Brought to you by:

CLOUDERA

AI Agents in Data Platforms

for dummies®

A Wiley Brand

Use an open lakehouse in an enterprise platform

Support AI agents within an enterprise data platform

Provide governance and isolation with AI agents

Cloudera® Special Edition

Dipankar Mazumdar

About Cloudera

Cloudera is the only data and AI platform company that large organizations trust to bring AI to their data anywhere it lives. Unlike other providers, Cloudera delivers a consistent cloud experience that converges public clouds, on-prem data centers, and the edge, leveraging a proven open-source foundation. As the pioneer in big data, Cloudera empowers businesses to apply AI and assert control over 100% of their data, in all forms, improving security, governance, and real-time and predictive insights. The world's largest brands across all industries rely on Cloudera to transform decision-making and ultimately boost bottom lines, safeguard against threats, and save lives. Learn more at cloudera.com.

AI Agents in Data Platforms

Cloudera® Special Edition

by Dipankar Mazumdar

for dummies®
A Wiley Brand

AI Agents in Data Platforms For Dummies®, Cloudera® Special Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

ISBN 978-1-394-46612-2 (pbk); ISBN 978-1-394-46613-9 (ebk); ISBN 978-1-394-46614-6 (ePub)

For general information on our other products and services, or how to create a custom *For Dummies* book for your business or organization, please contact our Business Development Department in the U.S. at 877-409-4177, contact info@dummies.biz, or visit www.wiley.com/go/custompub. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Development Editor/Project Manager: Rebecca Senninger

Acquisitions Editor: Traci Martin

Editorial Manager: Rev Mengle

Business Development Representative: Jeremith Coward

Content Refinement Specialist: Magesh Elangovan

Cloudera team: Valerie Glover

Table of Contents

Introduction

Most enterprise data platforms were built around deterministic workloads. Data engineers design pipelines, analysts write queries, and systems execute predefined workflows.

AI agents introduce a different execution model. Rather than following a fixed plan, agents *explore*. They discover datasets, inspect schemas, generate queries, evaluate intermediate results, and revise their actions based on what they learn. A single task may involve many exploratory interactions with a data platform before an agent arrives at a final answer.

This book makes the case that enterprise data platforms need to evolve to support autonomous agents interacting with enterprise data. Throughout the chapters that follow, we'll explore the architectural capabilities required to make that possible.

About This Book

This book explores how agents interact with data platforms, the capabilities they require, and why concepts such as isolation, context, governance, and metadata management become critical in agent-driven systems. It also examines how open lakehouse architectures and technologies such as Apache Iceberg provide many of these capabilities today, and how enterprise platforms can bring them together into governed and auditable agentic workflows. By the end of this book, you will have a framework for thinking about AI agents as a new class of workload within enterprise data platforms.

Icons Used in This Book

Throughout the book, you find icons to indicate special information. Here's a guide to what those icons mean.

TIP

The Tip icon indicates information that you can apply to your own projects to make them work better. Tips can save you time and help you avoid frustration.

REMEMBER

The Remember icon highlights information that's worth retaining after you put down this book.

Beyond This Book

This book introduces the foundational concepts required to support AI agents within enterprise data platforms. To continue exploring these topics, consider these resources:

- Supporting our AI Overlords Paper: Redesigning Data Systems to be Agent-First: `https://arxiv.org/abs/2509.00997`
- Can AI Agents Answer Your Data Questions? A Benchmark for Data Agents: `https://arxiv.org/pdf/2603.20576v1`
- Context and Isolation: The Key Primitives for AI Agents in Data Platforms: `https://www.cloudera.com/events/webinars/context-and-isolation.html`

IN THIS CHAPTER

- Exploring how agents interact with enterprise data systems
- Examining the shift from deterministic pipeline execution to exploration
- Understanding why existing architectures face new challenges

Chapter 1
Preparing Enterprise Data Platforms for Agents

Most existing data architectures and platforms have largely been built around deterministic and human-driven pipelines. Data is ingested, transformed, and served through well-defined workflows designed by engineers and executed by orchestration systems. These systems assume relatively controlled query patterns, predictable write paths, and carefully orchestrated updates through carefully designed pipelines.

However, the emergence of agentic AI workflows introduces a fundamentally different execution model. Unlike conventional scenarios where engineers carefully design pipelines and define how data should flow through a sequence of transformations, agents may not simply execute predefined logic. They explore the problem space — forming hypotheses, invoking tools dynamically, inspecting intermediate results, and iteratively refining their actions. Agents attempt multiple queries in your data lakehouse or operational database, experiment with alternative joins, test data transformations, and discard intermediate results before

producing a final output. This behavior introduces a new class of workload: data agents operating over enterprise data systems.

Supporting these workloads raises an important architectural question: Are modern data platforms actually designed to support autonomous actors interacting with data? To answer that question, you first need to understand how agents interact with data systems and why their behavior differs from the workloads that most enterprise data platforms were originally designed to support.

How Agentic Workloads Interact with Data Systems

Typical data workloads are shaped by intent that is largely predetermined. A pipeline runs a known transformation graph or a dashboard issues a small family of parameterized queries.

Agentic workloads behave differently because the "plan" is not fully known up front. An agent is effectively executing a tight reasoning loop where each step depends on what it just learned from the system — schema hints, intermediate results, error messages, cardinalities, or unexpected null patterns. The loop looks less like a single query and more like an interactive search process over the data and its metadata. Mechanically, when an agent is trying to answer a business question or produce a transformation, it often performs a sequence of micro-operations that are atypical for conventional ETL (extract, transform, load) or analytics patterns.

Metadata discovery as a first-class phase

Before writing a "real" query, the agent may enumerate tables from a catalog, inspect schemas, check partition layouts, and probe nullability. This goes beyond a single lookup. It's a repeated, incremental discovery. For an agent, understanding the available data is often part of the task itself.

Rapid iterative query refinement

Agents issue a chain of increasingly specific queries, each informed by the prior output. They might start with coarse filters and progressively tighten them as they learn distribution and semantics. Rather than executing a predefined analytical query, they continuously refine their understanding of the problem.

High-cardinality filters and point lookups

Agents frequently generate filters such as:

```
WHERE id IN (...)
```

with large lists, or they focus on narrow slices of data:

```
"Show me records for customer X, order Y, last
  seven days."
```

These queries are often used to validate hypotheses quickly before moving to the next reasoning step.

Correlated subqueries and validation queries

Agents ask many small questions instead of one large analytical question. For example:

- Does this join increase row counts?
- Are there duplicate records?
- Are dimension rows missing?
- Does this transformation preserve expected cardinality?

These often appear as `EXISTS` checks, anti-joins, or constraint-style probes. The objective is not necessarily to produce business insights. The objective is to validate assumptions and guide reasoning.

Speculative branching

When uncertain, agents may pursue multiple plausible paths in parallel. They may evaluate alternative join graphs, different

grouping keys, different time windows, or different interpretations of a business concept. Many of these attempts may ultimately be discarded. Unlike traditional workflows where a single path is chosen ahead of time, exploration itself becomes part of execution.

Error-driven exploration

Parser errors, missing columns, permission failures, or unexpected data types become signals. Agents adapt their strategy based on system feedback, and that feedback loop itself creates query churn. The next action is often determined by what happened during the previous one.

Changing the Execution Model Workflow

This shift has direct implications for how workloads interact with the data system. Conceptually, the execution model changes as well. In a traditional pipeline, execution follows a predetermined path:

pipeline → query → result

A data engineer defines the transformation graph, the system executes the query, and produces the result.

An agentic workflow is closer to an exploratory process:

discover → probe → revise → validate → branch → converge → commit

Instead of executing a fixed plan, the agent probes the system, inspects intermediate results, revises its hypothesis, and may explore multiple alternatives before converging on a final answer. The important distinction is that exploration becomes part of execution. Many of the interactions that occur during an agent workflow are not directly related to the final output. They exist because the agent must continuously evaluate and refine its understanding of the problem before deciding how to proceed.

REMEMBER

Unlike traditional pipelines where the workflow is known ahead of time, agentic workloads continuously adapt as new information becomes available. The execution plan emerges during runtime rather than being predefined.

The difference in the execution model becomes clearer when you understand how agents actually interact with the system. Let's take a detailed look at these patterns.

High fan-out exploration

Consider a simple objective: "Create a revenue table."

Even though the final objective is a single transformation, the agent may execute dozens or hundreds of exploratory queries while searching for the correct approach. An agentic workflow could look like this where the final deliverable is a single table, but the path to produce it involves substantial exploration:

1. **Query 1: Inspects the schema:**

```
DESCRIBE TABLE prod.sales.orders;
```

2. **Query 2: Checks the column distribution:**

```
SELECT min(total_amount), max(total_amount),
  count(*)
FROM prod.sales.orders;
```

3. **Query 3: Tests aggregation (metric sanity):**

```
SELECT region, sum(total_amount) AS revenue
FROM prod.sales.orders
GROUP BY region
LIMIT 20;
```

4. **Query 4: Tests the join:**

```
-- baseline
SELECT COUNT(*) AS orders FROM prod.
  sales.orders;

-- after join
```

```
SELECT COUNT(*) AS rows_after_join
FROM prod.sales.orders o
JOIN prod.sales.customers c
ON o.customer_id = c.customer_id;
```

5. **Query 5: Compares the two plans:**

```
EXPLAIN FORMATTED
SELECT region, SUM(total_amount)
FROM prod.sales.orders
GROUP BY region;

EXPLAIN FORMATTED
SELECT region, SUM(total_amount)
FROM prod.sales.orders
WHERE order_date >= date_sub(current_date(), 30)
GROUP BY region;
```

6. **Query 6: Validates the output:**

```
SELECT count(*), min(revenue), max(revenue)
FROM prod.analytics.revenue_by_region;
```

Correlated query patterns

Many of these queries are not independent. They often represent small variations of the same hypothesis:

```
SELECT sum(price) GROUP BY region
SELECT sum(price) GROUP BY region, product
SELECT sum(price) GROUP BY region WHERE
  date > '2024'
```

These queries repeatedly scan the same tables and compute similar aggregates while the agent refines its understanding of the data.

Feedback-driven reasoning loops

Many of these queries are not independent. They often represent small variations of the same hypothesis. An agent may repeatedly

scan the same datasets and compute similar aggregates while refining its understanding of the data:

> Agent runs query → Reads result → Decides next action → Runs next query

Traditional analytics workloads are often tolerant of slower feedback cycles because humans remain in the decision loop. Agentic workloads are different, as each result becomes an input into the next reasoning step. Because every step depends on the previous result, the responsiveness of the underlying data platform directly influences the agent's ability to reason effectively.

Mutations in agentic workloads

Agents do not only read data. They may also trigger mutations — creating derived tables, fixing data quality issues, or rewriting partitions. Sometimes this simply means executing an existing pipeline, such as launching a Spark job that ingests data from Kafka into an Iceberg table. For example:

```
User intent: "Fix duplicate customer records."
```

The agent's workflow looks like this:

1. Detects duplicates.
2. Proposes deduplication strategy.
3. Generates SQL queries.
4. Executes MERGE INTO table (`iceberg.sales.customers`).

The difference is that the decision to execute such operations can be made at runtime by the agent rather than being predefined and scheduled by engineers.

Why Existing Architectures Face New Challenges

By this point, a pattern should be clear: Agentic workloads behave very differently from the deterministic, human-driven workloads that most enterprise data platforms were designed to support.

Agents explore, iterate, revisit assumptions, and evaluate competing hypotheses. As a result, supporting agentic workloads requires more than simply attaching a large language model (LLM) to an existing data platform.

Organizations need environments where agents can experiment safely, access the context required for effective reasoning, and operate within appropriate governance controls. They also need the ability to audit and understand agent actions in the same way they audit human-driven workloads. These requirements ultimately lead to a small set of architectural capabilities that become essential for agent-ready data platforms.

IN THIS CHAPTER

- Understanding why agentic systems require isolation
- Exploring context and memory horizons
- Examining governance for autonomous data access
- Understanding auditability and deterministic replay

Chapter 2
Building the Foundation for Agentic Data Systems

If agents are going to become first-class actors inside enterprise data platforms, the systems they interact with must support a very different execution model from traditional pipelines. Two primitives are necessary to support this behavior:

- Isolation primitives that allow agents to safely experiment with data without corrupting production state
- Context and memory horizons that provide the informational substrate required for reasoning over enterprise data

In this chapter, we examine these foundational elements in detail and explore why they become critical for enterprise AI agents.

Using Isolation Primitives

One of the defining characteristics of agentic workloads is that they are inherently exploratory. An agent attempting to answer a question or produce a data transformation may generate several candidate solutions before selecting one. Instead of a single

predefined execution path, agents may explore several "what-if" hypotheses in parallel — for example, trying alternative join graphs, grouping keys, filters, or time windows before committing to a final transformation. Many of these exploratory attempts may need to be rolled back or abandoned.

Without isolation, speculative operations can easily impact canonical datasets. Consider a simple example. An agent tasked with producing a cleaned dataset may attempt multiple transformations:

```
agent task
 ├─ attempt A: join with customer dimension
 ├─ attempt B: filter inactive users
 └─ attempt C: aggregate transactions
```

If each attempt directly mutates the underlying table, the dataset could be left in an inconsistent state whenever a speculative attempt fails.

Copy-on write semantics

Enterprise data platforms therefore require a mechanism that allows agents to experiment with state without immediately committing those changes to production. One common way to handle this across these domains is versioned state with copy-on-write semantics.

REMEMBER

This requirement is not unique to modern data systems. Similar patterns appear in version control systems, distributed databases, and modern storage engines.

This pattern typically manifests through three related mechanisms: snapshot-based table states, branching or forked lineage, and immutable underlying storage. These mechanisms allow systems to maintain multiple isolated views of the same dataset simultaneously.

Figure 2-1 shows what an isolation environment looks like.

Each branch represents an isolated experimental environment in which the agent can perform reads and writes without affecting the canonical dataset. Failed branches are discarded, while validated outcomes are merged into the primary lineage.

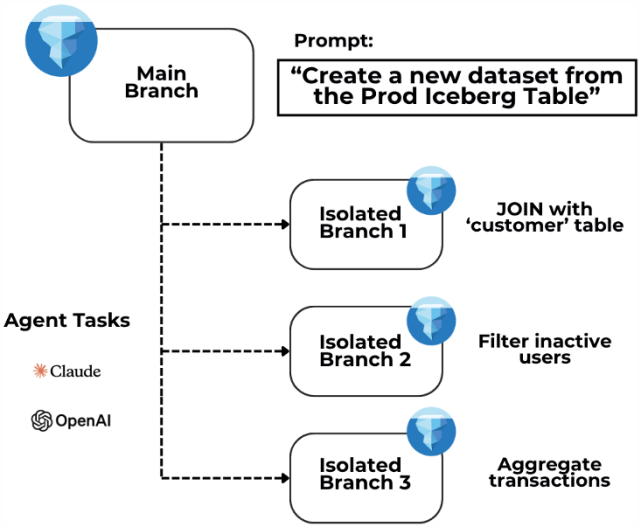

FIGURE 2-1: An isolation environment allows an agent to explore before committing to an action.

