Model Context Protocol: Enabling LLMs to Access and Reason Across Distributed Cloud Datasets

An LLM-Orchestrated Pipeline for Query Planning, Execution, and Merging Across Amazon S3 and Azure Blob Storage

**By:**

**Anxhesia Breshanaj**

**Supervisor: Harry Yu**

**Dissertation Submitted to the University of Derby in Partial Fulfilment of the**

**Requirements for the Award of Master of Science in Big Data Analytics**

**2024-2025**

#### Abstract

Large Language Models (LLMs) have demonstrated strong reasoning abilities but remain limited in accessing and analysing real-world datasets. Data is increasingly stored in cloud environments such as Amazon S3 and Microsoft Azure Blob Storage, where understanding the structure and contents of files is essential for effective use. This thesis presents a prototype system that uses the Model Context Protocol (MCP) to enable LLMs to perform simpler but critical tasks such as scanning individual datasets, interpreting descriptions and generating fact-based summaries. The system integrates a metadata pipeline that produces sidecar files describing dataset structure and content, which are then used by the LLM to ground its outputs. Evaluation with synthetic datasets demonstrated that the approach produces accurate file-level insights while clearly indicating gaps or missing information. These results highlight MCP’s potential to support practical, metadata-driven LLM applications for cloud-based data analysis.

Table of Contents

[List of Figures 3](#_Toc206696548)

[List of Tables 4](#_Toc206696549)

[Chapter 1: Introduction 4](#_Toc206696550)

[1.1 Background and Motivation 4](#_Toc206696551)

[1.2 Problem Statement 4](#_Toc206696552)

[1.3 Objectives of the Study 5](#_Toc206696553)

[1.4 Scope and Limitations 5](#_Toc206696554)

[1.5 Significance of the Study 6](#_Toc206696555)

[1.6 Organization of the Thesis 6](#_Toc206696556)

[Chapter 2: Literature Review 6](#_Toc206696557)

[2.1 Large Language Models and Their Capabilities 6](#_Toc206696558)

[2.2 Multi-Cloud Data Storage and Orchestration Challenges 7](#_Toc206696559)

[2.3 Use of LLMs in Dataset Planning 7](#_Toc206696560)

[2.4 Comparison with Existing Query Engines and Assistants 7](#_Toc206696561)

[2.5 Gaps in Current Research and How This Work Addresses Them 8](#_Toc206696562)

[2.6 Model Context Protocol (MCP) 8](#_Toc206696563)

[Chapter 3: Methodology 9](#_Toc206696564)

[3.1 Research Design 9](#_Toc206696565)

[3.2 Development Methodology 10](#_Toc206696566)

[3.3 System Architecture Overview 10](#_Toc206696567)

[3.4 Component Descriptions 11](#_Toc206696568)

[3.4.1 plan(): Query Planning Logic with LLM 11](#_Toc206696569)

[3.4.2 execute\_plan(): Client-Based Data Query Execution 12](#_Toc206696570)

[3.4.3 merge\_step(): Reasoned Answer Merging 13](#_Toc206696571)

[3.4.4 Metadata Pipeline: Automated File Profiling and Sidecar Management 15](#_Toc206696572)

[3.5 Cloud Integration Layer 19](#_Toc206696573)

[3.5.1 AWS S3 Integration (client\_s3.py) 19](#_Toc206696574)

[3.5.2 Azure Blob Integration (client\_azure.py) 22](#_Toc206696575)

[3.6 Interface Design (CLI and Streamlit UI) 24](#_Toc206696576)

[3.7 Summary 25](#_Toc206696577)

[Chapter 4: Evaluation and Results 26](#_Toc206696578)

[4.1 Evaluation Approach 26](#_Toc206696579)

[4.2 Planning Behavior and File Selection 27](#_Toc206696580)

[4.3 Execution of Sub-Questions 30](#_Toc206696581)

[4.4 Merging and Reasoning Across Results 31](#_Toc206696582)

[4.4.1 Successful Merges 31](#_Toc206696583)

[4.4.2 Partial or Failed Merges 33](#_Toc206696584)

[4.4.3 Role of Data Gap Annotations 34](#_Toc206696585)

[4.4.4 Reasoning Across Sources 34](#_Toc206696586)

[4.5 Examples of System Behaviours 35](#_Toc206696587)

[4.6 Summary of Results 36](#_Toc206696588)

[Chapter 5: Conclusion and Future Work 37](#_Toc206696589)

[5.1 Future Work 38](#_Toc206696590)

[5.2 Final Thoughts 38](#_Toc206696591)

[References 38](#_Toc206696592)

# List of Figures

[Figure 1:Planning Prompt Structure 12](#_Toc206696518)

[Figure 2:Execution Workflow 12](#_Toc206696519)

[Figure 3:Plan Output 13](#_Toc206696520)

[Figure 4: Merge Step Prompt Construction 14](#_Toc206696521)

[Figure 5:Execution Results Passed to Merge Step 15](#_Toc206696522)

[Figure 6:Metadata Pipeline Integration Flow 16](#_Toc206696523)

[Figure 7: Partial File Header Retrieval in core.py 16](#_Toc206696524)

[Figure 8:Primary Key and Date Range Detection in auto\_desc.py 17](#_Toc206696525)

[Figure 9:Example Sidecar Metadata for sales.csv 18](#_Toc206696526)

[Figure 10: FileMeta Dataclass Structure in metadata\_store.py 18](#_Toc206696527)

[Figure 11: MCP Persistence and Retrieval in SidecarStore 19](#_Toc206696528)

[Figure 12: AWS S3 Client Integration Flow 19](#_Toc206696529)

[Figure 13: S3 File Discovery in \_list\_meta() 20](#_Toc206696530)

[Figure 14: Auto-Prescan Workflow in \_auto\_prescan() 21](#_Toc206696531)

[Figure 15: S3 Query Execution Prompt in answer() 22](#_Toc206696532)

[Figure 16: Azure Blob Client Integration Flow 22](#_Toc206696533)

[Figure 17: Azure Blob Discovery in \_list\_meta() 23](#_Toc206696534)

[Figure 18: Prescan Workflow in \_prescan() 24](#_Toc206696535)

[Figure 19: Azure Query Execution Prompt in answer() 24](#_Toc206696536)

[Figure 20: Dual Interface Sequence Flow 25](#_Toc206696537)

[Figure 21: Structured Planning Prompt with Metadata Injection 28](#_Toc206696538)

[Figure 22: Example JSON Plan Output 29](#_Toc206696539)

[Figure 23: Pre-Execution Plan Validation 29](#_Toc206696540)

[Figure 24: Step-by-Step (1,2) Execution Results for LA Customer Query 30](#_Toc206696541)

[Figure 25: Step-by-Step (3,4) Execution Results for LA Customer Query 30](#_Toc206696542)

[Figure 26: Example of a Successful Merge Output for Sales Revenue Query 32](#_Toc206696543)

[Figure 27: Example of a Successful Merge Output for Sales Revenue Query 33](#_Toc206696544)

[Figure 28: Example of a Partial Merge Output for LA Customer Body-Type Query 34](#_Toc206696545)

[Figure 29:Example of a Merge Output for VINs serviced Query 36](#_Toc206696546)

# List of Tables

[Table 1: Classification of Query Outcomes by Root Cause 36](#_Toc206696547)

# Chapter 1: Introduction

## 1.1 Background and Motivation

In the last few years, Large Language Models (LLMs) have transformed how we think about machine reasoning and natural language understanding. Models like OpenAI’s GPT and Anthropic’s Claude have demonstrated the ability to interpret, summarize, and reason about a wide range of queries using plain English (OpenAI, 2023; Anthropic, 2024). At the same time, the way organizations manage and store their data has shifted. Instead of relying on a single central database or storage system, many businesses for different uses and clients use different cloud services to keep their data, most commonly AWS and Microsoft Azure.

While this distributed model offers flexibility, it introduces complexity. To analyse or even locate specific data, users often need to know exactly which file to open, what format it is in, and how to filter or process the contents. This becomes even more challenging when the number of files grow and are spread across different platforms or organized inconsistently. In my experience you must rely on specialized tools to answer even basic questions as there is not any tool to unify all in one. This manual process slows down decision-making and limits accessibility to those with technical expertise.

This research is motivated by the fact that LLMs are already very good at understanding language and interpreting meaning, but they cannot directly access real‑time data or private files without external retrieval mechanisms. They can explain a concept or write a summary, but they cannot open a file in cloud storage or figure out which dataset contains the answer to a specific question. On the other hand, traditional tools for data querying, like dashboards or SQL engines, can read files but cannot reason or interpret natural language in a flexible way. This disconnect between language understanding and data access is the problem space in which this thesis is situated.

