Snowflake Intelligence Medicare POS Analyst — Medium Series

Generated on January 29, 2026

Repo: https://github.com/sahilbhange/snowflake-intelligence-claims-analyst

Build an AI Claims Analyst on Snowflake Intelligence (Hub Article)

Series: Snowflake Intelligence Medicare POS Analyst
Last updated: January 29, 2026
Repo: https://github.com/sahilbhange/snowflake-intelligence-claims-analyst

Healthcare claims analytics is full of “death by a thousand cuts”: code systems (HCPCS), device catalogs (GUDID), provider taxonomies, and messy join logic across large claims fact tables. Most teams end up with:

- A data warehouse that can answer **structured** questions (counts, spend, trends), but not “what does this code mean?”
- A document search that can retrieve PDFs, but not turn them into **actionable, auditable** answers.
- A chatbot demo that looks nice—until someone asks a question that requires **both** context and correct SQL.

This project is a compact, end-to-end demo that closes that gap using **Snowflake Intelligence**, combining:

- **Cortex Analyst** for trustworthy natural-language analytics over curated tables (structured data)
- **Cortex Search** services for “definitions + context” across HCPCS codes, medical devices (GUDID), and providers (unstructured / semi-structured text)
- Agent-style orchestration patterns (tool selection + guardrails), plus instrumentation and evaluation scaffolding

If you want a hands-on build that goes from raw data → curated model → semantic layer → search → agent-style Q&A, this series is designed for you.

What you will build

By the end, you’ll have a working “Medicare POS Analyst” experience:

1. Load and curate Medicare DMEPOS claims + provider reference data into a clean model
2. Publish a **semantic model** for Cortex Analyst (YAML) so business users can ask questions without SQL
3. Create **Cortex Search** services for:
- HCPCS codes (definitions)
- Medical devices via GUDID (brand/device text search)
- Providers (semantic provider lookup and filtering)
4. Connect these sources into **Snowflake Intelligence**, so a single conversational interface can:
- generate SQL against curated tables
- retrieve relevant definitions and supporting context when needed

---

Quickstart (as implemented in the repo)

If you want to follow along exactly, the repo’s Quickstart flow is:

- One-command option:
- `make demo` (after your Snowflake CLI connection is configured)
- Step-by-step scripts (high level):
1. Roles + warehouse + database/schema:
- `sql/setup/setup_user_and_roles.sql`
2. Download source data:
- `python data/dmepos_referring_provider_download.py --max-rows 1000000`
- `bash data/data_download.sh`
3. Upload raw files to stages:

- `sql/ingestion/load_raw_data.sql` (update `PUT` file paths)
4. Build curated tables/views:
- `sql/transform/build_curated_model.sql`
5. Create Cortex Search services:
- `models/cortex_search_hcpcs.sql`
- `models/cortex_search_devices.sql`
- `models/cortex_search_providers.sql`
6. Instrumentation + seed eval prompts:
- `models/instrumentation.sql`
- `models/eval_seed.sql`
7. Optional governance + quality scaffolding:
- `sql/governance/metadata_and_quality.sql`
8. Upload semantic model:
- `models/DMEPOS_SEMANTIC_MODEL.yaml` (target: `MEDICARE_POS_DB.ANALYTICS`)
9. Add sources in Snowflake Intelligence:
- `models/snowflake_intelligence_setup.md`

Hub & Spoke map (Table of Contents)

Read this series in order, or jump directly to what you need:

Spoke 1 — Data foundation: raw → curated claims model

Goal: get a clean, explainable data model that's ready for Analyst and dashboards.
■ Article: *Part 1 — From Raw CMS Files to a Curated DMEPOS Claims Model*

Spoke 2 — Semantic layer: Cortex Analyst YAML that doesn't hallucinate

Goal: make natural-language analytics reliable by defining metrics, filters, and joins explicitly.
■ Article: *Part 2 — Designing a Semantic Model for Cortex Analyst (YAML)*