Rather than treating database mutations as a linear sequence of transactions, agent workflows begin to resemble branching search processes over data state, where each hypothesis operates on its own isolated snapshot. Supporting this pattern requires fast, lightweight forks that avoid physically duplicating data.

REMEMBER

Agentic workloads are fundamentally exploratory. Isolation allows agents to test multiple hypotheses without risking unintended changes to production datasets.

LEARN MORE ABOUT ISOLATION

Isolation in data and software systems is an actively explored problem. Here are some related works happening in this space.

The branching model closely resembles approaches being discussed in recent research on agent-first data systems, where speculative execution paths are isolated through forked database states. In these systems, agents frequently explore multiple alternative execution paths and therefore require mechanisms for efficiently creating and

(continued)

(continued)

discarding versions of data state. You can read more about it at https://arxiv.org/pdf/2509.00997.

Neon's serverless Postgres architecture (https://neon.com/use-cases/ai-agents#building-checkpoints-with-snapshots-and-branching) exposes database branching as a first-class capability built on copy-on-write storage. A database can be forked from a specific snapshot almost instantly, allowing isolated environments to share underlying storage while only materializing blocks that diverge.

These systems rely on the same principle: versioned state combined with copy-on-write semantics allows environments to be forked cheaply and safely.

Apache Iceberg and open table formats

Open table formats such as Apache Iceberg already provide several primitives that align naturally with these requirements. Iceberg organizes table state through a graph of immutable *snapshots*, which references a set of manifest files describing the data files that comprise the table. When a write occurs, Iceberg does not mutate existing files. Instead, it produces a new snapshot describing the updated table state.

Because data files remain immutable, new snapshots can share most of their underlying storage with previous versions. This makes the creation of new table states relatively inexpensive. Iceberg further extends this model through *zero-copy branching*, allowing multiple snapshot lineages to evolve independently.

A branch effectively creates a fork in table history:

```
main
 |
 ├─ agent-experiment-1
 |      ↑
 |   snapshot_a
 |
 └─ agent-experiment-2
        ↑
     snapshot_b
```

Each branch can evolve independently, enabling isolated experimentation. Only when a branch is explicitly promoted does its state affect the primary lineage.

From the perspective of agentic workflows, this model provides a natural execution sandbox. Agents can explore alternative transformations, generate derived datasets, or test data quality fixes without mutating production tables. Failed attempts simply produce short-lived branches that are discarded without side effects. This distinction becomes increasingly important as agents move beyond simple retrieval tasks and begin actively interacting with enterprise data systems.

The ability to create isolated working environments, experiment safely, and discard unsuccessful outcomes is not merely a convenience. It becomes a foundational requirement for allowing autonomous systems to operate against production data without introducing unacceptable risk.

Reasoning with Context and Memory Horizons

If isolation protects production data from speculative agent actions, context determines whether the agent can reason effectively in the first place.

LLMs do not operate directly on enterprise datasets. They operate on *context windows* — a bounded working set of information available at inference time. Everything an agent does, whether query generation, tool selection, reasoning steps, or validation, depends on the quality and structure of the context that is available to it.

This is why LLM providers increasingly view *context engineering* as a core design problem for agent systems. The challenge is not simply giving agents access to enterprise data, but structuring that access so that relevant information can be retrieved, interpreted, and incorporated into the reasoning loop efficiently. Enterprise data platforms contain vast amounts of information ranging from historical datasets, logs, documents, and metadata to derived tables. An agent cannot load this information directly into a prompt. Instead, it must retrieve and assemble the relevant

subset of context dynamically. In practice, agentic systems operate across multiple memory horizons, each with different latency, durability, and access characteristics.

Agent memory is divided into hot, warm, and cold contexts. These layers correspond closely to existing components in modern data architectures.

REMEMBER

The challenge is not providing agents with more data. The challenge is providing the right data, at the right time, in the right form.

Hot context (immediate execution state)

Hot context represents the active working state of the agent during reasoning and execution. This information is immediately available to the LLM during a step of inference and typically resides in memory or within the model's context window.

Typical elements of hot context include:

- The current prompt
- Recent conversation history
- Tool outputs from the current reasoning step
- Retrieved documents from a RAG (retrieval-augmented generation) pipeline

This state changes rapidly as the agent progresses through its reasoning loop and is typically ephemeral. Hot context therefore is defined by extremely low latency, ephemeral lifetime, and directly injected into the model's context window.

Because of these constraints, hot context is rarely persisted in large-scale storage systems. Instead, it's typically managed within agent runtimes or orchestration layers. Frameworks such as LangChain, CrewAI and various internal agent orchestrators manage this working state within the agent runtime, maintaining structured memory of prior steps, tool outputs, and retrieved context that are selectively assembled into the model's context window during each reasoning step.

However, hot context is not sourced from a single system. It's dynamically assembled at runtime by combining user inputs, prior reasoning steps, tool outputs, and information retrieved from external systems. For example, a retrieval step may query a vector database to obtain relevant records, which are then assembled into the prompt as contextual input. In this sense, hot context acts as the execution surface where retrieved knowledge, intermediate results, and instructions are brought together and actively consumed by the agent during reasoning.

When agents are integrated into data pipelines, hot context becomes the execution state passed between nodes in the workflow. Each step in the agent's reasoning loop (for example, retrieval, transformation, validation, or tool invocation) produces outputs that are incorporated into the hot context and made available to the next step. This context enables agents to propagate knowledge and decisions across the workflow while remaining lightweight and ephemeral.


TIP

A common misconception is that all agent memory lives inside the prompt. In reality, the prompt is often just the final assembly point where information from multiple memory layers is combined for reasoning.

Warm context (session and episodic memory)

Warm context represents persisted information from the agent's recent activity. This data must survive across multiple reasoning steps or workflow stages but does not need to be retained indefinitely.

It includes artifacts such as session memory, cached outputs from prior reasoning steps, task progress state for long-running workflows, and knowledge derived earlier in the interaction session.

For example, an agent performing exploratory analysis may discover schema relationships, statistical summaries, or anomalies that it wants to reuse during later reasoning steps.


TIP

Unlike hot context, warm context is typically persisted across steps or sessions, enabling agents to resume workflows, replay prior decisions, or retrieve previously derived insights.

From a systems perspective, warm context requires storage systems that provide moderate latency, frequent read/write access, and structured retrieval capabilities.

Several existing systems naturally fill this role. Vector databases are frequently used to store embeddings representing documents or observations encountered during an agent session. These systems participate in retrieval but are not directly part of the model's context window. Instead, they serve as the source from which relevant information is fetched into hot context during execution. Retrieval from these systems allows agents to recall relevant past interactions without loading entire histories into the prompt.

Operational data stores such as Redis or lightweight databases like SQLite are also commonly used to persist task state, workflow checkpoints, and intermediate outputs. Some platforms persist agent interaction logs into structured tables so that agents can revisit prior steps or perform replay analysis. Warm context therefore functions as the agent's short-term memory, bridging the gap between immediate reasoning and long-term knowledge stores.

