TRUSTLOGIX

WHITE PAPER

# **Beyond Agent Identity:** