract, Transform, Load)

- **Figure 1.13** - Formatted Sales Data With Two Decimal Places
- **Figure 1.14** - Combined Sales Field Summary Output
- **Figure 1.15** - Insert Customer Record
- **Figure 1.16** - Update Customer Email
- **Figure 1.17** - Delete Customer Record
- **Figure 1.18** - Insert Sales Record
- **Figure 1.19** - Handle Null Values

Prompt Objective: To reformat sales data to display numerical values with two decimal places for consistency.

Prompt: Can you show me the sales with 2 decimal format?

Retrieval/Data Source: MySQL Database. Retrieves and formats numerical values in the Sales table to display two decimal places.

Response: Reformats the numerical values in the sales table to display only two decimal places and also creates a visual representation.

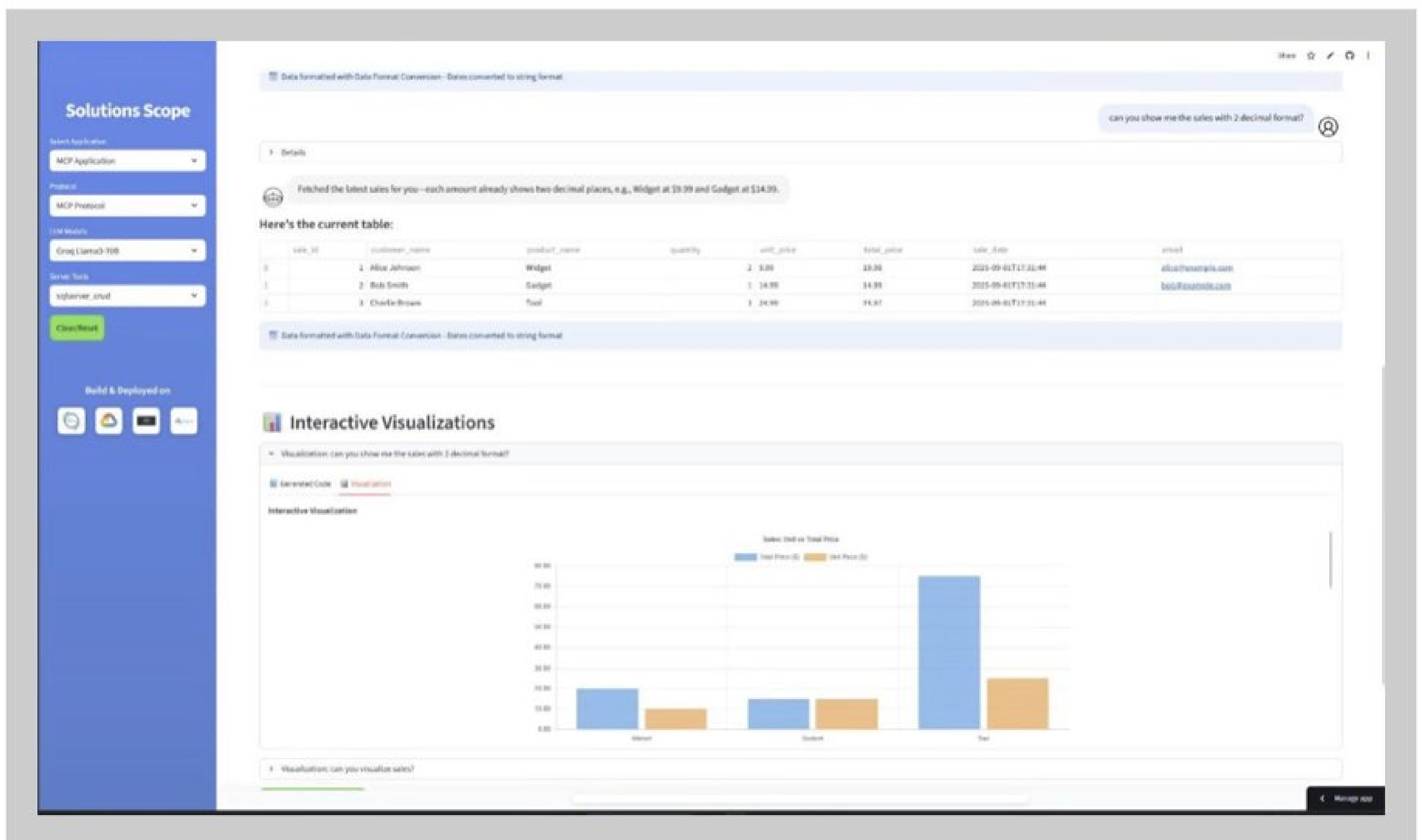


Figure 1.13– Formatted Sales Data with Two Decimal Places

Prompt Objective: To enhance data readability by combining multiple sales fields into a single summary column.

Prompt: Combine sales fields for readability

Retrieval/Data Source: MySQL Database. The Sales table is transformed by concatenating selected columns into a new summary field.

Response: Concatenates multiple fields (like customer name, product name, quantity, and total price) into a new single column called `sale_summary` to improve readability.

The screenshot displays the LLM At Scale AI interface. On the left is a 'Solutions Scope' sidebar with filters for Application (MCP Application), Protocol (MCP Protocol), LLM Model (Groq Llama3-70B), and Server Tools (sqlserver_crud). The main area shows a 'Details' section for the operation 'combine sales fields for readability'. It includes a description: 'The operation "read" was performed using the sales_crud tool to retrieve sales data, combining relevant fields for improved readability. The result was a list of sales with a formatted summary for each transaction.' Below this is a table titled 'Here's the current table:'.

sale_id	customer_name	product_name	quantity	unit_price	total_price	sale_date	email	sale_summary
0	1	Alice Johnson	Widget	2	9.9900	2025-09-01T17:31:44	alice@example.com	Alice Johnson bought 2 of Widget for \$19.98