## 1.2 Problem Statement

The central problem addressed by this thesis is how to enable LLMs to reason over and extract insights from real-world data that is distributed across multiple cloud storage systems, particularly Amazon S3 and Microsoft Azure Blob Storage. Although LLMs such as OpenAI’s GPT-4 or Anthropic’s Claude have demonstrated strong capabilities in interpreting and generating natural language, they operate as isolated language agents with no direct access to live data stored in cloud environments (OpenAI, 2023). In contrast, enterprise datasets are increasingly fragmented across different platforms, each with its own APIs, formats, and metadata conventions, making unified querying a complex and often manual task (Ghosh, 2023). Analysts or developers are typically required to know in advance which files to access, how they are structured, and how to combine them often using SQL queries or platform-specific tools. This setup restricts data accessibility to those with technical knowledge and does not scale well as organizations accumulate more cloud-native storage over time.

While tools like BigQuery Omni attempt cross-cloud federation, they are constrained by vendor lock-in and require heavy configuration (Google Cloud, 2020). BigQuery Omni allows federated querying across AWS and Azure but ties users into the Google Cloud ecosystem. Even open source query engines like Trino can execute SQL across multiple data sources, yet they demand significant technical setup and provide no semantic understanding of the data (Trino, n.d.). At the same time, most LLMs are limited to reasoning about abstract prompts and cannot independently determine where relevant information resides or how to retrieve it.

This thesis investigates whether it is possible to bridge this gap by turning the LLM into an active orchestrator that can use sidecar file summaries to generate multi-step reasoning plans. These plans specify which datasets to query, how to formulate cloud-specific sub-questions, and how to merge the results into a unified response. The problem is not just about enabling natural language querying, but about coordinating cloud-aware reasoning workflows across independently stored datasets, a task that requires both language understanding and structured system-level orchestration to interface with cloud systems.

## 1.3 Objectives of the Study

The first goal of this study is to carry out a detailed literature review of LLMs and their use in data-related reasoning tasks. This includes studying the capabilities and limitations of models like Claude, GPT, and similar LLMs, as well as examining related systems that attempt to bridge the gap between natural language and structured data.

The second goal is to design and implement a working system that makes use of the Model Context Protocol. MCP provides a standard way for the LLM to connect to external resources such as cloud storage. In this project, MCP is used to build a coordinating client that can expose S3 and Azure Blob Storage as accessible resources for the LLM. The system then allows the LLM to plan actions, execute queries through the cloud-specific clients, and merge the answers into a unified summary.

The third and final goal is to evaluate how accurate and helpful the system is when answering natural language questions over simulated data. This involves analysing how well the planner selects the right files, whether the queries produce correct answers, and how coherent the final responses are.

## 1.4 Scope and Limitations

This study is focused on building a system that connects language understanding to cloud-based data access. The scope includes the use of LLMs for planning, executing, and merging queries across files located in both Amazon S3 and Azure Blob Storage. One important feature of the system is its support for sidecar file files, which are used to store short descriptions of each dataset. These descriptions are generated automatically by scanning the beginning of each file and using the LLM to summarize what kind of data is present. The system uses this information, along with inferred file schemas, to plan how a user query should be handled. Another major part of the system is its ability to execute sub-questions on the correct file and then combine the answers with internal consistency checks. The merging process is also handled by the LLM, which ensures that answers from different files and clouds are synthesized into a single, response that is easy to understand.

However, it is important to note that the current version of the system has been developed using randomly generated data. As a result, the evaluation is based on structural correctness and functional behavior, rather than on business accuracy insights. The system also assumes that the files are already in readable format and contain clean, well-structured data, there is no data cleaning, transformation, or preprocessing involved. Additionally, while the system is built on the MCP to connect an LLM with cloud-based datasets, it does not currently support real-time streaming, advanced security integration, or large-scale data crawling. Its goal is to demonstrate reasoning, orchestration, and question answering over cloud-hosted static datasets in a proof-of-concept setting.

## 1.5 Significance of the Study

The significance of this work lies in its attempt to connect two major areas: language understanding and cloud-based data access. As LLMs become more capable, their usefulness will increasingly depend on their ability to interact with real data. Meanwhile, cloud storage continues to dominate the way data is managed especially as more businesses adopt multi-cloud strategies for resilience and flexibility (Barr, 2024). By integrating an LLM into a system that can plan and execute multi-step data queries across different clouds, this project explores a practical path toward more intelligent, conversational interfaces for data analysis.

This system also has the potential to reduce the reliance on technical tools or programming skills when accessing data insights. Users can simply ask what they want to know, and the system will figure out where to look and how to get the answer. In this way, the project contributes to the larger vision of building AI agents that can understand questions, find relevant information, and present helpful responses, all without requiring users to know the technical details of the data infrastructure.

## 1.6 Organization of the Thesis

This thesis is divided into five chapters. Following this introductory chapter, Chapter 2 provides a literature review of LLMs, MCP coordination, and related systems. Chapter 3 explains the methodology used to build the MCP system, including its architecture, components, and implementation details. Chapter 4 presents the evaluation of the system using test queries on simulated datasets. Chapter 5 concludes the work and suggests future directions for improvement and application in real-world environments.

# Chapter 2: Literature Review

## 2.1 Large Language Models and Their Capabilities

The recent progress in artificial intelligence has been shaped significantly by the development of LLMs. These models, built primarily on the transformer architecture introduced by Vaswani et al. (2017), have enabled systems to handle dependencies and perform complex language-based tasks through attention mechanisms. Models such as OpenAI's GPT-3 and GPT-4, Anthropic's Claude, and Meta's LLaMA represent milestones in this field, each expanding the scale and capability of previous iterations (Brown et al., 2020; OpenAI, 2023; Anthropic, 2023; Meta AI, 2023).

LLMs are trained on large corpora of text data, enabling them to perform tasks such as summarisation, translation, question answering, and code generation. Their ability to follow instructions and their ability to learn from few examples has made them useful across a wide range of domains (Brown et al., 2020). However, these models are inherently limited by their static knowledge base and can produce confident yet incorrect outputs, a phenomenon known as hallucination (Ji et al., 2023). This issue arises because LLMs generate responses based on probability rather than verified fact.

To mitigate such issues, research has explored extending LLMs with external tools, such as through the ReAct framework, which enables models to interleave reasoning steps with information retrieval or computation (Yao et al., 2023). LangChain has further operationalised this concept by providing developers with modular tool interfaces that connect LLMs to APIs, databases, and file systems (LangChain, 2023). These developments represent a crucial shift from static text generation to dynamic, grounded reasoning.

## 2.2 Multi-Cloud Data Storage and Orchestration Challenges

In modern enterprise environments, data is often stored across multiple cloud platforms, including Amazon S3, Azure Blob Storage, and Google Cloud Storage. Each of these services offers scalable object storage but with differing interfaces, metadata schemas, and governance models. This heterogeneity introduces challenges for unified data analysis (Ghosh, 2023).

A fundamental problem is the fragmentation of storage and metadata. There is no default interoperability between platforms, and each service has unique methods for organising and accessing files. Many organisations accumulate cloud services incrementally, resulting in separated data sets (Barr, 2024). Querying across these systems requires complex orchestration and often manual integration.

While solutions such as AWS Glue and BigQuery Omni attempt to bridge this gap, they often remain specific or require substantial setup (Google Cloud, 2020). BigQuery Omni, for example, enables querying by deploying processing across clouds but ties users into the Google ecosystem. As a result, there is a growing need for cloud-agnostic orchestration systems that can manage access, governance, and semantic alignment across providers without requiring full data centralisation.

## 2.3 Use of LLMs in Dataset Planning

The application of LLMs to dataset planning and navigation is a growing area of interest. Text-to-SQL systems have demonstrated that LLMs can map natural language queries to structured database commands, but this typically assumes a static schema and a single database backend (Rajkumar et al., 2022). They do not address scenarios with many unstructured files or multiple storage locations.

Recent innovations go further by integrating LLMs with metadata-driven discovery tools. LEDD demonstrated how LLMs can be used to auto-summarise data tables, organise them thematically, and recommend join paths, creating a kind of semantic data map (An et al., 2023). LangChain and ReAct, discussed earlier, further enable LLMs to serve as planning agents, generating task sequences and invoking data access tools in context-aware ways (Yao et al., 2023; LangChain, 2023).

These systems illustrate that LLMs are capable not only of query translation but also of intelligent orchestration. By leveraging sidecar file or lightweight previews of datasets, LLMs can reason about content structure and relevance before data access begins. This marks a shift toward semantic navigation of data lakes and warehouses through conversational interfaces, reducing the dependency on schema expertise.

## 2.4 Comparison with Existing Query Engines and Assistants

Existing query tools such as Power BI Q&A, Tableau Ask Data, and traditional SQL-based engines like Trino and Presto offer structured querying capabilities but often require predefined schemas or metadata mappings. These tools generally lack the contextual understanding and adaptability of LLMs (Microsoft, 2024; Tableau, 2024; Trino, n.d.).

For instance, Power BI Q&A enables users to ask natural language questions, but only within the bounds of a preconfigured dataset that has been set up by analysts, with known fields and relationships, and do not generalize to arbitrary files or new data sources (Microsoft, 2024). Similarly, Tableau’s Ask Data, a feature that let users query in natural language, while promising, was recently deprecated, reflecting the challenges of sustaining flexible NL interfaces in production (Tableau, 2024).

Distributed query engines like Trino enable SQL queries across heterogeneous data sources but assume technical proficiency and offer no semantic guidance (Trino, n.d.). Newer AI-powered tools such as Amazon QuickSight Q and Microsoft Copilot enhance usability through LLM integration but still depend on curated metadata and are often limited to single-cloud environments (AWS, 2023; Microsoft, 2023).

In contrast, LLM-based orchestration agent aims to dynamically plan queries across multiple datasets and clouds. This is a potential advancement in both flexibility and usability, though current implementations remain experimental and require safeguards against hallucination and performance unpredictability.

## 2.5 Gaps in Current Research and How This Work Addresses Them

From my review of the literature, several gaps are apparent. Firstly, while LLMs have demonstrated promise in isolated tasks like text-to-SQL or metadata summarisation, their use as orchestrators across distributed cloud data is still underexplored. There is limited published work showing LLMs autonomously planning queries across cloud providers such as S3 and Azure.

Secondly, most existing systems assume a static or tightly integrated data environment. They are not built for scenarios where datasets vary in schema, location, and access policy. This leaves a gap for systems that combine semantic reasoning with cloud-native access mechanisms.

Thirdly, few systems provide mechanisms to validate that LLM-generated answers are grounded in actual file content. Without such grounding, systems risk producing coherent but inaccurate outputs. Retrieval-augmented generation has helped in document QA, but its application to tabular or structured data retrieval remains limited (Lewis et al., 2020).

My research aims to bridge these gaps by evaluating whether an LLM can be used not just to interpret queries but also to orchestrate a multi-step reasoning and execution process, grounded in file-level metadata and real-time cloud access. This direction builds on the foundations laid by prior research while extending their applicability through the MCP, which provides a standardised way to expose datasets and tools to the LLM across heterogeneous cloud storage environments.

## 2.6 Model Context Protocol (MCP)

MCP is a recent standard that defines how LLMs can connect to external systems in a structured way. Instead of relying on one-off integrations or custom scripts, MCP provides a common format for describing what resources or functions are available, how they can be called, and how results should be returned (Anthropic, 2024). This gives the model a consistent way to “see” what it can access and to decide how to use it.

At its core, MCP allows an LLM to discover a list of available actions send a request with clear arguments and receive a structured response. This makes the interaction more predictable than free form prompting and reduces errors. It also allows new resources to be added without changing the LLM itself only the MCP server needs to describe the new function.

Researchers have shown that LLMs are more reliable when they are grounded in structured calls to external systems rather than generating answers without context (Yao et al., 2023; Lewis et al., 2020). MCP builds on this idea by offering a shared standard for these interactions. Early implementations have used MCP to connect LLMs with local file systems, cloud storage services, and databases, showing that it can serve as a bridge between language reasoning and live data access (Anthropic, 2024).

In this thesis, MCP is used to wrap cloud-specific clients for Amazon S3 and Azure Blob Storage. The orchestrator relies on MCP both to check what functions are available (e.g., listing files, reading a header preview) and to call them step by step as part of an execution plan. By using MCP, the system ensures that the LLM does not invent file paths or commands but instead works only with the resources that have been described and exposed through the protocol. This makes the reasoning process clearer, safer, and easier to extend in future versions.

# Chapter 3: Methodology

This chapter outlines the design and development process of my thesis, a prototype system that uses a Large Language Model (LLM) to coordinate querying across Amazon S3 and Azure Blob Storage. The system allows users to ask natural language questions, which the LLM transforms into multi-step, cross-cloud execution plans. The architecture is composed of modular Python components that support planning, querying, merging, and summarizing all driven by metadata-aware prompts.

## 3.1 Research Design

The system is designed around a layered architecture that separates concerns while maintaining tight integration between components. It comprises four principal layers: User Interface, Orchestration Engine, Metadata System, and Cloud Integration Layer.

The User Interface Layer provides two distinct entry points. A command-line interface (CLI) serves as a developer tool, offering low-level interaction for testing and diagnostics. A Streamlit based web application offers a more user-friendly interface intended for analysts and operational use. Both interfaces connect directly to the orchestration engine to submit user queries.

The Orchestration Engine (client.py) is built around the Model Context Protocol (MCP). MCP defines how the LLM can discover which resources are available, call them with clear arguments, and receive structured results. In practice, this means that when a user asks a question, the orchestrator uses MCP to expose the available datasets and functions, then plans and executes the steps needed to answer. Internally, it incorporates three roles: a planner for breaking down queries, an executor for running sub-queries, and a merger for synthesising outputs.

The Metadata System (metadata\_store.py) functions as the knowledge backbone of the architecture. It stores descriptive information, inferred schemas, and key relationships for each file in .meta.json sidecar files. These are structured according to the FileMeta dataclass and stored alongside the original datasets in the cloud. The SidecarStore class provides a uniform interface for retrieving this metadata, automatically routing requests to the correct storage backend based.

The Cloud Integration Layer implements platform-specific operations, using the boto3 SDK for Amazon S3 and the azure-storage-blob SDK for Azure Blob Storage (AWS, 2024; Microsoft, 2024). This layer handles tasks such as listing files, retrieving partial content for analysis, and extracting schema information, while ensuring minimal data transfer and maintaining compatibility with the orchestrator.

All layers interface with a central Large Language Model (Claude 3.5 Haiku), which provides semantic understanding, query planning, and summarisation. By combining MCP-driven orchestration with metadata-aware planning and cloud-specific access, the system enables accurate, efficient, and explainable query execution across heterogeneous storage environments.

## 3.2 Development Methodology

The development of the system followed an incremental, component-driven approach, beginning with the implementation of core cloud access capabilities and gradually expanding to include metadata management, planning logic, and orchestration. The sequence of work was deliberately structured so that each new feature could be built upon a tested, reliable foundation.

The first stage focused on integrating the system with cloud storage services. Amazon S3 access was implemented using the boto3 SDK, enabling object listing and partial content retrieval of the first 64 KB of each file to minimise data transfer. In parallel, Azure Blob Storage integration was developed using the azure-storage-blob SDK, incorporating Azure-specific authentication handling. Both modules were validated to ensure stable authentication, correct object listing, and reliable partial retrieval.

Once these cloud access modules were in place, the next stage introduced a metadata pipeline based on a sidecar mechanism. This design allowed the system to maintain descriptive, schema-level metadata without rescanning entire files on each query. The core.py module was developed to download file partially and prompt the Claude 3.5 Haiku LLM to generate concise factual descriptions. These descriptions, along with MIME type, size, and last-scanned timestamp, were stored as .meta.json sidecar files in the same cloud location via the SidecarStore class. To further enhance metadata utility, the auto\_desc.py module was implemented to analyse CSV structures, detect column names, identify primary and foreign keys, and determine date ranges. This enriched metadata would later allow the LLM to plan more accurate and efficient queries.

With cloud access and metadata generation complete, the orchestration engine (client.py) was implemented to convert natural-language questions into structured, executable plans. The plan() function generated these plans using LLM prompts grounded in available metadata, ensuring that only valid datasets and schemas were referenced. The execute\_plan() function dispatched sub-queries to the appropriate cloud-specific client, leveraging cached metadata wherever possible to reduce data transfer. Finally, the merge\_step() function synthesised results from multiple sources, performing deduplication and consistency checks, and producing a unified, fact-checked answer.

The final stage of development addressed user interaction. Two interfaces were implemented: a command-line interface (CLI) for rapid testing and automation, and a Streamlit-based web application for interactive use. The CLI displays the full reasoning chain including the generated execution plan, intermediate results from each cloud-specific query, and the final merged answer making it useful for debugging and detailed inspection. In contrast, the Streamlit interface is designed for end users, providing a clean, simplified experience where only the final answer is shown. This separation allows developers to work with full transparency during testing, while keeping the production-facing interface focused on concise, actionable outputs.

This staged methodology ensured that each major capability cloud access, metadata enrichment, query orchestration, and user interaction was developed on a stable, verified foundation. At every stage, earlier components were validated before proceeding, reducing integration risks and resulting in a robust, maintainable final system.

## 3.3 System Architecture Overview

The system adopts a metadata-driven orchestration architecture, where the Model Context Protocol (MCP) coordinates how the LLM interacts with datasets stored in multiple clouds. At its core is the orchestrator (client.py), which manages the complete query lifecycle through three sequential stages:

Firstly planning, the orchestrator analyses sidecar metadata, including file descriptions, detected schemas, and inferred relationships, to generate a structured execution plan. This plan specifies which datasets to query, their corresponding cloud location (Amazon S3 or Azure Blob Storage), and a tailored sub-question for each dataset.

Secondly, executing based on the plan, sub-queries are dispatched to cloud-specific clients. These clients retrieve minimal data segments, update missing metadata where necessary, and ensure that responses remain grounded in actual file content.

And finaly merging partial results from multiple sources are synthesised into a unified output through LLM-guided fact extraction, deduplication, and consistency checks.

Supporting these stages is a sidecar metadata pipeline, which maintains lightweight .meta.json descriptors co-located with each file in its originating cloud. This enables efficient planning and execution without the need to download entire datasets.

## 3.4 Component Descriptions

### 3.4.1 plan(): Query Planning Logic with LLM

The plan() function is responsible for converting a user’s natural language query into a structured, multi-step execution plan grounded entirely in verified metadata. To achieve this, the function first assembles three critical inputs: the dataset\_block, containing descriptive summaries of each dataset; the schema\_block, detailing inferred schemas and column names; and the files\_list, enumerating valid dataset paths with their corresponding cloud location. These inputs are dynamically inserted into a structured LLM prompt, shown in Figure 1, which clearly separates role definition, contextual information, and the user’s original question.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 1:Planning Prompt Structure

Structured prompt showing LLM prompt with rules.

The prompt is designed with explicit constraints to keep the LLM’s reasoning aligned with reality. These rules prohibit the use of datasets not present in the metadata, require accurate cloud assignments (“s3” or “azure”), and mandate that joins only occur when a shared key exists between datasets. Filtering conditions must refer to actual schema fields, plans are restricted to a maximum of three steps, and output must be in pure JSON format. If no valid datasets can be matched, the LLM is instructed to return an empty steps array. Embedding these constraints directly into the prompt ensures that any plan produced is both executable and grounded in actual data availability.

The prepared prompt is sent to the Claude 3.5 Haiku model using the LLM.messages.create() API with a temperature setting of 0.0 to enforce deterministic behaviour. The model’s response is streamed, parsed, and converted into a JSON object representing the execution plan. This structured output forms the foundation for the subsequent execution stage, ensuring that the system operates with clear, validated instructions derived entirely from known metadata.

### 3.4.2 execute\_plan(): Client-Based Data Query Execution

The execute\_plan() function takes the JSON plan generated by plan() and runs each step sequentially, passing the output of one step as contextual input to the next. Each step in the plan specifies the target dataset, its cloud location, and the sub-question to be executed against it. Before execution begins, the function verifies that all required fields are present specifically, that both cloud and dataset are defined. If either is missing, the step is skipped, and a warning is recorded in the results log, as illustrated in Figure 2.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 2:Execution Workflow

Core execution logic demonstrating sequential processing

For each valid step, the function constructs an enhanced\_prompt using Python f-string formatting, embedding the dataset path, cloud type, and the specific sub-question. This prompt instructs the LLM to act as a data extraction expert, returning only direct results from the file contents. It also defines a fallback response (“No data found for file…”) to ensure consistent handling of missing or unavailable datasets. The prompt is then sent to the appropriate cloud-specific client: s3\_answer() for Amazon S3 datasets or az\_answer() for Azure Blob Storage datasets.

Figure 3 shows a real example of an execution plan in action, where multiple steps, including filtering, joins, and aggregations are run in sequence. As each step completes, its result is appended to a results list, and the textual output is also stored in a history list. This history enables subsequent steps to incorporate information from earlier ones, supporting more complex reasoning and multi-stage queries.

A computer code with black text

AI-generated content may be incorrect.

Figure 3:Plan Output

Sample plan showing join condition detection and cloud-specific path handling. Note strict adherence to valid paths from metadata.

By combining strict input validation, consistent prompt construction, and selective cloud client dispatch, the execute\_plan() function ensures that every execution step is precise, traceable, and fully aligned with the validated plan produced in the earlier stage. This design prevents the execution of malformed or hallucinated instructions and guarantees that the orchestration engine interacts only with legitimate datasets in the specified cloud environments.

### 3.4.3 merge\_step(): Reasoned Answer Merging

The merge\_step() function is designed to consolidate two intermediate answers, each originating from a different cloud source, into a single, coherent output. It operates iteratively within the orchestrator, merging one fragment with another until only the final, unified answer remains. This approach ensures that information from all sources is compared, reconciled, and synthesised before being presented to the user.

When invoked, the function receives two answer fragments alongside their corresponding cloud identifiers and the original user question. As shown in Figure 4, the merge prompt explicitly presents both fragments, tagging them by source so the LLM can maintain traceability of each fact. The instructions within the prompt are deliberately detailed: the model is asked to extract all actionable facts, deduplicate overlapping information while preserving multi-source attribution, and check for inconsistencies by identifying items that appear in one fragment but are missing from the other. Any such discrepancy is flagged as a “Data Gap,” clearly stating which cloud lacks the information.

A screenshot of a computer code

AI-generated content may be incorrect.

Figure 4: Merge Step Prompt Construction

The \_merge\_step function with the prompt provided to the LLM during the merge phase.

The final step in the prompt instructs the LLM to synthesise a cohesive answer of no more than 180 words, using only the verified facts, and to conclude with a bold “Executive Takeaway” sentence. This ensures that the output remains concise, factually grounded, and easily interpretable. Figure 5 provides an example of execution results from earlier stages, which are passed into the merge function for reconciliation. Here, customer lists, sales matches, and inventory lookups from different steps are evaluated side-by-side, with missing data points explicitly noted.

A white background with black text

AI-generated content may be incorrect.

Figure 5:Execution Results Passed to Merge Step

An example set of intermediate outputs from earlier execution steps. Used as input to the merge process.

Technically, the merge process is implemented using the LLM.messages.create() method, configured with a slightly higher temperature of 0.1 to allow for more natural summarisation while still maintaining structure and factual accuracy. Token usage is capped at 1000 to prevent overly verbose outputs. By enforcing these constraints and following a repeatable, step-by-step merging sequence, the system ensures that the LLM acts not only as a summariser but also as a fact-checker, producing a final result that is both accurate and transparent.

### 3.4.4 Metadata Pipeline: Automated File Profiling and Sidecar Management

The metadata pipeline is the foundation of the system’s metadata-driven query planning, enabling the orchestrator to operate on structured semantic and technical information without requiring full dataset downloads. Its operation is summarised in the integration flow shown in Figure 6, where the process begins with the Core scanner (core.py) retrieving a header sample from the target file. This is performed via cloud-specific logic: Amazon S3 retrieval uses boto3.get\_object() with a byte range request, while Azure Blob Storage retrieval uses BlobClient.download\_blob() with explicit offset and length parameters. This process, illustrated in the \_download\_head() function in Figure 7, ensures consistent 64KB previews regardless of the storage backend.

A diagram of a cloud

AI-generated content may be incorrect.

Figure 6:Metadata Pipeline Integration Flow

Sequence diagram showing the interaction between Core, AutoDesc, MetadataStore, Cloud, and Orchestrator components.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 7: Partial File Header Retrieval in core.py

The implementation of partial file retrival supporting both Amazon S3 and Azure Blob Storage

The sampled header is then passed to the AutoDesc profiler (auto\_desc.py), which performs structural analysis on the dataset. This step detects primary keys and foreign keys using regex patterns, identifies temporal columns, and extracts date ranges to establish the temporal scope of the data. Schema inference is performed using Pandas, reading up to 5,000 rows to generate concise, human-readable descriptions such as "sales.csv · PK=order\_id · FK=customer\_id · Dates: 2023-01→2024-05". The implementation of this detection logic is shown in Figure 8, highlighting the regex-based PK/FK identification and temporal range extraction routines.

A computer screen shot of a code

AI-generated content may be incorrect.

Figure 8:Primary Key and Date Range Detection in auto\_desc.py

Code from the auto\_desc() showing structural analysis logic, including primary key detection, foreign key identification, and date range extraction from temporal fields.

An example of the final metadata produced from this process is presented in Figure 9, showing the .meta.json sidecar for the dataset sales.csv stored in Amazon S3. This file contains both semantic information, including a comprehensive dataset description, and technical details such as MIME type, file size, and last scanned timestamp. Together, these elements allow the orchestrator to understand dataset content and structure without directly inspecting the raw file.

A computer screen shot of a code

AI-generated content may be incorrect.

Figure 9:Example Sidecar Metadata for sales.csv

Generated .meta.json file representing the output of the AutoDesc profiling process for sales.csv in Amazon S3. It contains semantic descriptions alongside technical attributes used for planning and execution.

