AI ALLIANCE · ELECTRONICS RETAIL POC · 2026

How the pipeline actually works

A step-by-step breakdown of what each MCP tool does, how it does it, and why — at the level of logic and algorithm, not code. The companion deep dive to *Building a Governed Conversational BI Agent*.

TRANSFORMATION AT A GLANCE — WHAT EACH STEP PRODUCES

S1	S2	S3	S3.25	S3.5	S4	S5
Classify	**Validate**	**Ontology Map**	**SKU Prefilter**	**SKU Filter**	**TLK / SQL**	**Execute**
→ jurisdiction decision	→ validated analytical intent	→ matched ontology nodes	→ reduced item pool	→ exact item_names list	→ sql_id + confidence	→ result + summary
LLM	LLM + DET.	LLM × 2	DETERMINISTIC	LLM	TLK + LLM	DUCKDB + LLM

STEP 1

Classify

Jurisdiction before analysis

LLM

RECEIVES

User question (natural language)

PRODUCES

Category: electronics_retail / electronics_non_retail / general

WHAT IT DOES AND HOW

A language model classifies the question into one of three categories. The model prompt defines each category with examples, making this a structured classification task rather than open-ended reasoning.

electronics_retail — the question asks for analytical data about electronics merchandise. It must be a data question, not a knowledge question.

electronics_non_retail — about the company but not the retail data domain (e.g. store locations, return policies).

general — definitional, explanatory, or unrelated (e.g. "What is Bluetooth?").

Only `electronics_retail` enters the analytical pipeline. The other two receive a direct answer from Claude Desktop and stop.

WHY THIS STEP EXISTS

The distinction that matters most is between a question that *mentions* a product and a question that *asks for data about* a product.

"What is a smartphone?" and *"How many smartphones did we sell?"* share a product word. Only one of them requires the governed analytical backend.

S1 establishes the first boundary: the backend only reasons over questions that fall inside its analytical jurisdiction. Without this gate, the backend would attempt to run an analytical workflow over questions it was never designed to answer.

> DESIGN NOTE
>
> If this step returns `electronics_non_retail` or `general`, the conversational shell handles the answer directly using the LLM's general knowledge. The Project Instructions make this routing explicit and non-negotiable.

↓ only if electronics_retail ↓

STEP 2

Validate

Turning a question into a validated analytical intent

Phase A — LLM extraction

Phase B — deterministic validation

RECEIVES

Natural language question

PRODUCES

Structured intent: concept, metric, date_range, location, size — each validated

PHASE A — EXTRACTION (LLM)

A language model reads the question and outputs a structured JSON object with five fields: `concept` (the product, split into *main* and *attribute*), `metric`, `date_range`, `location`, `size`.

The concept split is important: *"gaming laptops"* → main: `laptops`, attribute: `gaming`. The attribute travels downstream to S3.5 where it helps the LLM discriminate between laptop types. Stripping it at this stage would cause false positives later.

The model is instructed to be conservative: only split when the qualifier is clearly separable. When in doubt, keep everything in `main`.

PHASE B — VALIDATION PER DIMENSION

Each dimension extracted in Phase A is validated using a different method, chosen for the nature of the constraint:

DIMENSION	METHOD	LOGIC
concept	`Deterministic`	SKU lookup first, then ontology search. No model call.
date	`Deterministic`	Pure Python regex + month/quarter maps. Validates against available data window (2012–today).
metric	`LLM` + `Det.`	LLM decomposes intent into metrics, operations, filters. Deterministic validator then checks combinations against the metrics ontology.
location	`LLM`	Simple model check: is this a real-world location?
size	`Extracted only`	No validation — passed downstream for deterministic filtering in S3.25.

> **KEY DESIGN DECISION**
>
> The metric validator doesn't just check if the metric word is known. It checks whether the **combination** of metric, dimension, and operation is institutionally legitimate — e.g. you can't average a stock (snapshot) metric the same way you average a sales (flow) metric.

↓ only if is_valid ↓

STEP 3

Ontology Map

Semantic grounding in the product hierarchy

- Two LLM passes (temperature 0, seed 0)
- Deterministic synonym fallback

RECEIVES

`concept.main from S2`

PRODUCES

`matched_nodes: list of {family, subfamily, product_type, sku_count}`

HOW THE TWO-PASS SEARCH WORKS

Before any LLM call, the system looks up the concept in `synonyms.json` and collects all known roots in Spanish and English. These roots are injected into both LLM prompts, so the model sees not just "laptops" but also "notebook", "ultrabook", "chromebook", etc.

Pass 1 — Recall `LLM`

The model scans the ontology skeleton — a compact summary of all subfamilies with 8 example product types each. Its instruction is explicit: **prefer inclusion over exclusion**. Every subfamily that could plausibly contain the concept is returned, including ambiguous ones. False negatives here are more expensive than false positives because downstream steps handle pruning.