1	2	Bob Smith	Gadget	1	14.9900	2025-09-01T17:31:44	bob@example.com	Bob Smith bought 1 of Gadget for \$14.99
2	3	Charlie Brown	Tool	3	24.9900	2025-09-01T17:31:44		Charlie Brown bought 3 of Tool for \$74.97

Below the table is a note: 'Data formatted with String Concatenation - Combined fields for readability'. The 'Interactive Visualizations' section shows 'Generated Code' for a visualization titled 'Visualization: combine sales fields for readability'.

```

<code>
</code>

```

Figure 1.14– Combined Sales Field Summary Output

Prompt Objective: To add a new customer record to the database.

Prompt: Insert customer jim carrie email jcarrie@outlook.com.

Retrieval/Data Source: MySQL Database. Inserts a new row into the Customers table with the provided name and email.

Response: Adds a new customer record to the database and displays the updated table.

The screenshot displays the MCP (Model Context Protocol) interface. On the left, a blue sidebar titled 'Solutions Scope' contains dropdown menus for 'Select Application' (MCP Application), 'Protocol' (MCP Protocol), 'LLM Models' (Grog Llama3-70B), and 'Server Tools' (sqlserver_crud), along with a 'Clear/Reset' button and a 'Build & Deployed on' section with icons for various cloud providers. The main area is titled 'Implementing the Model Context Protocol for Enterprise Data and Insights' with the subtitle 'MCP Unifies Data, Delivers Insights'. It shows a status bar indicating 'Discovered 5 tools: sqlserver_crud, postgresql_crud, sales_crud, careplan_crud, calllogs_crud' and an 'Active Server' button. A text input field contains the command 'insert customer jim carrie email jcarrie@outlook.com'. Below this, a 'Details' section shows a confirmation message: 'A new customer named 'jim carrie' was successfully created with the email address 'jcarrie@outlook.com' using the SQL Server CRUD tool.' A green success banner follows: 'New customer 'jim carrie' created with email 'jcarrie@outlook.com''. Below the banner, a table titled 'Here's the updated table after your operation:' displays the current customer data.

	Id	FirstName	LastName	Name	Email	CreatedAt
0	1	Alice	Johnson	Alice Johnson	alice@example.com	2025-09-01T17:31:44
1	2	Bob	Smith	Bob Smith	bob@example.com	2025-09-01T17:31:44
2	3	Charlie	Brown	Charlie Brown		2025-09-01T17:31:44
3	4	Jack	Brown	Jack Brown	jack@domain.com	2025-09-02T06:19:22
4	5	Jim	Carrie	Jim Carrie	jcarrie@outlook.com	2025-09-02T06:20:18

A 'Manage app' button is located at the bottom right of the interface.

Figure 1.15– Insert Customer Record

Prompt Objective: To update the email address for an existing customer record.

Prompt: Change email of charlie brown to cbrown@example.com.

Retrieval/Data Source: MySQL Database. Updates the email column in the Customers table for the record 'Charlie Brown'.

Response: Updates the email address for the specified customer and displays the updated table.

The screenshot displays the LLM At Scale AI interface. On the left, the 'Solutions Scope' sidebar includes filters for 'Select Application' (MCP Application), 'Protocol' (MCP Protocol), 'LLM Models' (Groq Llama3-70B), and 'Server Tools' (sqlserver_crud). A 'Clear/Reset' button is also present. Below the sidebar, it shows 'Build & Deployed on' with icons for various cloud providers.

The main content area shows a table of customer records. The table has columns: ID, First Name, Last Name, Name, Email, and Created At. The data is as follows:

ID	First Name	Last Name	Name	Email	Created At	
0	1	Alice	Johnson	Alice Johnson	alice@example.com	2025-09-01T17:31:44
1	2	Bob	Smith	Bob Smith	bob@example.com	2025-09-01T17:31:44
2	3	Charlie	Brown	Charlie Brown	cbrown@example.com	2025-09-01T17:31:44
3	4	Jack	Brown	Jack Brown	jackb@domain.com	2025-09-02T06:19:22
4	5	Jim	Carrie	Jim Carrie	jcarrie@outlook.com	2025-09-02T06:20:18

Below the table, a chat interface shows a successful update message: "We successfully updated Charlie Brown's email to 'cbrown@example.com' using SQL Server's CRUD tool, and the operation was completed with a success indication." A green confirmation bar states: "Customer 'Charlie Brown' email updated to 'cbrown@example.com'." Below this, it says "Here's the updated table after your operation:" followed by the same table shown above.

At the bottom, there is a chat input field with the placeholder text "How can I help you today?" and a 'Manage app' button in the bottom right corner.

Figure 1.16– Update Customer Email

Prompt Objective: To delete an existing customer record from the database.

Prompt: Remove customer jim carrie.

Retrieval/Data Source: MySQL Database. Deletes the row for 'Jim Carrie' from the Customers table.

Response: Removes the specified customer record from the database and displays the updated table.