Once the structural and schema information is generated, the system creates a FileMeta object (Figure 10) containing both semantic metadata (description, title, cloud location) and technical metadata (MIME type, file size, last scanned timestamp). This object is then passed to the MetadataStore (metadata\_store.py), which handles storage and retrieval of .meta.json sidecar files. The write\_meta() method, shown in Figure 11, persists this metadata to the same cloud environment as the original dataset, automatically routing to S3 or Azure based on the path pattern.

A computer screen shot of a code

AI-generated content may be incorrect.

Figure 10: FileMeta Dataclass Structure in metadata\_store.py

Each sidecar stores both semantic descriptions and technical properties, enabling the orchestrator to perform schema-aware query planning.

A computer screen shot of a code

AI-generated content may be incorrect.

Figure 11: MCP Persistence and Retrieval in SidecarStore

Implementation of the write\_meta(), get\_meta(), and read\_all() methods within the SidecarStore class. These functions handle the full lifecycle of sidecar metadata, including saving to the correct cloud platform, retrieving individual records with cache support, and bulk loading metadata by prefix.

By integrating core.py, auto\_desc.py, and metadata\_store.py into a single automated process, the metadata pipeline delivers a complete lifecycle from physical file scanning and schema inference to cloud-native sidecar storage and retrieval. This enables the LLM-powered orchestrator to make informed, schema-aware query plans while keeping data transfers minimal and maintaining full compatibility with heterogeneous cloud storage environments.

## 3.5 Cloud Integration Layer

The Cloud Integration Layer connects the orchestration system to the storage platforms where the data is kept. In this project, it is responsible for handling access to Amazon S3 and Microsoft Azure Blob Storage. Each platform has its own way of storing and retrieving files, so dedicated clients were built to manage these differences. The layer takes care of tasks such as finding available files, reading a small part of a file to build metadata, and running the sub-queries created by the planner. By keeping this logic separate, the system stays easier to maintain and can be extended to other storage platforms in the future. The next subsections explain in more detail how the S3 and Azure Blob clients work, including how they manage discovery, metadata, and query execution.

### 3.5.1 AWS S3 Integration (client\_s3.py)

The AWS S3 client serves as the dedicated orchestration layer between the MCP orchestrator and Amazon S3 storage, implementing all S3-specific operations required for both metadata generation and query execution. As illustrated in Figure 12, client\_s3 acts as the intermediary, handling file discovery, metadata persistence, and S3-specific LLM prompt preparation.

A diagram of a software project

AI-generated content may be incorrect.

Figure 12: AWS S3 Client Integration Flow

Diagram showing client\_s3 as the intermediary between the orchestrator, Amazon S3 bucket, SidecarStore, and the metadata pipeline components (core.py and auto\_desc.py). It illustrates the flow of operations.

When invoked by the orchestrator during the execution of a plan, the S3 client first performs file discovery by listing objects in the configured bucket through the \_list\_meta() function (Figure 13). This process uses paginated calls to list\_objects\_v2 and automatically excludes .meta.json sidecar files to avoid redundant entries. For each file, it returns a minimal metadata dictionary containing the file path and title. This initial discovery ensures that the system has a complete and up-to-date inventory of available datasets before any further processing.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 13: S3 File Discovery in \_list\_meta()

Implementation of the \_list\_meta() function in client\_s3.py.

If a dataset has not been previously profiled, the client triggers the auto-prescan workflow (Figure 14). This step begins with a lightweight description generated by the auto\_describe() function using a small content sample retrieved via an S3 Range request. A FileMeta object is then created, populated with basic attributes, and stored in the cloud via the SidecarStore. Following this, the system calls scan\_object() from core.py to perform a more detailed scan, enriching the metadata with accurate MIME type, size, and potentially improved descriptions. This process ensures that every dataset has an associated .meta.json sidecar before it is used in query planning or execution.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 14: Auto-Prescan Workflow in \_auto\_prescan()

Code implementing the auto-prescan logic for unprofiled files. It retrieves a small content sample, generates a lightweight description via auto\_describe(), creates a FileMeta object, persists it using the SidecarStore, and triggers a full scan with scan\_object() to enrich metadata.

During query execution, the S3 client constructs LLM prompts with dataset-specific context (Figure 15). This includes the file path, schema (inferred where possible), and a preview of the file’s contents. The prompt is passed to Claude 3.5 Haiku with a temperature setting of 0.1 to prioritise factual accuracy. Execution results are parsed, and any new or improved descriptions returned by the LLM are written back into the metadata store, keeping sidecar information current. This tight integration between S3 data access, metadata enrichment, and prompt engineering ensures that the orchestrator can issue targeted, schema-aware queries without unnecessary full file downloads.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 15: S3 Query Execution Prompt in answer()

Section of the answer() function in client\_s3.py where dataset context is combined into an LLM prompt.

From a technical perspective, the S3 client supports AWS credential chain management, automatically falling back through environment variables, IAM roles, and CLI configuration. It implements robust error handling for throttling exceptions, retries failed requests, and falls back to binary sampling for malformed objects. Performance is improved through concurrent head-object requests and local caching of metadata during batch operations, while security is maintained via SSE-S3 encryption support and IAM policy validation before execution.

By abstracting S3-specific logic into a dedicated client, the system maintains clean separation between the orchestration logic and cloud storage implementation, enabling easier maintenance and extension to other cloud providers.

### 3.5.2 Azure Blob Integration (client\_azure.py)

The Azure Blob client provides the cloud-specific orchestration layer between the MCP orchestrator and Microsoft Azure Blob Storage, implementing all Azure-native operations for dataset discovery, metadata management, and query execution. As illustrated in Figure 16, client\_azure serves as the intermediary between the orchestrator and Azure storage, coordinating with the metadata pipeline and structural analysis components to ensure seamless integration.

A diagram of a computer

AI-generated content may be incorrect.

Figure 16: Azure Blob Client Integration Flow

Diagram showing client\_azure as the intermediary between the orchestrator, Azure container, SidecarStore, and the metadata pipeline components (core.py and auto\_desc.py). The flow illustrates execution requests, metadata handling, blob reading, and content scanning.

When triggered by the orchestrator, the Azure client begins with blob discovery via the \_list\_meta() function Figure 17. This process uses BlobServiceClient to enumerate all blobs within the configured container while automatically excluding .meta.json sidecar files. The function returns minimal metadata for each blob, including its path and title, enabling the orchestrator to maintain an accurate inventory of available datasets.

A computer screen shot of a computer code

AI-generated content may be incorrect.

Figure 17: Azure Blob Discovery in \_list\_meta()

Implementation of the \_list\_meta() function in client\_azure.py, using BlobServiceClient to list blobs in a container, exclude .meta.json files, and return minimal metadata for orchestrator use.

For any blob that lacks existing metadata, the client runs the prescan workflow implemented in \_prescan() Figure 18. This step creates a metadata for the blob, populating it with the path, cloud type, and basic attributes. The SidecarStore is then used to persist the metadata in the same Azure container. Once all of this is complete, the client calls scan\_object() from core.py to retrieve a partial file preview, analyse its structure, and generate a semantic description. This process enriches the metadata with schema details, primary and foreign keys, and temporal ranges, ensuring that each blob is fully profiled before query execution.

A computer code with many colored text

AI-generated content may be incorrect.

Figure 18: Prescan Workflow in \_prescan()

Code implementing the Azure prescan workflow, which creates a metadata scaffold for new blobs, saves it to the SidecarStore, and calls scan\_object() for structural analysis and schema enrichment.

During query execution, the Azure client constructs LLM prompts tailored for Azure datasets Figure 19. The prompt includes the blob’s path, any available schema information, and a preview of the file’s contents. This is passed to Claude 3.5 Haiku with a temperature setting of 0.1 to ensure factual precision. The execution results are parsed, and if the LLM produces new or refined descriptions, these are written back into the metadata store, keeping the sidecars in sync with the latest analysis. The prompt-building process mirrors that used in the S3 client, maintaining a consistent interface across clouds while allowing for Azure-specific schema hints and format handling.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 19: Azure Query Execution Prompt in answer()

Section of the answer() function in client\_azure.py that builds Azure-contextualised LLM prompts with blob path, schema, content preview and updates metadata with refined descriptions.

In this implementation, authentication is handled via an Azure Storage connection string retrieved from environment variables. By isolating Azure-specific logic in a dedicated client, the system achieves clean separation of concerns and ensures that the orchestrator can reason uniformly across both AWS and Azure environments, preserving the ability of the MCP (Model Context Protocol) architecture to work across different storage systems.

## 3.6 Interface Design (CLI and Streamlit UI)

The system offers two user-facing interfaces, a Command-Line Interface (CLI) for developers and automation workflows, and a Streamlit web interface for interactive exploration by analysts. Both interfaces connect to the same backend through the client.answer() method, ensuring consistent query execution and results regardless of the access channel. The interaction flow for both is shown in Figure 20.

