Snowflake Cortex Agents: A Practical Architecture Guide

When to use Agents, when to skip them, and how to build a cost-effective AI layer for financial data

---

In early 2025, while building a custom MCP server for multi-database natural language queries, an invitation to Snowflake's World Tour provided an opportunity to explore their semantic view architecture in a hands-on lab. Within weeks, discussions began at work about integrating Cortex Agents into an ELT pipeline for client-facing analytics. Cortex Agents enable sophisticated orchestration across structured databases and unstructured document sources which is a capability that would require significant custom development otherwise. The technical analysis that followed revealed important architectural constraints and cost trade-offs that aren't immediately obvious from Snowflake's documentation.

This guide covers those findings: when Agents add value versus when simpler approaches work better, the critical read-only limitation that shapes architecture decisions, and implementation patterns for financial data systems.

What Cortex Agents Actually Do

Cortex Agents utilize Cortex Analyst and Cortex Search as tools to plan tasks and generate responses. Cortex Analyst handles text-to-SQL for structured data (relational databases, your Snowflake environment). Cortex Search handles semi-structured and unstructured data, such as PDFs, markdown files, contracts, etc. stored in Snowflake Stages (cloud storage locations for loading/unloading data).

When a user asks a question as a prompt within an integrated chat context window. the Agent parses the request and decides which tools to invoke. This orchestration is where the intelligence lives and where the costs accumulate. The entire orchestration layer is charged based on tokens used.

The backbone of this system is Semantic Views and Models. Semantic Views translate how business users describe data into executable queries. They define business logic, relationships, filters, and metrics specific to your domain.

Key Concepts

Before diving into the architecture, it's important to clarify some terminology that can be confusing:

User-Defined Functions (UDFs) in Snowflake are reusable functions created via DDL (`CREATE FUNCTION`) that can be called by queries. They're schema objects that persist in the database. In this guide, UDF always refers to Snowflake User-Defined Functions, not user-defined fields or calculated columns.

Calculated columns/fields are expressions added to views or semantic models. When the article discusses "expression generation," it's creating these calculated columns—not UDFs. The distinction matters because Cortex Agents can call UDFs but cannot create them.

Key Terminology

Term	What It Is	When to Use
Cortex Agents	Orchestration layer that plans tasks, selects tools, and generates responses. Charged per token for orchestration.	Complex multi-tool queries; questions spanning structured AND unstructured data
Cortex Analyst	Text-to-SQL tool used BY Agents to query structured data. Requires semantic models/views. Charged per token.	Structured data queries (holdings, financials, compliance)
Cortex LLM Functions	Direct LLM calls without Agent orchestration. Simpler, cheaper. No planning or tool selection, just prompt in, response out.	Expression/formula generation; JSON structure generation; simple transformations
Semantic Views/Models	YAML-based definitions that translate business language to database schema. Contains expressions, filters, metrics, and verified queries.	Defining business logic for Cortex Analyst
UDFs	Snowflake schema objects created via DDL. Must be pre-built; Agents cannot create them.	Reusable, complex calculations registered as Agent tools

> **Key Insight:** For expression generation (creating calculated columns via natural language), use Cortex LLM Functions directly, not full Cortex Agents. Agents add orchestration overhead and cost that isn't needed for simple "generate JSON from this prompt" tasks. Reserve Agents for complex, multi-tool queries against structured and unstructured data.

What Can Users Actually Ask?

Here are concrete examples of what end users can ask and what happens behind the scenes:

- **"What are my current holdings by issuer?"** The Agent routes this to Cortex Analyst, which generates SQL against the semantic view, applies SCD Type 2 filters automatically, and returns aggregated results.
- **"Summarize the covenant terms in the Acme credit agreement"** The Agent routes this to Cortex Search, which locates the PDF in the Stage, extracts the relevant sections, and returns a summary.
- **"Compare our exposure to private companies against what their contracts allow"** The Agent uses *both* tools Analyst queries the holdings data, Search retrieves contract documents, and the Agent synthesizes the results.
- **"Create a field that calculates debt-to-EBITDA ratio"** This bypasses the Agent entirely. A direct LLM Function call returns JSON defining a calculated column expression. Your application layer validates the expression and adds it to the view definition. This creates a calculated column, not a UDF. The distinction matters because Agents cannot create UDFs (schema objects), but calculated columns are part of view definitions that your application manages.

The first three require Agent orchestration. The fourth doesn't, and that distinction is where cost savings live.