Warm context often functions as the persisted workflow state between execution steps when agents are embedded into data pipelines. For example, an agentic pipeline may store intermediate analysis results, partial transformations, or extracted insights so that subsequent tasks in the workflow can reuse them without recomputing earlier steps. In this sense, warm context plays a role similar to checkpoint data in traditional data pipelines, enabling agents to resume workflows, branch decisions, or revisit prior results.

REMEMBER

Hot context represents what the agent sees right now. Warm context represents what the agent has learned recently.

Cold context (long-term knowledge)

Cold context represents the durable knowledge layer that agents can query but rarely load entirely. Examples include historical datasets stored in data lakes and lakehouses, curated warehouse tables, enterprise knowledge bases, document repositories, historical logs, and training corpora. These datasets may span

terabytes or petabytes and typically represent the canonical knowledge of the organization.

Cold context has several defining characteristics that include large-scale storage, higher query latency, and access through retrieval or analytical queries.

Modern data architectures already provide robust infrastructure for this layer. Lakehouse platforms built on open table formats such as Apache Iceberg serve as durable storage systems for large-scale enterprise datasets. These systems maintain versioned datasets through snapshot-based design, enabling consistent reads and efficient query planning while storing data in object storage systems such as Amazon S3 or Apache Ozone, and exposing them through query engines such as Spark, Trino, and Hive.

From the perspective of agentic systems, these lakehouse datasets effectively act as organizational memory. Agents can issue queries against these tables to retrieve relevant data slices, compute aggregates, or inspect historical patterns.

TIP

Because these systems store data in open table and file formats, they can be accessed by multiple compute engines and retrieval systems, enabling flexible integration into agent pipelines.

Beyond large-scale datasets, cold context also encompasses structured knowledge systems such as enterprise knowledge graphs, document repositories, and curated analytical models that encode relationships, semantics, and higher-level abstractions over raw data. In lakehouse architectures, catalogs play a critical role in organizing and exposing this knowledge layer. Catalogs act as the discovery and interpretation layer for agents, exposing structured metadata such as schemas, data locations, lineage relationships, and access policies that allow agents to reason about what data exists, how it is related, and how it can be safely accessed.

Rather than interacting directly with raw storage paths or isolated datasets, agents rely on the catalog to resolve high-level intents into concrete data access patterns. This may include discovering relevant datasets, understanding dataset structure, understanding lineage relationships, and generating queries through standardized interfaces.

How all three work together

The catalog functions as the navigational index for cold context. It allows agentic systems to locate and interpret enterprise data before retrieval layers selectively extract relevant slices of information that are promoted into warm or hot context for reasoning.

Figure 2-2 shows what this hierarchy looks like.

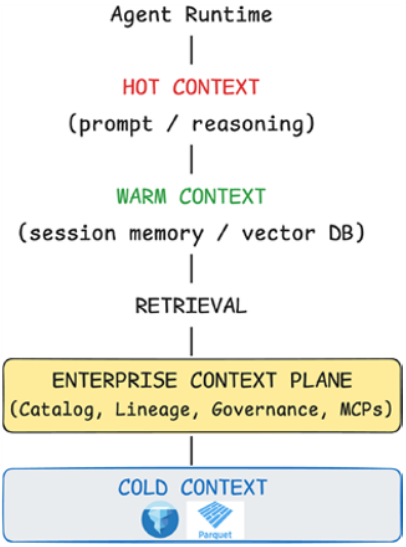

FIGURE 2-2: Agents start with hot context, then progress to warm and cold context for further decision making.

TIP

Not every piece of enterprise knowledge belongs inside the prompt. Effective agent systems retrieve information from cold context, preserve useful findings in warm context, and assemble only the relevant subset into hot context during reasoning.

How Open Lakehouse Architectures Naturally Fit the Context Layer

When discussing agent context and memory, it's tempting to think about context purely through the lens of prompts, vector databases, or retrieval systems. While these components play an

important role, they represent only a small part of the broader information landscape that agents must navigate in enterprise data platforms.

Enterprise organizations already maintain vast repositories of knowledge in:

- Historical datasets that capture years of business activity
- Curated warehouse tables that encode business definitions and transformations
- Knowledge bases that contain operational procedures and institutional knowledge
- Catalogs that describe schemas, ownership, lineage, and access policies

These systems represent the accumulated memory of the organization. So, the challenge is not necessarily building entirely new memory systems for agents, but enabling agents to effectively navigate and retrieve information from the systems that already exist.

This is where modern open lakehouse architecture plays a huge role. Open table formats such as Apache Iceberg already provide large-scale storage systems capable of maintaining historical datasets, derived analytical models, and operational data. Through snapshot-based design, these systems maintain durable and versioned representations of enterprise information, enabling agents to reason over both current and historical states of data.

However, data alone is not sufficient. An agent cannot reason effectively if all it sees are tables and files. To answer questions, generate queries, or determine whether a dataset is relevant, the agent must also understand what data exists, how datasets relate to one another, who owns them, and how they can be accessed safely. This is where catalogs become equally important. Catalogs provide the discovery and interpretation layer for enterprise data. They expose structured metadata such as schemas, business logic, data locations, relationships, ownership information, and access policies. Rather than requiring agents to interact directly with storage systems, catalogs help translate high-level intent into concrete data access patterns.

An agent looking for customer transaction data, for example, does not need to know where the underlying files reside. Instead, it can rely on catalog metadata to discover the appropriate datasets, understand their structure, identify related tables, and determine whether access is permitted. In many ways, catalogs provide the semantic understanding that enables agents to navigate enterprise data landscapes.

REMEMBER

This distinction is important. Open table formats provide access to the data itself; catalogs provide the context needed to understand that data. Together, they create a structured knowledge environment that agents can use to navigate to make sense of enterprise data.

Ultimately, from an agent's perspective, Iceberg tables, catalogs, lineage systems, and metadata systems collectively form a durable knowledge layer that can be queried, interpreted, and incorporated into the reasoning process. Retrieval systems help agents locate relevant information, but the underlying knowledge — data, metadata, lineage, policies, and business context — already resides within the data platform.

System-Level Properties for Agentic Data Architectures

So far, I've focused on the mechanical primitives that agents require to operate effectively within enterprise data systems. Isolation enables agents to experiment safely, while context provides the informational substrate required for reasoning.

However, once agents begin interacting with enterprise data platforms at scale, another layer becomes equally important. When autonomous systems are allowed to explore datasets, generate queries, and even propose mutations, the data platform itself must provide a set of operational guarantees that ensure these interactions remain safe, explainable, and governable.

Governance and deterministic replay and audibility are important to make autonomous systems work, but they're not directly part of the agent's reasoning process. Instead, they establish the boundaries and controls that enable autonomous systems to operate safely within enterprise environments.

Governance

If we are heading toward a world where agents interact directly with enterprise data platforms, governance becomes a control mechanism for autonomous data access. Unlike human analysts who typically operate within known workflows, agents can programmatically discover datasets, generate queries, and propose transformations at machine speed. Without strong governance controls, this autonomy could quickly lead to unintended access patterns or uncontrolled data mutations.

In lakehouse architectures, governance is typically enforced through the catalog layer, which acts as the policy authority for datasets rather than merely an inventory of data assets. While catalogs enable discovery, their more important role in agentic environments is enforcing structured access boundaries around enterprise data.