**A diagram of a cloud

AI-generated content may be incorrect.**

Figure 20: Dual Interface Sequence Flow

Sequence diagram showing the identical backend process for CLI and Streamlit UI, from query submission to result presentation.

The CLI is a lightweight, dependency-free tool that can be run in one-shot mode or as an interactive REPL session. It is optimised for low-latency querying, script integration, and verbose output for debugging, including full execution plans and intermediate results.

The Streamlit interface is designed to remain responsive by offloading execution and caching intermediate results.

Although their presentation differs, both interfaces share the same capabilities, including query history, error handling, and metadata preview. The choice of interface depends on the user’s workflow: CLI for speed and automation, Streamlit for accessibility and visual feedback.

## 3.7 Summary

This chapter detailed the methodology behind the design and implementation of a metadata-driven system built on the Model Context Protocol (MCP), which enables Large Language Models (LLMs) to execute structured queries across Amazon S3 and Azure Blob Storage. The architecture is organised into four layers: a User Interface Layer, an Orchestration Engine, a Metadata System, and a Cloud Integration Layer. MCP serves as the standard interface that allows the LLM to discover available tools (such as S3 and Azure clients), call them with structured prompts, and receive consistent outputs. These layers are tightly integrated through a metadata pipeline that stores enriched .meta.json sidecar files alongside datasets, enabling the LLM to plan and reason efficiently without downloading entire files.

By combining cloud-specific SDKs, structured metadata management, schema-aware prompt engineering, and MCP-driven tool exposure, the implementation demonstrates a robust approach to LLM guided querying across distributed storage. The integration of boto3 for Amazon S3, azure storage blob for Azure Blob Storage, pandas and regex for schema analysis, Streamlit for web based interaction, Anthropic’s SDK for LLM communication, and Python’s standard libraries such as json, os, and datetime results in an architecture that is modular, standards-based, and optimised for efficiency, accuracy, and maintainability. Collectively, these design choices provide a solid foundation for the evaluation of performance, accuracy, and usability in the next chapter.

# Chapter 4: Evaluation and Results

## 4.1 Evaluation Approach

The evaluation of the system focuses on testing whether the implemented pipeline, built on the Model Context Protocol (MCP), can accurately interpret natural language questions, identify the relevant datasets, execute targeted queries on Amazon S3 and Azure Blob Storage, and merge the results into a coherent final answer. This proof-of-concept evaluation emphasises structural correctness and functional behaviour rather than domain specific accuracy, due to the use of synthetic datasets.

The datasets used in this evaluation were generated to simulate realistic but artificial business scenarios, including automotive sales, dealership inventories, customer financing, vehicle servicing, and supplier transactions. While these datasets provide representative schemas and data structures, their content is not drawn from real world operations. Consequently, the system sometimes produces results that include Data Gap annotations explicit indicators that certain information was unavailable in the selected datasets. Rather than being an error, this behaviour demonstrates the system’s ability to detect incomplete data and communicate these gaps to the user, a capability that is critical when working in heterogeneous, distributed environments.

Testing was conducted exclusively with the Anthropic Claude 3.5 Haiku (2024-10-22) model, selected for its lower token cost and fast response times compared to larger LLMs, while still offering sufficient reasoning performance for the target tasks. Although the architecture supports interchangeable models (including larger Claude variants and OpenAI GPT series), restricting evaluation to a single model allowed for consistent behaviour analysis while controlling operational costs.

The evaluation process follows a consistent five-stage workflow. Query Submission, where a user enters a natural language question through either the CLI or Streamlit interface. The planner (plan()) generates a JSON-formatted execution plan, grounded in metadata and constrained by explicit rules. Then executor (execute\_plan()) runs each sub-question against the correct cloud client (S3 or Azure) using cloud-specific previews and schemas. After that the merge function (merge\_step()) combines partial results iteratively, deduplicating facts and marking Data Gaps where applicable. Finaly, fact-checked summary is presented to the user.

This methodology ensures that each stage of the system’s reasoning from file selection to final summarisation can be examined in isolation and in sequence. The following sections (4.2–4.5) provide a detailed breakdown of the system’s behaviour in each stage, supported by screenshots and sample outputs.

## 4.2 Planning Behavior and File Selection

The planning stage is where the system transforms a user’s natural language question into a structured, multi-step execution plan that is both schema-aware and cloud-specific. This process is handled by the plan() function, which uses sidecar metadata, inferred schemas, and a strict rule set to ensure that the generated plan is both valid and executable.

When a query is received, the planner first gathers three key metadata blocks:

Dataset Block – A list of datasets with their LLM-generated descriptions from .meta.json sidecar files.

Schema Block – Inferred column structures, data types, and detected relationships such as primary/foreign keys.

Valid Paths List – Exact dataset paths and associated cloud platforms, ensuring the LLM can only select from verified files.

These components are then inserted into a structured LLM prompt that defines the model’s role as a planner operating through the Model Context Protocol (MCP). The prompt is divided into sections: role definition, dataset descriptions, schemas, valid file paths and the user’s original question, followed by a strict rule set. These rules enforce cloud-specific dataset selection, prevent hallucinated file names or columns, limit the plan to a maximum of three steps, and require join conditions to be based on shared keys present in the schema.

Figure 21 shows an example of this structured planning prompt, including the injected dataset and schema information. By grounding the LLM’s reasoning in verified metadata, the system ensures that file selection is not only relevant to the question but also executable without further human intervention.

A screenshot of a computer code

AI-generated content may be incorrect.

Figure 21: Structured Planning Prompt with Metadata Injection

Example of the planner’s LLM prompt showing dataset descriptions, schemas, and strict execution rules. This ensures that the LLM can only select valid datasets and avoids hallucinations.

Once the prompt is submitted to the LLM, the response is expected in pure JSON format, containing an array of steps. Each step includes:

• The dataset path (matching an entry from the Valid Paths List).

• The cloud identifier (“s3” or “azure”).

• A sub\_question specifically tailored to the dataset’s schema and content.

An example output is shown in Figure 22, where the planner has selected two datasets from different clouds and created targeted sub-questions for each. The JSON structure is intentionally minimal, allowing the executor to process it directly without additional parsing or interpretation.

A computer code with text

AI-generated content may be incorrect.

Figure 22: Example JSON Plan Output

Planner output showing the selected datasets, associated clouds, and schema-aware sub-questions. The strict adherence to verified file paths demonstrates successful grounding in available metadata.

To validate the robustness of the planning logic, the system includes a pre-execution check for malformed or hallucinated plan entries. As shown in Figure 23, any step referencing a non-existent dataset path or invalid cloud specification is automatically discarded, and a warning is logged. This ensures that execution proceeds only with datasets that exist in the configured cloud environments.

**A screenshot of a computer program

AI-generated content may be incorrect.**

Figure 23: Pre-Execution Plan Validation

Validation logic that prevents execution of hallucinated dataset paths or incorrect cloud identifiers, ensuring that only legitimate, accessible files are processed.

The effectiveness of this stage is measured by its planning accuracy, the proportion of plans that select the correct datasets given the query and metadata. During testing with the synthetic datasets, planning accuracy remained high, with most failures linked to incomplete metadata rather than prompt misinterpretation. The inclusion of a Data Gap marker in later stages further demonstrates that even when planning selects the correct file, the system can still recognise missing information downstream.

## 4.3 Execution of Sub-Questions

Following the generation of the multi-step execution plan in 4.2, the orchestrator sequentially processes each sub-question against the designated dataset and cloud location. Each step gives an intermediate output "sub-answer", which is then carried forward to subsequent steps, enabling cumulative reasoning. Figures 24 & 25 illustrate the outputs generated for each stage of execution.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 24: Step-by-Step (1,2) Execution Results for LA Customer Query

Output displaying results for each step.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 25: Step-by-Step (3,4) Execution Results for LA Customer Query

Output displaying results for each step.

The first sub-question targeted customers.csv within the Amazon S3 environment, filtering entries to include only those located in Los Angeles. The system returned eight unique customer records, each with an associated Customer ID (e.g., C0001, C0007, C0011). These identifiers formed the primary key for downstream joins, constraining subsequent steps to operate solely on this subset of the customer base (Figure 24).

Using the Customer IDs from Step 1, the second sub-question queried sales.csv (S3) to identify corresponding vehicle sales. The result revealed only one confirmed transaction: Sale S0007, linked to VIN 0W10LBT3G2QZW3C1Q, associated with Customer C0023. The absence of additional matches for the other seven LA customers was explicitly noted, signalling sparse sales activity for this segment (Figure 24).

The third sub-question interrogated car\_inventory.csv (S3) to verify the VIN obtained in Step 2. No match was found, and the system generated a “data gap” output stating that the VIN was absent from current inventory records. This step demonstrates the system’s ability to surface and report missing links rather than omit them silently (Figure 25).