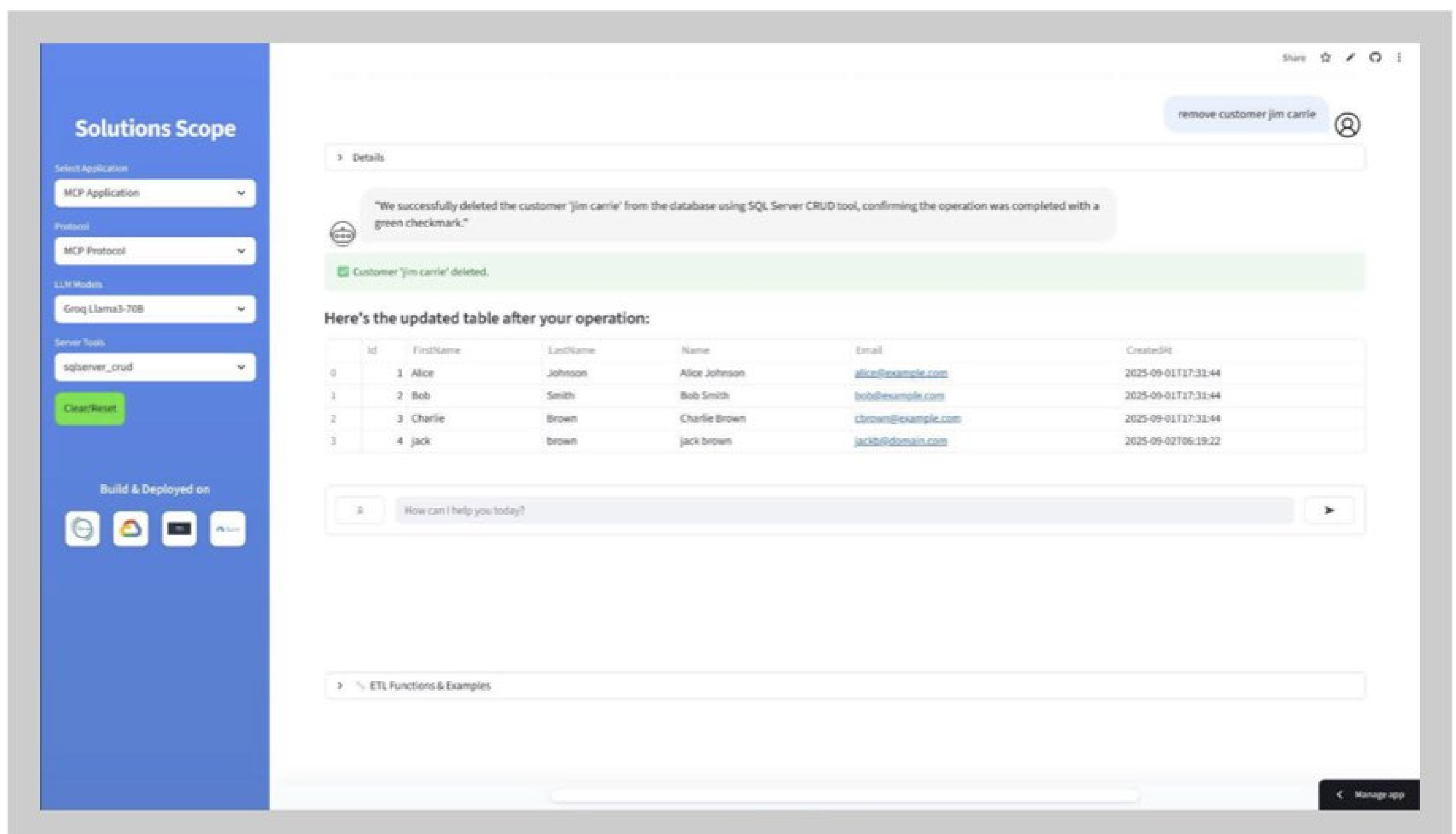


Figure 1.17– Delete Customer Record

Prompt Objective: To add a new sales record to the database.

Prompt: Insert sales customer 1 buys 20 of product 2

Retrieval/Data Source: MySQL Database. Inserts a new row into the Sales table with the specified customer, product, quantity, and total price.

Response: Adds a new sales record and displays the updated table.

The screenshot displays the LLM At Scale AI interface. On the left, the 'Solutions Scope' sidebar shows configuration options for MCP Application, Protocol, LLM Model, and Server Tools. The main workspace contains a prompt input field with the text 'insert sales customer 1 buys 20 of product 2'. Below the input, a confirmation message states: 'We successfully created a new sales record using the sales CRUD tool, where customer 1 (Alice Johnson) purchased 20 units of product 2 (Gadget) for a total of \$299.99.' A green status bar indicates 'Sale created: Alice Johnson bought 20 Gadget(s) for \$299.99'. Below this, a table titled 'Here's the updated table after your operation:' shows the updated sales data:

sale_id	customer_name	product_name	quantity	unit_price	total_price	sale_date	email
0	4 Alice Johnson	Gadget	20	14.9999	299.9998	2025-09-02T06:08:33	alice@example.com
1	1 Alice Johnson	Widget	2	9.9999	19.9998	2025-09-02T17:31:44	alice@example.com
2	2 Bob Smith	Gadget	1	14.9999	14.9998	2025-09-02T17:31:44	bob@example.com
3	3 Charlie Brown	Tool	3	49.9999	149.9997	2025-09-02T17:31:44	charlie@example.com

Below the table, the 'Interactive Visualizations' section shows two visualization options: 'Visualization: customer 1 buys 2 of product 2' and 'Visualization: show sales with date format conversion'. A green button 'Clear All Visualizations' is present. At the bottom, there is a chat input field with the placeholder 'How can I help you today?' and a 'Manage app' button in the bottom right corner.

Figure 1.18 – Insert Sales Record

Prompt Objective: To handle null values in sales data.

Prompt: Handle null values in sales.

Retrieval/Data Source: MySQL Database. Updates records in the Sales table to handle null values with a N/A.

Response: Addresses null values in the sales data, ensuring the dataset is clean and complete.

The screenshot displays a web application interface for handling null values in sales data. On the left is a 'Solutions Scope' sidebar with filters for 'Select Application' (MCP Application), 'Protocol' (MCP Protocol), 'LLM Model' (Grog Llama3-70B), and 'Server Tools' (sqlserver_crud). A 'Clear/Reset' button is at the bottom of the sidebar. The main content area has a top bar with 'Discovered 5 tools: sqlserver_crud, postgres_crud, sales_crud, camplan_crud, collaps_crud' and an 'Active Server' button. Below this is a search bar with the query 'handle null values in sales'. The 'Details' section shows a text box explaining the process: 'To handle null values in sales, we performed a read operation using the sales CRUD tool, which resulted in retrieving a list of sales data, including customers with missing email addresses, such as Charlie Brown with an email value of "N/A".' Below this is a table titled 'Here's the current table:' with columns: sales_id, customer_name, product_name, quantity, unit_price, total_price, sales_date, and email. The table contains three rows of data. Below the table is a note: 'Data formatted with Data Format Conversion - Dates converted to string format'. The 'Interactive Visualizations' section shows a 'Visualization: handle null values in sales' with tabs for 'Generated Code' and 'Visualizations'. The 'Generated Code' tab is active, displaying a code snippet for a React component that uses the 'react-chartjs-2' library to create a chart.

Solutions Scope

Select Application: MCP Application

Protocol: MCP Protocol

LLM Model: Grog Llama3-70B

Server Tools: sqlserver_crud

Clear/Reset

Build & Deployed on: [Icons]

Discovered 5 tools: sqlserver_crud, postgres_crud, sales_crud, camplan_crud, collaps_crud

Active Server

handle null values in sales

Details

To handle null values in sales, we performed a read operation using the sales CRUD tool, which resulted in retrieving a list of sales data, including customers with missing email addresses, such as Charlie Brown with an email value of "N/A".

Here's the current table:

sales_id	customer_name	product_name	quantity	unit_price	total_price	sales_date	email
1	Alice Johnson	Widget	2	9.9900	19.9800	2025-09-02T06:46:49	alice@example.com
2	Bob Smith	Gadget	1	14.9900	14.9900	2025-09-02T06:46:49	bob@example.com
3	Charlie Brown	Tool	3	26.9900	79.9700	2025-09-02T06:46:49	N/A

Data formatted with Data Format Conversion - Dates converted to string format

Interactive Visualizations

Visualization: handle null values in sales

Generated Code

```

<\/pre>

```

Figure 1.19 – Handle Null Values

LLM at Scale.AI

Business transformation and industrial research

A leading multi-agent orchestration platform that embraces and accelerates Gen AI adoption, sustains it, and scales it safely.



Contact

 admin@llmatscale.ai
 <https://llmatscale.ai/>

USA | Singapore | India