Spoke 3 — Retrieval: Cortex Search services for codes, devices, and providers

Goal: add "definition search" and "entity lookup" so your agent can answer *what is X?* questions.
■ Article: *Part 3 — Cortex Search for HCPCS, GUDID Devices, and Providers*

Spoke 4 — Snowflake Intelligence orchestration: when to use Analyst vs Search

Goal: wire the experience into an agent-style flow with routing, guardrails, and evaluation.
■ Article: *Part 4 — Building a Snowflake Intelligence Agent that Uses Analyst + Search*

Spoke 5 — Operationalizing: evals, observability, governance, and demo prompts

Goal: make the demo credible: repeatable evaluation, safety boundaries, and quality signals.
■ Article: *Part 5 — Shipping the Demo: Evals, Instrumentation, and Governance*

How to read this series (recommended)

• **If you're new to Snowflake Intelligence:** read Parts 1 → 4, skim Part 5.
• **If you already have a warehouse model:** start at Part 2 (semantic layer), then Part 3 (search) and Part 4 (orchestration).
• **If you care most about agent reliability:** focus on Part 2 + Part 4 + Part 5.

Why this project is a strong "learning demo"

This is more than a “prompting” tutorial:

- It forces you to build the *minimum* data platform assets that agents actually need (curated model + semantics + retrieval).
- It demonstrates the real-world pattern: **structured analytics + retrieval + guardrails** in one place.
- It’s reproducible: scripts, make target, and explicit assets make the walkthrough easy to follow.

Next: Part 1 — raw → curated model.

Part 1 — From Raw CMS Files to a Curated DMEPOS Claims Model

Series: Snowflake Intelligence Medicare POS Analyst
Part 1 of 5 — Data foundation
Repo: https://github.com/sahilbhange/snowflake-intelligence-claims-analyst

In this series, we’re building an AI-enabled “claims analyst” experience. Before we touch semantic models or search, we need a baseline reality:

> If the curated model is wrong, the agent will be confidently wrong—faster.

This article focuses on what most AI demos skip: **data modeling and curation**.

What this part covers

By the end of Part 1, you will have:

- A working Snowflake database + schema for the demo
- Raw files staged in Snowflake
- Curated tables/views built for analysis at a clear grain
- A mental model of what’s “fact” vs “dimension” in this dataset

---

Step 0 — Create your working Snowflake environment

Start with roles, warehouse, and database/schema:

- `sql/setup/setup_user_and_roles.sql`

Why this matters: Most “agent demos” break when permissions are messy. If you use a clean, minimal role setup from day 1, the rest of the series becomes frictionless.

Step 1 — Download the source data

The repo includes helpers to pull the datasets needed for the claims + reference model:

- `python data/dmepos_referring_provider_download.py --max-rows 1000000`
- `bash data/data_download.sh`

Tip: Keep the first run smaller (e.g., 100k–300k rows) so you can iterate quickly.

Step 2 — Upload raw files into Snowflake stages

Use:

- `sql/ingestion/load_raw_data.sql`

Update the `PUT` file paths to your local download location.

At this point, you should be able to validate:

• files exist in stage
• file formats are correct
• basic row counts look sane

Step 3 — Build the curated model

Now run:

• `sql/transform/build_curated_model.sql`

This is the most important script in the project because it turns raw files into something a semantic layer can safely sit on top of.

What a “good” curated model looks like for an agent

Even if you never publish dashboards, you want:

• **Stable naming:** predictable column names and consistent typing
• **Explicit grain:** it’s clear what each table row represents
• **Joinable dimensions:** provider and code dimensions can be joined without surprise duplication
• **Derived fields:** common filters and metrics are pre-defined (dates, payment amounts, state, specialty, etc.)