Pass 2 — Precision `LLM (PARALLEL)`

One call per candidate subfamily from Pass 1, run in parallel. The model sees the full list of product types within each subfamily and selects those that genuinely match. Synonyms are also injected here to catch abbreviated product names.

FALLBACK AND SPECIAL CASES

If both passes return zero nodes, the system falls back to a deterministic lookup in `ontology_synonyms.json`. This file maps terms directly to ontology nodes. "phones" → Smartphone, "laptop bag" → Laptop Sleeve. No model call. This catches non-Spanish or very abbreviated concepts that the LLM doesn't recognise from the skeleton.

If `granularity = all` (the question has no product concept — e.g. "total billing in 2025"), S3 returns an empty matched_nodes list and S3.25 and S3.5 are skipped entirely. S4 generates SQL without a product filter.

Results are cached per concept within the server session. The same concept in different questions within a session doesn't trigger new LLM calls.

> **WHY TWO PASSES AND NOT ONE?**
>
> The ontology has 95 product types across 29 subfamilies. Showing all of them to a single LLM call would exceed context and produce noisy results. Pass 1 narrows to relevant subfamilies; Pass 2 applies precision within each. This hierarchical search is both cheaper and more accurate than a flat scan.

↓ item pool from matched nodes ↓

STEP 3.25

SKU Prefilter

Cheap deterministic reduction before the LLM pass

- Size — pure Python regex
- Date — single DuckDB query on OITM

RECEIVES

`Item pool from matched_nodes + size + date_range from S2`

PRODUCES

Smaller item pool passed to S3.5

SIZE FILTER — PURE PYTHON, ZERO MODEL CALLS

Item names in the catalogue follow a consistent format: the trailing parenthesis encodes the size, e.g. `Smartphone 6.1" Standard 128GB (BLACK)`.

The filter extracts the size token from that parenthesis using a regex, then matches it against the size requested in S2 using a normalisation table that handles variants like `BLACK → BLACK`, `SILVER → SILVER`, etc.

If the user asked for size L, all items not ending in a size-L token are removed. No database access, no model call. Pure Python string matching over the item name strings already in memory.

DATE FILTER — DUCKDB ON OITM.CREATE_DATE

The date filter removes items that **could not have existed** during the requested period. It queries the product master table (OITM) for the creation date of each candidate item and retains only those created on or before the end of the requested period.

This is **not a sales-in-period filter**. It does not ask whether the item sold during the period. It asks whether the item existed by the end of the period — a conservative, cheaper check that doesn't require joining the sales table.

> **IMPORTANT CAVEAT**
>
> Because this filter uses `create date` and not sales activity.

it is safe for **sales queries** (items with zero sales in the period contribute zero and don't change the result) but should **not** be used as the sole filter for stock queries where an item might have been created after the period.

DESIGN PRINCIPLE

Use deterministic logic where the constraint is deterministic. Reserve LLM calls for semantic ambiguity. Both filters here are pure logic — size is a string match, date existence is a factual check. No model should be involved in either.

↓ filtered item pool ↓

STEP 3.5

SKU Filter

Live database grounding: ontology nodes → exact item_names

- Deduplication by product signature
- LLM relevance classification

RECEIVES

Item pool from S3.25 + full concept (main + attribute)

PRODUCES

item_names — exact strings from the live database, ready for SQL IN clause

WHY THIS STEP IS NECESSARY

The ontology is stable. The database is not. A product type like `Laptop 13" Ultrabook` exists as a stable node — but in the database it appears as:

```
Laptop 13" Ultrabook 8GB/256GB SILVER
Laptop 13" Ultrabook 16GB/512GB SILVER
Laptop 13" Ultrabook 16GB/512GB GREY
```

If S4 used the ontology node name — or a model-recalled product name — in the SQL filter, the query would execute without error but return wrong results. Silently.

S3.5 resolves this by classifying the **actual strings in the live database** for relevance to the concept. The item_names it produces are the only strings allowed in the SQL IN clause.

THE FULL CONCEPT MATTERS HERE

The concept passed to S3.5 is **"gaming laptops"**, not just "laptops". The attribute (*gaming*) is what allows the LLM to correctly reject `Laptop Sleeve` while keeping `Laptop 17" Gaming`. Stripping the attribute at S2 would break this step.

HOW IT WORKS — TWO PATHS, AUTOMATIC ROUTING

Before any LLM call, size variants are collapsed into **product signatures** by stripping the trailing colour/storage token: `Laptop 13" Ultrabook 8GB/256GB SILVER` → `Laptop 13" Ultrabook 8GB/256GB`. The LLM judges the signature once; the decision is then expanded to all size variants.