Catalog-managed table systems allow organizations to apply governance controls such as:

- Schema enforcement and evolution policies
- Fine-grained access controls and role-based permissions
- Table-level and column-level data policies
- Lineage tracking across transformations

For agentic workflows, these capabilities act as guardrails around autonomous execution. Agents may explore datasets or generate analytical queries, but the catalog ensures that access policies, schema constraints, and lineage rules remain intact. Any transformation or dataset publication must still pass through the same governance layer that applies to human-driven workloads. This separation between exploration and authorization becomes critical in agentic data architectures.

REMEMBER

Autonomy does not eliminate governance requirements. It increases them.

The more freedom agents have to explore and interact with enterprise data, the more important it becomes to enforce consistent policies around access, modification, and publication.

Deterministic replay and auditability

When autonomous agents generate queries, derive datasets, or modify table states, organizations must be able to reconstruct how those outcomes were produced. This requirement introduces the need for deterministic replay and full auditability of agent workflows.

Unlike traditional pipelines where transformation logic is explicitly defined in code, agent-generated workflows may involve exploratory reasoning, iterative query generation, and multiple speculative attempts before arriving at a final result. If the outcome of such a workflow affects production data or downstream analytics, enterprises must be able to answer several fundamental questions:

- What data did the agent read?
- Which version of the dataset was used?
- What queries or transformations were executed?
- Can the result be reproduced exactly?

Snapshot-based data architectures provide a strong foundation for addressing these requirements. In lakehouse systems such as Apache Iceberg, table states are represented as immutable snapshots connected through a metadata lineage graph. Each snapshot captures the precise set of data files that constituted the table at a given point in time. Because these snapshots remain accessible even as new updates occur, systems can support time-travel queries that reconstruct historical table states. This makes it possible to replay the exact data environment that an agent observed during a workflow.

Beyond replaying historical table states, Iceberg also exposes a rich set of metadata tables that make these workflows auditable and introspectable. Query engines such as Spark, Trino, and others can directly query these metadata tables to inspect how a table evolved over time and which operations produced each state.

For example:

- The history table records the sequence of snapshot commits for a table.

- The snapshots table provides detailed information about each snapshot, including operation type and summary statistics.
- Lower-level metadata structures such as manifests, files, entries, and partitions reveal exactly which data files were added or removed and how the physical layout of the table evolved.

For agent-driven workflows, this visibility can become part of the reasoning loop itself. Agents can query metadata tables to understand how a dataset has changed over time, identify recent updates or anomalies, or determine which partitions and files were modified by a particular operation. Instead of treating a table as an opaque dataset, the agent can inspect its operational history and structural metadata before deciding how to proceed. For example, an agent investigating a data quality issue could examine the history and snapshots tables to identify when a change occurred, inspect the associated manifests to determine which files were introduced, and then replay queries against earlier snapshots to validate a hypothesis.


TIP

One of the advantages of snapshot-based architectures is that they preserve not only data, but also historical context. This allows both humans and agents to reason about how a dataset evolved over time rather than only observing its current state.

IN THIS CHAPTER

- Using Iceberg branches to create isolated execution environments
- Discovering enterprise data through catalogs and lineage
- Governing agent access through centralized metadata services
- Connecting agents to enterprise data through MCP

Chapter 3
Implementing Agentic Primitives in the Open Lakehouse

While much of the discussion so far has focused on architectural primitives required for agentic workloads, a lakehouse architecture in its current form already offers several of these capabilities. This chapter shows how Cloudera Data and AI Platform built on an open lakehouse foundation incorporates isolation, context, governance, and auditability, as well as how it provides concrete mechanisms that map directly to these primitives and allows agent-driven workflows to operate within the data platform itself.

TIP

The open lakehouse architecture provides an environment where agents can safely explore data, understand its context, and operate within established governance boundaries. These capabilities allow agents to interact with enterprise data in a way that is informed, traceable, and aligned with organizational controls.

Providing Isolation with Cloudera Iceberg Branches

Cloudera's platform provides isolation through Iceberg branch-based isolation. Iceberg tables can be branched in Hive (Cloudera Data Warehouse [CDW]) or Spark (Cloudera Data Engineering), allowing multiple candidate remediation paths to be evaluated against the same production table state without mutating the main lineage. This gives a practical execution model for agent-driven workflows inside the lakehouse itself: Branch from a stable snapshot, test alternative fixes, validate the outcome, and only then fast-forward or promote the selected result.

Consider a production Iceberg table in Cloudera's lakehouse populated from Kafka events. Over time, the incoming records have developed drift: some `customer_id` values arrive as numeric strings, other values include prefixes or alphanumeric tokens; some `order_amount` values are null or malformed; and a few timestamps no longer conform to the canonical format. The agent is unsure which remediation strategy is safest.

One strategy may cast fields into the canonical schema, another may quarantine malformed records, and a third may implement strict filtering. Instead of applying one speculative fix directly on the production table, with Cloudera's Iceberg branching, the agent can fork the table state into multiple isolated branches and let each branch represent a different repair strategy.

Here is how branches can be created in CDW with the Hive compute engine:

```
ALTER TABLE default.kafka_orders_iceberg CREATE
  BRANCH agent_cast_strategy;

ALTER TABLE default.kafka_orders_iceberg CREATE
  BRANCH agent_quarantine_strategy;

ALTER TABLE default.kafka_orders_iceberg CREATE
  BRANCH agent_strict_filter_strategy;
```

At this point, all three branches begin from the same table state, but each can evolve independently.

```
main
 |
 ├─ agent_cast_strategy
 |      ↑
 |  snapshot_a
 |
 ├─ agent_quarantine_strategy
 |      ↑
 |  snapshot_b
 |
 ├─ agent_strict_filtering_strategy
 |      ↑
 |  snapshot_c
```

On one branch, the agent may attempt a canonical casting strategy by normalizing incoming fields into the expected schema. This is the least disruptive path when drift is mostly syntactic and the values are still recoverable into the target schema.

```
-- branch: agent_cast_strategy
CREATE TABLE default.orders_clean_cast
STORED BY ICEBERG AS
SELECT
  CAST(customer_id AS BIGINT)          AS
   customer_id,
  CAST(order_id AS STRING)             AS
   order_id,
  CAST(order_amount AS DECIMAL(18,2))  AS
   order_amount,
  CAST(order_ts AS TIMESTAMP)          AS
   order_ts,
  CAST(status AS STRING)               AS status,
  CAST(source_topic AS STRING)         AS
   source_topic,
  CAST(ingest_date AS DATE)            AS
   ingest_date,
  CAST(notes AS STRING)                AS notes
FROM default.kafka_orders_iceberg.branch_agent_
   cast_strategy
WHERE customer_id IS NOT NULL;
```

A second branch can implement a quarantine-first strategy. In this model, records that satisfy the expected contract continue into the curated table, while malformed rows are diverted into a quarantine dataset for later inspection.

```
CREATE TABLE default.orders_clean_quarantine
STORED BY ICEBERG AS
SELECT *
FROM default.kafka_orders_iceberg.
  branch_agent_quarantine_strategy
WHERE customer_id RLIKE '^[0-9]+$'
  AND order_amount RLIKE '^[0-9]+(\\.[0-9]+)?$'
  AND order_ts RLIKE '^[0-9]{4}-[0-9]{2}-
  [0-9]{2}.*';

CREATE TABLE default.kafka_orders_bad_records
STORED BY ICEBERG AS
SELECT *
FROM default.kafka_orders_iceberg.
  branch_agent_quarantine_strategy
WHERE NOT (
      customer_id RLIKE '^[0-9]+$'
  AND order_amount RLIKE '^[0-9]+(\\.[0-9]+)?$'
  AND order_ts RLIKE '^[0-9]{4}-[0-9]{2}-
  [0-9]{2}.*'
);
```