Think of the curated model as the contract between:
• humans (analysts, stakeholders)
• automation (Cortex Analyst SQL generation)
• retrieval (Search results you might join back to facts)

Sanity checks (do this before moving on)

Before Part 2 (semantic model), validate:

1. **Row counts**: do curated tables match expectations?
2. **Keys**: do you have stable keys for provider, HCPCS, and claim-ish grain?
3. **Null hotspots**: are “join keys” populated where expected?
4. **Simple questions** (write SQL once):
- Top 10 states by claim volume
- Top 10 HCPCS by allowed amount
- Average payment per claim by specialty

These become your baseline truth for evaluating Cortex Analyst later.

What you gained (and why it’s a big deal)

Most people try to build an agent directly on top of raw, messy tables. This project does the opposite:

• Curate first
• Add semantic meaning second
• Add retrieval third
• Orchestrate last

That order is why the demo holds up under real prompts.

Next: Part 2 — publish a semantic model YAML for Cortex Analyst.

Part 2 — Designing a Semantic Model for Cortex Analyst (YAML)

Series: Snowflake Intelligence Medicare POS Analyst
Part 2 of 5 — Semantic modeling for reliable text-to-SQL
Repo: https://github.com/sahilbhange/snowflake-intelligence-claims-analyst

In Part 1, we created a curated claims model. Now we make it usable in natural language.

Cortex Analyst is powerful, but it’s not magic: it needs a **semantic contract**. In this project that contract is:

- `models/DMEPOS_SEMANTIC_MODEL.yaml`

This article shows how to think about semantic modeling for agent reliability—not just “make it pass validation.”

The goal: make SQL generation predictable

A good semantic model does three things:

1. **Maps business language to schema**
- “claim volume” → which column/count?
- “allowed amount” → which numeric field?
2. **Constrains ambiguity**
- defines join paths (relationships)
- defines canonical filters (state, year, rentals vs non-rentals, etc.)
3. **Codifies safe output habits**
- top-N defaults
- rounding conventions for currency
- row limits for safety and performance

Upload the model

The repo’s recommended target is:

- Database/schema: `MEDICARE_POS_DB.ANALYTICS`
- YAML file: `models/DMEPOS_SEMANTIC_MODEL.yaml`

Upload it via Snowsight (Cortex Analyst) or by staging the file and referencing it, depending on your preferred workflow.

How to structure a robust YAML

Even if you keep the YAML small, follow this mental model:

1) Logical tables match business entities

Typical pattern:

- Provider dimension (who)
- Claims fact (what happened)
- Optional code/device dimensions (what it is)

2) Relationships are explicit and minimal

Define a single canonical join path for “facts → provider,” etc.
Avoid “multiple plausible joins” unless you add strong disambiguation.

3) Metrics are first-class

Instead of forcing the model to derive everything from raw columns, define metrics:

- claim count
- total allowed
- total payment
- payment-to-allowed ratio (if used)

Metrics reduce hallucinations because the model can use known-good SQL patterns.

4) Named filters reduce prompt entropy

Add named filters for repeated patterns:

- last year (or last available year)
- specific state
- rentals vs non-rentals
- high-volume provider thresholds (if relevant)

When the agent sees “top providers in CA,” it can snap to a known filter + metric combination.

5) Custom instructions are guardrails, not novels

If you include custom instructions, keep them tight:

- default time range if none is provided
- cap output rows
- always add ORDER BY for “top/highest”
- round currency to 2 decimals

Long “wall of rules” tends to become self-contradictory. Prefer a short set of strong priors.

Verification workflow (don’t skip)

After uploading the YAML, test with prompts that map cleanly to known SQL:

- “Top 10 states by claim volume”
- “Average payment by specialty”
- “Top HCPCS by total allowed amount”

Then test ambiguity cases:

- “Top providers” (what does “top” mean?)
- “Most expensive devices” (is this allowed amount, payment, or charges?)

If the SQL output is inconsistent, fix the semantic model—not the prompt.

Common semantic model pitfalls (and how this repo avoids them)

- **Pitfall:** too many columns → the model loses the plot
Fix: keep only analyst-relevant columns; push everything else to “later extensions.”