The final sub-question queried car\_models.csv (S3) to determine the most common body types purchased by LA customers. Due to the absence of confirmed VIN-to-model mappings from Step 3, the system defaulted to producing a general body type ranking across the dataset. SUV, Crossover, Truck, and Hatchback each appeared four times, producing a four-way tie for the top-ranked category (Figure 25).

The execution process demonstrated a clear dependency chain:

1. Primary Key Identification – Step 1’s Customer IDs enabled Step 2’s VIN search.

2. VIN Linking – Step 2’s VIN informed Step 3’s inventory check.

3. Model Mapping – Step 3’s results influenced Step 4’s aggregation scope.

The results also highlighted how early-stage data gaps can propagate downstream, reducing the specificity of later outputs. Nevertheless, the sequential design preserved traceability: every figure and statistic in later steps can be traced back to a specific record or the absence thereof in earlier stages.

## 4.4 Merging and Reasoning Across Results

The merging stage represents the final synthesis point in the orchestration pipeline, where discrete sub-answers from each execution step are combined into a unified response. This is handled by the merge\_step() logic, which takes the sequential outputs produced by the cloud-specific clients (§4.3) and compiles them into a coherent, end-user-facing narrative.

The process operates iteratively. After each sub-question executes, its result, whether a set of factual records, an aggregated statistic, or a data gap notice, is appended to a cumulative context. When the final step completes, this context contains the full trail of reasoning: the raw facts retrieved from each dataset, their inferred relationships, and any unresolved lookups.

### 4.4.1 Successful Merges

In cases where all required join keys were present in the intermediate results, merging was straightforward. For example, in the “Which sales rep logged the highest revenue per sale?” query, Figure 26, the system successfully:

1. Aggregated sales revenue by employee ID from sales.csv (S3).

2. Joined these IDs with employees.csv (S3) to obtain names.

3. Ranked the sales reps, producing a clean mapping of identifiers to revenue figures.

The final answer preserved source annotations ([s3]) and flagged missing employee details for one record, maintaining both completeness and transparency. A screenshot of a computer

AI-generated content may be incorrect.

Figure 26: Example of a Successful Merge Output for Sales Revenue Query

This figure shows the system’s output for the query “Which sales rep logged the highest revenue per sale?”.

Similarly, Figure 27, “Which supplier did Pinnacle Autos spend the most with in 2024?” merged Azure-sourced supplier cost data with dealership metadata from S3 without error, yielding an accurate, traceable answer.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 27: Example of a Successful Merge Output for Sales Revenue Query

This figure shows the system’s output for the query: “Which supplier did Pinnacle Autos spend the most with in 2024?”.

### 4.4.2 Partial or Failed Merges

When intermediate keys were missing, later stages still executed but returned reduced-value outputs. In “For customers based in LA, list the top three body types they bought”, the merge failed to link the single VIN found in sales.csv to any record in car\_inventory.csv, preventing downstream identification of model IDs or body types. The merge logic handled this gracefully by outputting a general body-type ranking from the available model dataset, annotated with a Data Gap notice explaining the missing linkage.

Figure 28 demonstrates this case, showing that although some facts (customer IDs and one sale record) were retrieved, the absence of VIN-to-model matches resulted in fallback output. The figure highlights how the system clearly documented gaps while still providing partial insight into likely customer preferences.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 28: Example of a Partial Merge Output for LA Customer Body-Type Query

This figure presents the system’s output for the query “For customers based in LA, list the top three body types they bought.”

A similar pattern occurred in “For financed sales only, what’s the average down-payment percentage by body type?”. VIN-to-model mappings existed, but they could not be matched to the specific financed sales in question, and the car\_models.csv lacked any down-payment data. The merge step preserved all retrieved partial facts but explicitly stated that a precise calculation was not possible.

### 4.4.3 Role of Data Gap Annotations

The inclusion of Data Gap markers is a design choice. Rather than returning incomplete answers silently, the system embeds these markers into the merged output, signalling exactly where and why the reasoning chain broke. This was evident in some of the cases and is a much needed section that gives a reason why the answer is not completed.

### 4.4.4 Reasoning Across Sources

The merging logic treats each step’s output as both a factual contribution and a potential filter for subsequent steps. When joins succeed, the reasoning chain narrows the scope with each iteration, producing targeted answers. When joins fail, the reasoning broadens, defaulting to general statistics while still carrying forward the original constraints for context. This dual behaviour ensures that the system can deliver something useful in the presence of incomplete data, while remaining honest about its limitations.

Overall, the merging stage was most effective when the intermediate keys (e.g., customer IDs, VINs, model IDs) existed in both the source and target datasets and when metadata accurately described dataset schemas, enabling correct join selection.

Failures typically resulted from synthetic data omissions a limitation of the evaluation setup rather than the orchestration logic itself. In real-world deployments with complete, consistently keyed datasets, these failure cases would be substantially reduced.

## 4.5 Examples of System Behaviours

To illustrate how the system performs under different conditions, this section presents three representative examples from the evaluation. These cases highlight how the orchestration pipeline adapts to complete, partial, and failed merges while maintaining transparency about the reliability of its outputs.

The first example involved the query “Which sales rep logged the highest revenue per sale?”, Figure 26. The system correctly retrieved sales totals from the sales.csv file on S3 and then joined them with employee details from employees.csv. This allowed it to map revenue values to specific individuals. The merged output identified the top three performers: E036 ($101,858), Morgan Miller ($92,616), and Alex Smith ($76,680). While the system lacked a name for E036, it explicitly flagged this as a Data Gap rather than omitting or fabricating the information. This example represents the system’s ideal behaviour: accurate dataset joins, correct attribution, and a clear ranking that supports actionable insights.

The second example considered the query “For customers based in LA, list the top three body types they bought.”, Figure 27. The planner first filtered eight customers located in Los Angeles from customers.csv. However, only one of these customers had a matching sales record in sales.csv, and the associated VIN could not be linked to inventory records in car\_inventory.csv. As a fallback, the system produced a general frequency count of body types from car\_models.csv, listing SUVs, Trucks, and Crossovers as common categories. Crucially, the system did not overstate the specificity of these results. Instead, it returned a qualified output and explained that the VIN-to-model linkage was missing. This shows the pipeline’s resilience: it still delivers useful context even when data joins are incomplete.

The third example demonstrates a full merge failure. The query “List VINs serviced in 2025 and show their original sale price”, Figure 29, produced complete service details from service\_records.csv in Azure, including the type, date, and cost of each record. However, attempts to link these VINs back to sales.csv in S3 failed, preventing the retrieval of original sale prices. The system preserved all verified service facts while clearly reporting that VIN matching was unsuccessful, thereby avoiding unsupported claims. This behaviour illustrates what can be termed a “merge collapse,” where later steps lose specificity because of missing keys. Even in this case, the system remained transparent by carrying forward partial context and stating why the original question could not be fully answered.

A close up of a document

AI-generated content may be incorrect.

Figure 29:Example of a Merge Output for VINs serviced Query

This figure presents the system’s output for the query “List VINs serviced in 2025 and show their original sale price.”

Taken together, these examples demonstrate how the system adapts to different data conditions. When join keys exist, it produces precise, actionable insights. When only partial matches are possible, it supplements the answer with broader statistics while signalling limitations. And when no match can be made, it still returns all valid facts while openly acknowledging gaps. This adaptive behaviour ensures that users receive the maximum available insight, always paired with clear indicators of scope and reliability.

## 4.6 Summary of Results

The evaluation demonstrated that the metadata-driven system, implemented through the Model Context Protocol (MCP), can translate natural language queries into structured, schema-aware execution plans, retrieving targeted data from both Amazon S3 and Azure Blob Storage and merging these results into coherent, traceable answers.

Table 1: Classification of Query Outcomes by Root Cause

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Outcome Type | Total Queries | Dataset Gap Issue | System/LLM Errors | Clean Success |
| Success | 5 | 0 | 0 | 5 |
| Partial | 4 | 3 | 1 | 0 |
| Failed | 4 | 4 | 2 | 0 |
| Total | 13 | 7 | 3 | 5 |

Table shows all the outcomes of the queries and the reasons why they failed

Across 13 evaluation queries I did, performance outcomes were categorised into successful, partial successful and failure, see Table 1. It shows that 5 queries were fully successful, achieving precise outputs where dataset relationships were complete and join keys were present. 4 queries were partially successful, where its needed to mention that early planning and execution steps worked but missing linking keys in the datasets prevented complete answer. Finally, 4 queries failed, with output blocked entirely due to data unavailability or system errors.

The breakdown highlights that dataset gaps were the most frequent case of failure, adding up to 7 cases. These issues come from it being synthetic dataset. In a real file scenario, this number could be down to 0. System or LLM reasoning errors occurred in 3 cases, typically in scenarios requiring simple logical comparisons or multi-step percentage calculations. While successful outcomes were only 5 and they were achieved in queries involving robustly connected metadata across S3 and Azure.