The Critical Limitation: Agents Cannot Create Objects

Agents are read-only. They cannot execute DDL. They cannot create views, UDFs, or any persisted object in Snowflake. They're essentially very smart users with limited permissions.

What do we mean by "objects"? In Snowflake (and databases generally), *objects* are schema-level entities created via Data Definition Language (DDL) statements. This includes:

- **VIEWs** saved queries that act as virtual tables
- **TABLEs** physical data storage
- **UDFs (User-Defined Functions)** custom functions you create with `CREATE FUNCTION`
- **Stored Procedures** executable code blocks
- **Stages** locations for loading/unloading data
- **Streams, Tasks, Sequences** other first-class Snowflake objects

Agents can *query* these objects, but they cannot *create, alter, or drop* them. Any `CREATE`, `ALTER`, or `DROP` statement is off-limits.

This has significant architectural implications. If you want to persist queries generated by Cortex Analyst, or allow users to create calculated columns through natural language, you need a separate application layer.

What Agents Can Do

- SELECT data via Cortex Analyst
- Search documents via Cortex Search
- Execute functions and procedures you've explicitly registered as custom tools
- Return JSON structures for your application layer to process

What Agents Cannot Do

- CREATE, ALTER, or DROP anything
- Persist objects to Snowflake
- Create UDFs dynamically
- Add columns to existing views

When to Use Agents vs. LLM Functions

This is the most important architectural decision. Getting it wrong means either paying for unnecessary orchestration or building complexity you don't need.

Use Cortex Agents When

- Complex natural language queries require SQL generation
- Questions need to search across structured AND unstructured data
- Multi-step reasoning is required ("What are the top issuers, and what do their contracts say?")
- You need planning, tool selection, and reflection

Use Cortex LLM Functions Directly When

- Generating expressions or formulas from natural language

- Creating JSON structures
- Simple transformations or translations
- Any task where you just need "prompt in structured response out"

Example: Expression Generation with SNOWFLAKE.CORTEX.COMPLETE

For an expression builder feature, the task is straightforward: take a user's natural language request and return a JSON structure. No orchestration, no tool selection, no multi-step planning needed.

```sql
SELECT SNOWFLAKE.CORTEX.COMPLETE(
    'llama3-8b',
    'You are an expression builder for a financial data platform.
Available columns from DIM_CREDIT:
- TOTAL_LEVERAGE: Total Leverage
- LTM_EBITDA: LTM EBITDA
- SENIOR_DEBT: Senior Debt
- EQUITY_SWEEP_PCT: Equity Sweep Percentage

User request: "multiply LTM EBITDA by Total Leverage"

Return ONLY valid JSON matching this exact schema:
{
  "name": "COLUMN_NAME_UPPERCASE",
  "displayName": "Human Readable Name",
  "expr": "SQL expression using column names above",
  "type": "NUMBER(28,10)"
}
Response:'
) AS expression_json;
```

This returns:

```
{
  "name": "EBITDA_X_LEVERAGE",
  "displayName": "EBITDA x Total Leverage",
  "expr": "LTM_EBITDA * TOTAL_LEVERAGE",
  "type": "NUMBER(28,10)"
}
```

This single LLM call costs a fraction of what Agent orchestration would cost for the same result.

Building Semantic Views for Financial Data

Semantic Views address the mismatch between how business users describe data and how it's stored in database schemas. They're generally categorized by potential questions a user would ask.

For example, a holdings classification semantic view would include the tables, relationships, filters, and metrics required to answer questions about holdings classifications—DimHoldings, DimIssuer, and the join logic between them.

The reliability challenge: LLMs generate SQL based on probabilistic inference. Without explicit schema constraints, the model may produce syntactically valid SQL that references nonexistent columns or incorrect join conditions, yielding plausible but wrong results. Semantic views solve this by providing a validated schema definition—the LLM generates queries only against explicitly defined tables, columns, and relationships.

To ensure consistent answers, semantic models provide libraries of reusable expressions, filters, and for complex questions, verified queries. Verified queries are predefined SQL statements that the Agent uses when a question matches a known pattern—guaranteeing correctness without requiring SQL generation.