- **Pitfall:** unclear grain → counts inflate
Fix: define grain explicitly in descriptions; ensure joins don’t duplicate.

- **Pitfall:** missing synonyms → users rephrase and break things
Fix: add descriptions that include synonyms (claim volume = claim count, etc.)

Next: Part 3 — add Cortex Search services for codes, devices, and providers.

Part 3 — Cortex Search for HCPCS, GUDID Devices, and Providers

Series: Snowflake Intelligence Medicare POS Analyst
Part 3 of 5 — Retrieval for “what is this?” questions
Repo: https://github.com/sahilbhange/snowflake-intelligence-claims-analyst

Cortex Analyst is great for questions that can be answered with SQL. But in claims work, users constantly ask questions that need **definitions and context**:

- “What is HCPCS E1390?”
- “Find oxygen concentrators”
- “What does this device brand map to?”
- “Find endocrinologists in California” (semantic entity lookup)

That’s what Cortex Search is for.

In this repo, Search is implemented as three services:

- `models/cortex_search_hcpcs.sql`
- `models/cortex_search_devices.sql`
- `models/cortex_search_providers.sql`

Why search services matter in an agent

Search does two things an LLM should not guess:

1. **Retrieves the authoritative text** (descriptions, definitions, product names)
2. **Converts fuzzy user intent into joinable entities**
- a provider name → NPI
- a device phrase → a GUDID record
- a code description → HCPCS code

That retrieval becomes grounded context the agent can:
- quote
- link to
- join back to fact tables
- and use to explain results

Service 1: HCPCS semantic search

User prompts this enables:
- “What is HCPCS code E1390?”
- “Which codes relate to oxygen therapy?”

Design notes:
- Index: code + short/long descriptions
- Output: code, description, (optional) category/grouping

Service 2: Devices semantic search (GUDID)

User prompts this enables:
- “Find oxygen concentrators”
- “Search for [brand] [model]”
- “What is the FDA device identifier for …?”

Design notes:
- Devices are text-heavy; names and synonyms matter
- Retrieval results often need light post-processing:
- normalize brand/model naming
- pick top-k and show short summaries

Service 3: Provider semantic search

User prompts this enables:
- “Find endocrinologists in California”
- “Providers near ZIP …”
- “Which providers match ‘arthritis’ specialty terms?”

Design notes:
- Search is strongest when you embed:
- provider name variations
- specialty text
- address/city/state fields (as text)
- The best user experience is when Search results return:
- provider identifier (NPI)
- display name
- a few supporting attributes (specialty, state, city)

How search and Analyst work together

A good orchestration pattern is:

1. Use **Search** to resolve entities/definitions
2. Use **Analyst** to run SQL using that entity as a filter
3. Respond with:
- the computed answer
- the retrieved definitions
- the “how we got here” explanation

Example:

> “Find endocrinologists in California, then show top 10 by claim volume.”

Search handles the specialty + geography entity lookup; Analyst handles the aggregation.

Next: Part 4 — wire Analyst + Search into a Snowflake Intelligence agent-style experience.

Part 4 — Building a Snowflake Intelligence Agent that Uses Analyst + Search

Series: Snowflake Intelligence Medicare POS Analyst
Part 4 of 5 — Orchestration and guardrails
Repo: https://github.com/sahilbhange/snowflake-intelligence-claims-analyst

At this point you have:

- Curated tables/views (Part 1)
- A semantic model YAML for Cortex Analyst (Part 2)
- Cortex Search services for codes, devices, and providers (Part 3)

Now you assemble them into a single experience in Snowflake Intelligence.

The key design problem is tool routing:

> When should the system use Analyst vs Search, and how do we avoid “wrong tool” answers?