PATH	WHEN	LOGIC
Path B — LLM all	≤ 10,000 unique signatures (most real queries)	All signatures sent to the LLM in one call. The model receives the ontology path per item as primary signal plus few-shot examples of both failure modes.
Path A — Lexical + LLM	> 10,000 signatures (very broad concepts)	A lexical root filter eliminates clear mismatches first. The LLM only sees the ambiguous remainder.

The two failure modes the LLM resolves that lexical cannot:

False positives: `Laptop Sleeve` contains "Laptop" but is a bag accessory. Lexical passes it; LLM correctly rejects it.

False negatives: `Chromebook 11"` is a laptop but the root "laptop" is absent. Lexical rejects it; LLM recovers it.

If the LLM call fails, the step falls back to returning all items — a safe recall guarantee.

↓ exact item_names list (server-side) ↓

STEP 4

TLK Lookup / SQL

SQL generation under institutional authority

- TLK template matching (deterministic)
- LLM adaptation or generation (MEDIUM/LOW)

RECEIVES

metric_info (S2) + date_range (S2) + item_names (S3.5) + DDL + TLK library

PRODUCES

sql_id (stored server-side) + confidence regime

HOW TLK MATCHING WORKS

The question is **normalised** before matching: product references become `<ITEM_NAME>`, dates become `<YYYY>` / `<MM>`. This creates a structural representation of the question that can be compared against template questions regardless of specific values.

The normalised question is matched against the 70 verified templates in the TLK library. If a template matches:

SQL is instantiated **deterministically** from the template. Placeholders are filled with the validated values from S2 and the exact `item_names` from S3.5. No LLM generation. No interpretation.

The `metric_info` from S2 is passed explicitly into every LLM prompt in this step — the model doesn't need to re-infer the analytical intent from the question text. It already knows what metric, operation, and filters are required.

CONFIDENCE REGIMES AND GUARDS

ACCURATE	Exact TLK template match. SQL built deterministically from the template and the S3.5 item_names list. No free generation.
MEDIUM	Partial TLK match. LLM adapts the closest verified template. Template lineage is visible in the output.
LOW	No TLK match, or date expression is too complex for template instantiation (multi-year, relative, seasonal). LLM generates SQL from DDL + metric_info + ontology context. Warning shown to user.
BLOCKED	item_names list from S3.5 is empty. Query rejected entirely. Prevents the system from silently returning a grand total of the full catalogue.

THE SQL GROUNDING RULE

Regardless of confidence regime, SQL must use `item_name IN (...)` built from the S3.5 output. **ILIKE patterns are forbidden. Model-recalled product names are forbidden.** The SQL must be grounded in live item strings — or it does not execute. The sql_id is stored server-side; the shell only receives the identifier.

↓ sql_id passed to S5 ↓

STEP 5

Execute

Execution and governed response synthesis

- DuckDB SQL execution
- LLM natural language summary

RECEIVES

`sql_id + confidence + original question`

PRODUCES

Computed result + natural language summary + confidence badge

EXECUTION

S5 retrieves the SQL from the server-side store using the `sql_id` returned by S4. The SQL is never sent through the conversational shell — the frontend only held the identifier. This keeps the analytical artifact inside the governed backend.

RESPONSE SYNTHESIS AND AUTHORITY QUALIFICATION

A language model generates a concise natural language summary of the result, using the original question as context. The model is not interpreting data freely — it is summarising a computed result within a narrow prompt.

DuckDB executes the SQL against local parquet files that mirror the Athena production schema. The execution result is a CSV-format table with row counts and computed values.

If execution fails (malformed SQL, missing table), the error is reported and the pipeline stops. It does not retry or improvise an alternative query.

The response always includes three elements:

The numerical result — the direct answer to the question.

The summary — a brief interpretation in plain language.

The confidence badge — ACCURATE, MEDIUM, or LOW, always visible. If LOW, a warning is prepended before the summary explaining that the SQL was auto-generated and the result should be verified.

> WHY THE BADGE MUST BE VISIBLE
>
> A confidence regime hidden inside the backend has no value. The badge is for the user — it tells them whether this answer follows a verified institutional pattern or was generated without one. A system that presents all answers with the same authority is not being honest about what it knows.

METHOD TYPES ■ LLM — language model call ■ Deterministic — algorithm, regex, or database query; no model ■ Mixed — both in sequence

Supplementary Material A · Pipeline Deep Dive. Companion to *Building a Governed Conversational BI Agent: Lessons from a Retail POC.*
Developed at Sevilla FC's Technology Department in collaboration with IBM under the AI Alliance Organisation.

AI ALLIANCE · 2026