On a third branch, the agent may decide to apply stricter validation rules. This is useful when downstream correctness matters more than row retention and the objective is to preserve only records that fully satisfy the canonical schema contract.

```
CREATE TABLE default.orders_clean_strict
STORED BY ICEBERG AS
SELECT
  CAST(customer_id AS BIGINT)        AS
  customer_id,
  order_id,
  CAST(order_amount AS DECIMAL(18,2)) AS
  order_amount,
  CAST(order_ts AS TIMESTAMP)         AS order_ts,
  status,
```

```
  source_topic,
  ingest_date,
  notes
FROM default.kafka_orders_iceberg.
   branch_agent_strict_filter_strategy
WHERE customer_id RLIKE '^[0-9]+$'
  AND order_amount RLIKE '^[0-9]+(\\.[0-9]+)?$'
  AND order_ts RLIKE '^[0-9]{4}-[0-9]{2}-
   [0-9]{2}.*'
  AND status IN ('COMPLETED','PENDING','FAILED',
   'REFUND');
```

After executing the candidate remediation strategies, the agent can then validate the outcomes directly inside CDW using simple SQL checks. The following query compares the resulting row counts across the production table and the three branch-derived outputs.

REMEMBER

The production table remains immutable throughout the process. Candidate outcomes are evaluated independently before selecting the final result. Iceberg branching provides the isolation primitive that enables agents to safely explore multiple remediation strategies over the same dataset state.

Effective Context and Governance with Cloudera Data Catalog, Data Lineage, and Shared Data Experience

Before deciding how to query or transform data, an agent must be able to understand the structure of the data environment it operates within. This includes discovering available datasets, identifying their relationships, understanding how they were produced, and operating within governance boundaries.

In other words, agents need access not only to data itself, but also to the metadata graph that describes the data real estate. In the Cloudera Data Platform, these capabilities are provided through Cloudera Data Catalog, Cloudera Data Lineage, and Cloudera Shared Data Experience (SDX). Together, these services expose a governed metadata plane that enables agents to discover datasets, understand their dependencies, and access them safely.

Discovering assets through Cloudera Data Catalog

The first step for an agent exploring a data environment is asset discovery. Cloudera Data Catalog indexes datasets across the lakehouse and exposes them through a searchable metadata interface. Iceberg tables registered in the platform become immediately visible to both human users and automated systems.

As shown in Figure 3-1, the catalog exposes several Iceberg tables created during transformation experiments. For each dataset, the catalog provides metadata such as dataset ownership, creation timestamps, and storage format.

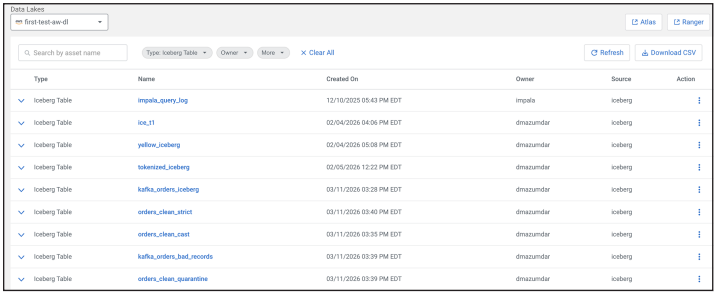

FIGURE 3-1: Catalogs provide key metadata for each dataset.

From an agent's perspective, the lakehouse transforms from a collection of files into a structured asset graph of data. Instead of blindly scanning storage locations, the agent can see what datasets exist and which ones may be relevant to its task.

Understanding data relationships through Cloudera Data Lineage

Discovering datasets is only the first step. Agents must also understand how those datasets are related. Cloudera Data Catalog integrates directly with Data Lineage, allowing users and automated systems to visualize the relationships between datasets and the processes that created them.

Figure 3-2 shows how the original dataset `kafka_orders_iceberg` produces several derived datasets (for three isolated branches) representing different transformation strategies.

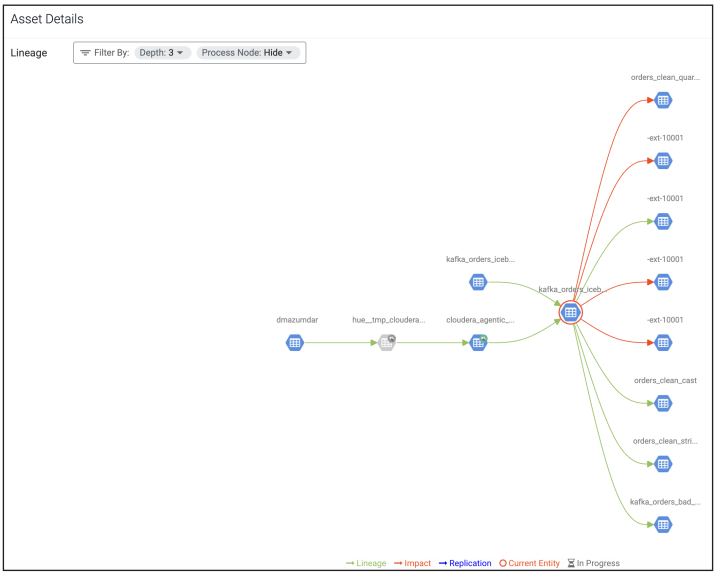

FIGURE 3-2: Three isolated branches for three potential strategies.

The lineage graph shows how these tables were produced by a transformation step applied to the original dataset and this relationship information becomes part of the decision context for the agent. Instead of treating each experimental dataset independently, the agent can now understand the full transformation graph (for example, which strategies were attempted).

Lineage provides visibility into the experimental graph produced by the agent, enabling it to reason about the outcomes of strategies before committing changes to production datasets.

TIP

Catalogs tell agents what data exists. Lineage helps them understand how that data came to exist.

Enforcing governance and policies with Shared Data Experience

Discovery and lineage provide critical context, but agents must also operate within enterprise governance boundaries. This is where Cloudera SDX becomes essential. It provides the unified governance layer for the entire lakehouse and catalog. Security policies, metadata classifications, and access controls are defined once and enforced consistently across all compute engines and services interacting with the data. For agent-driven workflows, this ensures that automated systems operate under the same governance constraints as human users and production applications.

Because SDX operates beneath the compute layer, policies apply equally to SQL engines, data engineering pipelines, and AI applications. The lakehouse is then secure by design for agent workflows.

Standardizing Access via MCPs and Tools

Agents need a mechanism to query datasets, retrieve metadata signals, and interact with enterprise data systems in a consistent way. So, they require a more standardized interface designed specifically for machine-driven interactions with external systems. This is where Model Context Protocol (MCP) comes in.

MCP has emerged as an open standard for connecting AI applications to external systems. Instead of embedding custom connectors for every data system, agents can communicate through MCP servers that expose specific capabilities such as querying databases, retrieving metadata, or interacting with APIs. Cloudera provides this capability through Cloudera MCP servers, which act as a standardized bridge between AI agents and enterprise data systems.

Cloudera's initial MCP server implementation (`https://github.com/cloudera/iceberg-mcp-server/tree/main`) exposes Apache Iceberg tables through Impala, enabling agents to query lakehouse datasets while still respecting SDX governance policies.