The repo points you to:

- `models/snowflake_intelligence_setup.md` (source setup guidance)

The orchestration pattern

Use a simple decision tree:

Use Cortex Analyst when:

- the user asks for aggregates, trends, “top N”, comparisons, metrics
- the question maps to the curated model + semantic metrics

Examples:

- “Top 10 states by claim volume”
- “Average payment by specialty”
- “Payment-to-allowed ratio by state”

Use Cortex Search when:

- the user asks “what is X?”
- the user asks for an entity lookup (“find providers/devices/codes”)
- the question needs textual context that isn’t in the fact tables

Examples:

- “What is HCPCS E1390?”
- “Find oxygen concentrators”
- “Find endocrinologists in California”

Use both when:

- the user combines lookup + analytics

Examples:

- “Find oxygen concentrators and show total allowed amount by state.”
- “Which providers match endocrinology in CA and what is their claim volume?”

Guardrails that make the demo credible

Even demos need safety and trust boundaries:

1. **Row limits**
- Default to top 10, or require filters for large result sets
2. **Explainability**
- Always show the metric definition (from semantic model)
- Provide the key filters you applied
3. **No PHI / no patient-level output**
- Keep analysis at provider + HCPCS grain (as the project does)
4. **Deterministic handling of “top/highest”**
- Always add ORDER BY + LIMIT
5. **Currency formatting**
- Round to 2 decimals; label units clearly

Demo prompt playbook

These are great “show me it works” prompts:

• “Top 10 states by claim volume”
• “What is HCPCS code E1390?”
• “Find oxygen concentrators”
• “Find endocrinologists in California”

Try them in that order. It demonstrates:
• structured analytics
• definition lookup
• semantic search
• and a combined “claims analyst” narrative

Next: Part 5 — evaluation + instrumentation + governance scaffolding.

Part 5 — Shipping the Demo: Evals, Instrumentation, and Governance

Series: Snowflake Intelligence Medicare POS Analyst
Part 5 of 5 — Making it repeatable and trustworthy
Repo: https://github.com/sahilbhange/snowflake-intelligence-claims-analyst

You can build a flashy agent in an hour. You can build a demo that holds up under real questions only if you add discipline:

- evaluation
- instrumentation
- governance

This repo includes starting points for all three.

Instrumentation: know what the agent actually did

Start with:

- `models/instrumentation.sql`

What you want to log (at minimum):

- prompt text (or a safe hash if needed)
- tool calls (Analyst vs Search)
- SQL produced (if using Analyst)
- top-k search results (if using Search)
- latency and token/cost signals (where available)

Even a small logging table changes the quality of iteration.

Seed evaluations: build a regression set

Use:

- `models/eval_seed.sql`

The intent is to create a baseline prompt suite:

- easy prompts that should never regress
- “combined prompts” that test tool routing
- edge prompts that often cause hallucinations (“top providers” ambiguity, etc.)

Treat eval prompts like unit tests:
- run them after any semantic model update
- run them after any curated model change
- compare outputs for drift

Governance + quality scaffolding

Optional but strongly recommended:

• `sql/governance/metadata_and_quality.sql`

In a demo, governance doesn’t have to be heavy—but it must be present:

• basic metadata (what tables mean, who owns them)
• basic data-quality checks (row counts, null rates on join keys)
• documented grain and join rules

This is what makes the series credible for real teams.

“Why this demo works” — the punchline

Most AI agent demos fail because they treat data as an afterthought.

This project is effective because it is a complete pipeline:

1) curated model
2) semantic contract
3) retrieval services
4) orchestration with guardrails
5) evaluation and observability

That stack is the difference between:
• a nice chatbot screenshot
• and an AI-enabled analytics workflow people actually trust

Where to take it next

If you want to extend this series:

• Add richer provider aliasing (name variants, organizations)
• Add a lightweight UI (Streamlit) for guided prompts
• Add a “definition + analytics” blended response template with citations
• Add cost controls (warehouse sizing, search service refresh cadence)

If you publish this as a learning series, you’ll stand out because you’re teaching the thing most people skip:
how to make the agent reliable.