Creating Semantic Views

Semantic views can be created directly in Snowflake, or deployed from YAML to a Stage. The YAML approach makes them reusable beyond Snowflake.

```
CALL SYSTEM$CREATE_SEMANTIC_VIEW_FROM_YAML(
    'ANALYTICS_DB.DWH',
    $$
name: holdings_analytics
description: Semantic view for portfolio holdings data
tables:
  - name: holdings
    base_table:
      database: ANALYTICS_DB
      schema: DWH
      table: DIM_HOLDINGS
    dimensions:
      - name: issuer_name
        expr: ISSUER_NAME
        data_type: VARCHAR
        synonyms: ["issuer", "company"]
      - name: portfolio_name
        expr: PORTFOLIO_NAME
        data_type: VARCHAR
    measures:
      - name: total_positions
        expr: COUNT(*)
        data_type: NUMBER
$$,
    FALSE
);
```

Handling SCD Type 2 in Semantic Views

Financial data warehouses typically use SCD Type 2 for dimension tables. This requires explicit handling in your semantic views to ensure Agents return current records and don't duplicate historical versions.

```
filters:
  - name: current_holdings
    description: Only current holding records (handles SCD Type 2)
    expr: |
      EFFECTIVE_DATE_START <= CURRENT_DATE()
      AND (EFFECTIVE_DATE_END > CURRENT_DATE()
           OR EFFECTIVE_DATE_END IS NULL)
```

Include custom instructions that enforce this filter on every query:

```
custom_instructions: |
  CRITICAL RULES FOR ALL QUERIES:

  **SCD Type 2 Handling:**
  - ALWAYS filter for current records using the effective date pattern
  - Apply this filter to EVERY dimension table
  - This is required for accurate results and is NOT optional

  **Joins:**
  - When joining to Issuer or Credit, include the SCD Type 2
    filter in the JOIN condition, not just WHERE
```

End-to-End Architecture: Persisting User-Created Fields

The practical path to allowing users to create calculated columns through natural language requires an application layer that handles persistence. Although Cortex can generate JSON, it cannot persist anything by itself.

The Flow

1. **User Request:** "Create a calculated field that multiplies LTM EBITDA by Total Leverage"
2. **API Layer:** Receives request, identifies this as expression generation (not a query), determines semantic context
3. **Cortex LLM Function:** API calls SNOWFLAKE.CORTEX.COMPLETE with prompt and column context, returns JSON expression definition
4. **Validation:** Parse JSON response, validate expression syntax against Snowflake, check column references exist, sanitize for SQL injection
5. **Persistence:** Save to your metadata store (MongoDB, Postgres, etc.)
6. **View Regeneration:** Your application picks up the new field definition and regenerates the view
7. **Available for Queries:** New column available in BI tools, reports, and Cortex Agent queries

The Validation Step is Critical

Before persisting any LLM-generated expression, validate it:

```sql
-- Test compile the expression without executing
SELECT LTM_EBITDA * TOTAL_LEVERAGE AS test_expr
FROM DIM_CREDIT
WHERE 1 = 0;  -- Returns no rows but validates syntax
```

If this fails, the expression is invalid. It is imperetive to not persist a broken definition. An error alert to the user is an alterate, safer approach.

UDF Strategy: Pre-Built Library Over Dynamic Creation

Since Agents cannot create UDFs, but can call them as tools, the practical approach is a pre-built UDF library. This requires upfront investment but provides consistent, tested calculations.

```sql
-- Create the UDF separately
CREATE OR REPLACE FUNCTION calculate_weighted_duration(
    market_values ARRAY,
    durations ARRAY
)
RETURNS FLOAT
LANGUAGE SQL
AS
$$
    -- calculation logic
$$;
```

Then register it as a custom tool in your Agent configuration:

```yaml
tools:
  - type: function
    name: calculate_weighted_duration
    description: 'Calculates portfolio weighted average duration
                  given market values and durations'
```

Cost Analysis

The entire orchestration layer is charged based on tokens used. Cortex Analyst charges per token. By keeping orchestration logic in your API layer when appropriate, costs can be reduced significantly.

Cost Comparison by Use Case

User Intent	Tool	Relative Cost
"Create a calculated field for X"	Cortex LLM Function	$
"What were my holdings on Dec 31?"	Cortex Agent + Analyst	$$
"Summarize the Acme contract"	Cortex Agent + Search	$$

User Intent	Tool	Relative Cost
"Compare holdings to contract terms"	Cortex Agent + Both	$$$

Recommended Architecture

The most practical architecture combines:

- **Pre-built UDF library** for complex, reusable calculations
- **Semantic Views with verified queries** for consistent Agent responses
- **Cortex LLM Functions** for expression generation (cheaper than full Agent)
- **Metadata store** (MongoDB, Postgres) for persisting user-created calculated columns
- **API layer** orchestrating the flow and handling validation/persistence

Conclusion

Cortex Agents are powerful for complex, multi-source queries; however they're overkill for many common tasks. Understanding when to use the full Agent stack versus a direct LLM call saves money and reduces complexity.