Through this interface, an agent built in Cloudera Agent Studio (part of Cloudera AI) can query Iceberg tables directly.

IN THIS CHAPTER

- Understanding the role of the execution layer in agentic systems
- How agents reason, iterate, and interact with enterprise data platforms

Chapter 4
Bringing AI Agents to Life with the Cloudera Data and AI Platform

Building an agentic system requires a runtime environment where agents can plan, execute, collaborate, and interact with enterprise systems in a controlled manner. Modern lakehouse architectures (including Cloudera Data and AI Platform) already provide many of the foundational primitives required for agentic workloads.

The final piece is an execution layer that can compose these components into executable workflows where agents can reason, iterate, and interact with enterprise data systems in a controlled and observable way.

This is where Cloudera Agent Studio (part of Cloudera AI) comes into the picture. It acts as the orchestration and execution layer that sits on top of the lakehouse foundation, enabling developers to build, run, and manage agent-driven workflows that use these underlying primitives.

Let's take a look into what Agent Studio brings.

Planning and Executing Iteratively

Agentic workloads are inherently exploratory. Agents do not execute a fixed query plan, but iteratively discover datasets, probe schemas, validate assumptions, and refine their approach. Agent Studio's orchestration layer treats this exploration as part of execution. It decomposes high-level objectives into multi-step plans, executes them iteratively, and evaluates intermediate results before committing to a path.

discover → probe → revise → validate → converge

Each step in this loop maps to concrete interactions with the underlying data platform. As the agent iterates, it translates its reasoning into actions over the lakehouse. For example, it:

- Queries datasets through MCP-enabled interfaces.
- Inspects schemas and relationships via the catalog and lineage systems.
- Validates transformations on isolated data states before committing changes.

Rather than treating planning and execution as separate tasks, Agent Studio enables agents to continuously move between reasoning and action, adjusting their approach as new information becomes available.

REMEMBER

Agentic workflows are rarely linear. The execution plan often emerges during runtime as the agent learns more about the environment it is operating within.

Coordinating Multi-Agent Execution

Enterprise data workflows often span multiple things including data discovery, transformation, validation, and governance checks, each requiring different reasoning strategies and tools. Agent Studio enables this through a multi-agent architecture where specialized agents collaborate under a coordinated orchestration layer. Each agent produces structured outputs that

become inputs for subsequent steps, making the reasoning process explicit and traceable. This is important when operating over systems like a lakehouse.

Instead of relying on a single monolithic agent to perform every task, responsibilities can be distributed across specialized agents that collaborate toward a shared objective. For example, one agent could explore datasets using the catalog, another could validate transformations on isolated branches, and another could enforce governance constraints or validate outputs at the same time.

Because all interactions are structured and persisted, this model aligns with the auditability and replay requirements discussed earlier. The full chain of reasoning, including which datasets were accessed, which transformations were applied, and which decisions were made becomes inspectable.

Managing Context

As discussed in Chapter 2, context is not just about access to data. It is about delivering the right subset of information to the agent at the right time. Agent Studio treats context as a first-class system concern. Instead of passing large volumes of raw data to the model, it constructs task-specific context dynamically during execution.

For example, an agent may:

- Retrieve relevant datasets through MCP-based interfaces.
- Use metadata from the catalog and lineage systems to guide reasoning.
- Incorporate intermediate artifacts generated during execution.

This approach ensures that agents operate with precise, structured inputs rather than unbounded data. By constructing context dynamically, Agent Studio helps improve both the efficiency and reliability of multi-step workflows.

TIP

More context is not necessarily better context. Effective agent systems focus on delivering relevant information rather than maximum information.

Executing in an Isolated and Controlled Environment

As agents become capable of generating code, executing queries, and interacting directly with enterprise systems, isolation becomes important at multiple layers of the architecture. Agent Studio enforces isolation at the execution layer through sandboxed runtimes.

The isolation Agent Studio provides at the execution layer complements the isolation mechanisms already present at the data layer through Iceberg branching within the lakehouse.

Together, this creates a multi-layered isolation model:

- Data isolation through Iceberg snapshots and branches
- Execution isolation through sandboxed runtimes in Agent Studio
- Access control through Shared Data Experience (SDX) governance policies

This layered approach ensures that agent-driven workflows can safely interact with enterprise systems without introducing unintended side effects.

REMEMBER

Isolation isn't provided by a single component. It emerges from multiple layers working together across data, execution, and governance boundaries.

Performing a Data Quality Investigation: An Example Implementation

To see how the concepts discussed until now come together in practice, consider an agentic data quality investigation over an Apache Iceberg table in an enterprise lakehouse. Rather than relying on a single general-purpose agent, the workflow is composed of multiple specialized agents coordinated through Cloudera Agent Studio, as shown in Figure 4-1. This architecture keeps responsibilities clearly separated and ensures that conclusions are grounded in evidence rather than model speculation.

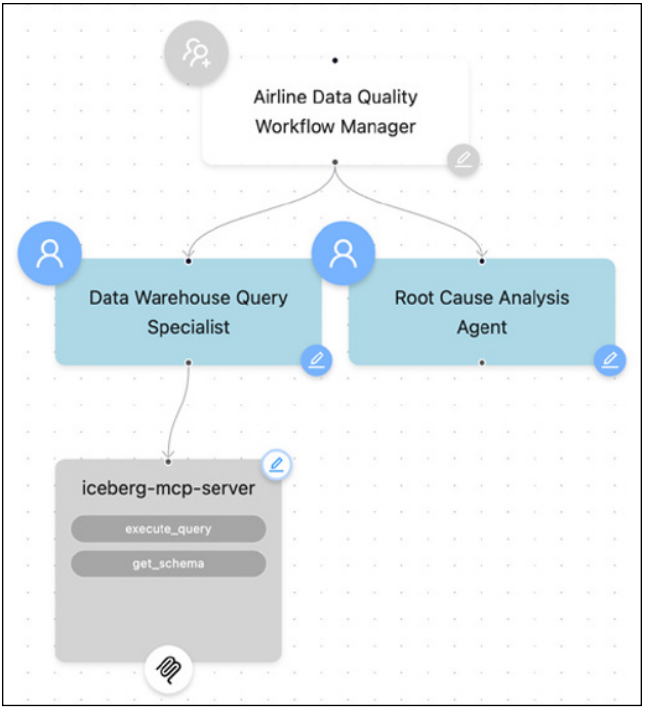

FIGURE 4-1: Cloudera Agent Studio brings together multiple specialized agents, with the Workflow Manager at the center.

Here's how Cloudera Agent Studio orchestrates the workflow:

1. A user submits a natural language request such as "Investigate data quality issues in `airlines.flights_iceberg` and produce a root cause analysis report based on the standard validation playbook."
2. The Workflow Manager interprets the request and delegates execution to the appropriate specialists rather than performing the investigation itself.
3. The Data Warehouse Query Agent, the first specialist, connects to the open lakehouse through an Iceberg MCP server, inspects schemas, executes read-only SQL, and returns factual results.


TIP

By restricting the agent to data retrieval and evidence collection, the system reduces the likelihood of unsupported conclusions and ensures that reasoning remains grounded in actual observations.

4. The Workflow Manager then passes the collected evidence to a second agent, Root Cause Analysis Agent.

   Unlike the query agent, this component does not generate SQL or interact directly with the underlying tables.