Importantly to say, these results indicate that the planning phase was reliable. Failures occurred during the execution and merging phases, where dataset gaps prevented key joins from being completed. Nevertheless, the system maintained transparency though Data Gap section, ensuring that the users were aware of where and why results could not be fully produced.

Overall, the system confirms that the MCP based orchestration is robust in logic but constrained by data availability. With more complete production datasets, the proportion of fully successful queries would be expected to increase substantially.

# Chapter 5: Conclusion and Future Work

This project set out to design and evaluate a metadata driven system built on the Model Context Protocol (MCP), capable of interpreting natural language questions, identifying relevant datasets in Amazon S3 and Azure Blob Storage, executing targeted queries and merging the results into coherent, user facing answers. The evaluation demonstrated that the system could plan queries accurately by grounding its reasoning in verified metadata, execute steps in sequence with clear interdependencies, and merge partial outputs into final responses that are both informative and transparent, generally avoiding hallucinated datasets through metadata grounding.

The use of synthetic datasets allowed for controlled experimentation across realistic business scenarios such as automotive sales, financing, vehicle servicing, and supplier transactions. While these datasets successfully represented the kinds of schemas and relationships expected in real-world environments, their artificial nature inevitably introduced gaps in coverage, such as missing VINs, incomplete join keys, or limited transaction records, which in turn limited the specificity of some final outputs. Rather than masking these limitations, the system consistently surfaced them through explicit Data Gap annotations, demonstrating a key design principle: the ability to communicate uncertainty and incompleteness in a clear, actionable way.

Evaluation was intentionally constrained to a single large language model Anthropic Claude 3.5 Haiku (2024-10-22) to ensure consistent behaviour and control operational costs. This model was chosen for its balance of reasoning capability and low latency, but the architecture can support alternatives such as larger Claude variants or OpenAI’s GPT models. The results therefore reflect the capabilities and constraints of this single model; broader model benchmarking was beyond the scope of this study but remains a logical extension for future research.

Overall, the work demonstrates that metadata-grounded orchestration can be a viable strategy for enabling large language models to reason over heterogeneous, distributed datasets. The system met its core objectives: it planned without hallucinating datasets, executed across multiple clouds with correct dataset targeting, and merged results into a final answer while maintaining transparency about limitations. Even when joins failed due to missing or inconsistent keys, the pipeline preserved all available context and provided fallback outputs, ensuring that the user always received the most relevant insight possible given the available data.

## 5.1 Future Work

Several areas of improvement have been identified that could enhance both the accuracy and utility of the system in real-world deployments. First and foremost, replacing the synthetic datasets with production-grade data would enable a more rigorous test of the pipeline’s capabilities. With complete and consistently keyed datasets, the proportion of fully resolved queries would be expected to increase substantially, reducing the prevalence of partial merges and Data Gap annotations.

Secondly, expanding evaluation to include multiple LLMs such as GPT-4o, Claude Opus, or domain-specific fine-tuned models, would provide a more comprehensive understanding of how different architectures handle metadata-grounded planning and multi-step execution. This could reveal strengths and weaknesses unique to each model, informing both model selection and prompt optimisation strategies.

Thirdly, exploring incremental metadata updates would improve efficiency by avoiding full file rescans on every query. Instead of rebuilding dataset descriptions and schemas from scratch, the system could detect when files have changed and update only the affected metadata entries, reducing latency and cloud API costs.

Finally, while the current system generally avoided hallucinated datasets by grounding planning in verified metadata, implementing additional safeguards, such as programmatic schema validation or explicit dataset whitelisting, could further minimise false positives and strengthen execution reliability.

## 5.2 Final Thoughts

This work demonstrates that Large Language Models, when grounded in accurate metadata and supported by a structured orchestration pipeline built on the Model Context Protocol (MCP), can effectively query and reason across distributed datasets stored in Amazon S3 and Azure Blob Storage. While the use of synthetic data and a single LLM imposes limitations on the generalisability of the results, the system nonetheless provides a strong proof of concept for a robust, transparent, and extensible approach to metadata-driven cloud data analysis. With richer datasets, broader model evaluation, and ongoing metadata refinement, this approach has the potential to form the foundation of a practical, production-grade tool for complex, cross-platform data reasoning in enterprise environments.

# References

An, Z., Zhou, K., Ji, H., Xu, Y., Xie, Y., Chen, W. and Lin, J. (2023) LEDD: A Large-scale English Dataset for Data-to-Text. arXiv preprint arXiv:2306.04850. Available at: <https://arxiv.org/abs/2306.04850> (Accessed: 5 July 2025).

Anthropic (2023) Introducing Claude. Anthropic Blog. Available at: <https://www.anthropic.com/news/introducing-claude> (Accessed: 6 July 2025).

Anthropic (2024) Claude 3.5 Model Card. Anthropic. Available at: <https://www.anthropic.com/claude/haiku> (Accessed: 2 August 2025).

Anthropic (2024) Model Context Protocol (MCP): Standard for connecting AI models to external resources. Available at: <https://modelcontextprotocol.io> (Accessed: 3 August 2025).

AWS (2023) Amazon QuickSight Q. Amazon Web Services. Available at: <https://aws.amazon.com/quicksight/q/> (Accessed: 8 July 2025).

AWS (2024) Boto3 Documentation. Amazon Web Services. Available at: <https://boto3.amazonaws.com/v1/documentation/api/latest/index.html> (Accessed: 8 July 2025).

Barr, J. (2024) Building data lakes and analytics on AWS. AWS News Blog. Available at: <https://aws.amazon.com/blogs/aws/> (Accessed: 9 July 2025).

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., … and Amodei, D. (2020) ‘Language Models are Few-Shot Learners’, Advances in Neural Information Processing Systems (NeurIPS), 33, pp. 1877–1901. Available at: <https://arxiv.org/abs/2005.14165> (Accessed: 9 July 2025).

Ghosh, P. (2023) Cloud Data Management and Governance. O’Reilly Media. (Accessed: 10 July 2025).

Google Cloud (2020) Introducing BigQuery Omni: Analyze data across clouds. Available at: <https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-omni> (Accessed: 12 July 2025).

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y.J., Madotto, A. and Fung, P. (2023) ‘Survey of Hallucination in Natural Language Generation’, ACM Computing Surveys (CSUR), 55(12), pp. 1–38. Available at: <https://dl.acm.org/doi/10.1145/3571730> (Accessed: 12 July 2025).

LangChain (2023) LangChain Documentation. Available at: <https://docs.langchain.com> (Accessed: 13 July 2025).

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., … and Zettlemoyer, L. (2020) Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv preprint arXiv:2005.11401. Available at: <https://arxiv.org/abs/2005.11401> (Accessed: 13 July 2025).

Meta AI (2023) LLaMA: Open and Efficient Foundation Language Models. Available at: <https://ai.meta.com/research/publications/llama-open-and-efficient-foundation-language-models/> (Accessed: 14 July 2025).

Microsoft (2023) Introducing Microsoft 365 Copilot. Microsoft Blog. Available at: <https://blogs.microsoft.com/blog/2023/03/16/introducing-microsoft-365-copilot/> (Accessed: 14 July 2025).

Microsoft (2024) Azure Blob Storage client library for Python. Microsoft Learn. Available at: <https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python> (Accessed: 15 July 2025).

Microsoft (2024) Power BI Q&A limitations. Microsoft Learn. Available at: <https://learn.microsoft.com/power-bi/natural-language/q-and-a-limitations> (Accessed: 15 July 2025).

OpenAI (2023) GPT-4 Technical Report. arXiv preprint arXiv:2303.08774. Available at: <https://arxiv.org/abs/2303.08774> (Accessed: 16 July 2025).

Rajkumar, R., Li, Z., Chen, J., Li, X., Chen, Y., Wang, Y. and Li, Y. (2022) Evaluating LLMs on Text-to-SQL: A Benchmark and Evaluation Framework. arXiv preprint arXiv:2206.01349. Available at: <https://arxiv.org/abs/2206.01349> (Accessed: 16 July 2025).

Tableau (2024) Ask Data in Tableau. Available at: <https://www.tableau.com/products/new-features/ask-data> (Accessed: 18 July 2025).

Trino (n.d.) Concepts – Trino Distributed SQL Query Engine. Available at: [https://trino.io/docs/current](https://trino.io/docs/currentl) (Accessed: 18 July 2025).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I. (2017) Attention is All You Need. Advances in Neural Information Processing Systems (NeurIPS), 30. Available at: <https://arxiv.org/abs/1706.03762> (Accessed: 19 July 2025).

Yao, S., Zhao, J., Yu, D., Gao, Y. and Liu, S. (2023) ReAct: Synergizing Reasoning and Acting in Language Models. arXiv preprint arXiv:2210.03629. Available at: <https://arxiv.org/abs/2210.03629> (Accessed: 19 July 2025).