The key insight is that Agents are read-only orchestrators, not builders. Any persistence, object creation, or state management needs to happen in your application layer. Design for that constraint from the start, and the architecture falls into place.

Appendix: Example Semantic View for Credit Holdings

Below is a condensed semantic view for credit portfolio analytics, demonstrating SCD Type 2 handling, verified queries, and custom instructions. The full version which includes Credit and Portfolio dimension tables and additional verified queries is available in the GitHub repository.

```yaml
name: credit_holdings_analytics
description: |
  Analyze portfolio holdings with issuer, credit, and asset details.
  Supports point-in-time queries and aggregations across portfolios.

tables:
  - name: Holdings
    description: Portfolio positions with issuer and asset relationships
    base_table:
      database: ANALYTICS_DB
      schema: DWH
      table: DIM_HOLDINGS
    primary_key:
      columns: [DIM_KEY]
    dimensions:
      - name: HoldingKey
        expr: DIM_KEY
        data_type: NUMBER
      - name: IssuerName
```

```
        expr: ISSUER_NAME
        data_type: VARCHAR
        synonyms: ["issuer", "company", "funding source"]
      - name: PortfolioName
        expr: PORTFOLIO_NAME
        data_type: VARCHAR
        synonyms: ["portfolio", "fund"]
      - name: IsCurrent
        expr: >
          CASE WHEN EFFECTIVE_DATE_END IS NULL
               OR EFFECTIVE_DATE_END > CURRENT_DATE()
          THEN TRUE ELSE FALSE END
        data_type: BOOLEAN
    filters:
      - name: current_holdings
        description: Only current holding records (handles SCD Type 2)
        expr: |
          EFFECTIVE_DATE_START <= CURRENT_DATE()
          AND (EFFECTIVE_DATE_END > CURRENT_DATE()
               OR EFFECTIVE_DATE_END IS NULL)

  - name: Issuer
    base_table:
      database: ANALYTICS_DB
      schema: DWH
      table: DIM_ISSUER
    primary_key:
      columns: [DIM_KEY]
    dimensions:
      - name: IssuerName
        expr: NAME
        data_type: VARCHAR
      - name: IsPrivate
        expr: IS_PRIVATE
        data_type: BOOLEAN
      - name: Industry
        expr: INDUSTRY_NAME
        data_type: VARCHAR
    filters:
      - name: current_issuers
        expr: |
          EFFECTIVE_DATE_START <= CURRENT_DATE()
          AND (EFFECTIVE_DATE_END > CURRENT_DATE()
               OR EFFECTIVE_DATE_END IS NULL)

relationships:
  - name: holdings_to_issuer
    left_table: Holdings
    left_column: DIM_ISSUER_KEY
    right_table: Issuer
    right_column: DIM_KEY
    type: many_to_one

verified_queries:
  - name: current_holdings_by_issuer
    question:
      - "What are my current holdings by issuer?"
      - "Show me holdings grouped by company"
    sql: |
      SELECT
        i.NAME as IssuerName,
        COUNT(DISTINCT h.DIM_KEY) as PositionCount
      FROM DWH.DIM_HOLDINGS h
      INNER JOIN DWH.DIM_ISSUER i
        ON h.DIM_ISSUER_KEY = i.DIM_KEY
        AND i.EFFECTIVE_DATE_START <= CURRENT_DATE()
        AND (i.EFFECTIVE_DATE_END > CURRENT_DATE()
             OR i.EFFECTIVE_DATE_END IS NULL)
      WHERE h.EFFECTIVE_DATE_START <= CURRENT_DATE()
        AND (h.EFFECTIVE_DATE_END > CURRENT_DATE()
             OR h.EFFECTIVE_DATE_END IS NULL)
      GROUP BY i.NAME
      ORDER BY PositionCount DESC

custom_instructions: |
  CRITICAL RULES:
  - ALWAYS apply SCD Type 2 filters to every dimension table
  - Use INNER JOIN with SCD filters in the JOIN condition
  - Use COUNT(DISTINCT DIM_KEY) for position counts
  - Limit results to 1000 rows unless specified

  FORBIDDEN:
  - Omitting SCD Type 2 filters
  - SELECT *
  - Joining more than 4 tables without approval
```

For the complete semantic view with all dimension tables (Credit, Portfolio), additional verified queries, and implementation notes, see the companion GitHub repository.