5. The Root Cause Analysis Agent synthesizes the returned evidence into a structured report that summarizes the validation checks performed, confirmed findings, severity, likely causes, and recommended next steps.

Throughout execution, the workflow relies on the same architectural primitives discussed throughout this book. Enterprise context is retrieved from the governed open lakehouse through standardized interfaces, orchestration is handled by the manager agent, and data access occurs through controlled MCP tools connected to Apache Iceberg. Importantly, the workflow operates within governance boundaries and executes standardized validation logic rather than allowing unconstrained interaction with production data. The result is a system in which autonomous workflows can safely interact with enterprise data while remaining explainable, auditable, and grounded in evidence.

IN THIS CHAPTER

- Ten important concepts
- Future changes in agentic systems

Chapter 5
More than Ten Key Takeaways

This book explores how the rise of AI agents is introducing a new class of workloads into enterprise data platforms. This chapter summarizes the key points to remember as you move forward and highlights several areas where the industry may continue to evolve.

- **Agents are not just another workload.** One of the easiest mistakes to make is to think of agentic workloads as simply another type of data workload. Traditional data workloads typically follow predefined execution paths, whether they are analytics queries, dashboards, reports, or data pipelines. Data agents gather information, evaluate options, revise their plans, and often change direction based on what they learn along the way. That creates a very different relationship with enterprise data systems. Instead of simply serving requests, the platform becomes part of an ongoing reasoning process. Supporting agents therefore requires more than APIs and query engines; it requires environments that allow exploration, iteration, and discovery.
- **Agentic workloads are inherently exploratory.** Most data workloads are relatively predictable. A query underneath a dashboard or a pipeline generally follows a known path from

start to finish. Agents rarely work that way. They may pursue several lines of inquiry at once, test assumptions, abandon approaches that do not work, and refine their strategy as new information becomes available. Exploration is not something that happens before execution — it is part of execution itself. Data platforms that support agents must be designed with that reality in mind.

- **Isolation is a foundational primitive.** Because agents are constantly experimenting and evaluating alternatives, they need safe places to work. Without isolation, exploratory actions can affect production systems or contaminate gold-layer datasets, which have a huge downstream impact. Capabilities such as snapshots, branching, immutable storage, and versioned state become critical because they enable agents to investigate possibilities without introducing risk.


TIP

The capability to separate experimentation from production is one of the most important requirements for agent-ready platforms.

- **Context matters more than data access.** Giving an agent access to more data does not necessarily make it more effective. What matters is whether the agent can access the right information at the right time. Context is what enables an agent to interpret data, connect ideas, and make informed decisions. Effective systems manage different layers of memory and knowledge, ensuring that relevant information is available when needed without overwhelming the reasoning process with unnecessary detail.
- **Catalogs are more than metadata repositories.** For human users, catalogs help people discover and understand data assets within an organization. For agents, they become even more important. A catalog helps answer questions about what data exists, how it is structured, who owns it, how it relates to other datasets, and whether it can be trusted. In many ways, the catalog becomes the map that enables agents to navigate the enterprise data landscape. Without that map, finding and interpreting information becomes significantly more difficult.
- **Open tables and catalogs create organizational memory.** Open table formats provide durable, versioned access to enterprise data, while catalogs provide the metadata, semantics, lineage, and governance needed to understand that data. Together, they begin to form something larger

than either component alone: a shared organizational memory. This combination gives agents a structured environment in which information can be discovered, interpreted, and connected across systems and teams.

- **Governance becomes more important as autonomy increases.** As agents gain the capability to discover datasets, generate queries, and potentially take actions on behalf of users, the scope and complexity of governance requirements expand. Governance becomes increasingly important because organizations need clear policies, controls, and safeguards to ensure that autonomous behavior remains aligned with business requirements, regulatory obligations, and security expectations.
- **Auditability must be built into the platform.** Trust depends on visibility, and organizations need to understand what an agent did, what information it accessed, what decisions it made, and whether those decisions can be reproduced.

  
  TIP

  These questions become especially important when agents are involved in business-critical processes. Auditability cannot be treated as an afterthought. It must be built into the platform from the beginning so that actions can be traced, reviewed, and understood when necessary.
- **Agent runtimes build on top of data platforms.** Agent frameworks operate alongside the underlying data platform. Storage, metadata management, governance, lineage, and operational controls remain essential components of the enterprise data stack. Agent runtimes rely on these capabilities to access information, understand context, and execute tasks effectively.

  
  REMEMBER

  This is one reason integrated platforms become increasingly important in agentic environments. Agents rarely interact with a single system in isolation. A single task may require discovering data through a catalog, retrieving context from multiple sources, validating permissions, tracing lineage, and accessing underlying datasets. When these capabilities are tightly integrated, agents can move more efficiently between reasoning and execution while maintaining governance and auditability.
- **The foundations already exist.** Perhaps the most encouraging observation is that many of the capabilities needed for agentic workloads already exist today. Modern open lakehouse architectures provide versioned data, rich

metadata, governance controls, and mechanisms for auditability and isolation. Open table formats strengthen this foundation by enabling open access to data through standardized interfaces, reducing vendor lock-in and enabling agents and applications to interact with data consistently across different tools and platforms. The challenge is learning how to expose, combine, and operationalize these existing capabilities in ways that enable agents to use them effectively.

REMEMBER

What matters most is bringing these capabilities together through an integrated platform that unifies data, metadata, governance, and execution, providing a cohesive foundation on which agents can be built, deployed, and scaled.

- **Future directions for agentic data systems.** The emergence of agentic workloads within data platforms is still in its early stages. Many of today's systems were originally designed for human-driven analytics, batch processing, and traditional application workloads. As agents become more deeply integrated into enterprise environments, data platforms will likely continue to evolve.

REMEMBER

One area that may receive increased attention is speculative execution management. Agentic workloads can produce extremely high rates of speculative attempts, potentially creating contention in commit protocols. Managing this speculative concurrency may require new orchestration mechanisms capable of coordinating large numbers of short-lived experimental branches while preserving snapshot consistency.

Future data systems may therefore need to incorporate capabilities such as:

- Branch lifecycle management for large volumes of ephemeral experimental states
- Speculation-aware scheduling to coordinate concurrent agent execution
- Shared intermediate computation across speculative runs to avoid redundant scans
- Context indexing and metadata-aware retrieval for efficient data discovery
- Tighter integration between catalog governance and agent orchestration layers

How an AI agent operates in a data platform

Unlike traditional approaches, AI agents have a different execution model. They explore, discovering datasets, inspecting schemas, generating queries, evaluating intermediate results, and revising their actions based on what they learn. Supporting this kind of workflow requires data platforms to be designed to support autonomous actors. This book shows how Apache Iceberg, and open lakehouse architecture, provides many of these capabilities, and how Cloudera's Data & AI Platform handles these newer workloads, and brings them together into governed and auditable agentic workflows.

Inside...

- How agents interact with data platforms
- Create isolated environments
- Store data into hot, warm, and cold contexts
- Governance for AI agents
- Apache Iceberg primitives for agents
- Build and deploy with Cloudera Agent Studio
- Best practices for AI agents

CLOUDERA

Dipankar Mazumdar is the Director of Developer Relations at Cloudera where he leads worldwide developer strategy. He has spent his career at the intersection of Data Engineering and AI, contributing to Apache Iceberg, and authoring books on lakehouse architectures and open data formats.

ISBN: 978-1-394-46612-2
Not For Resale

for dummies®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.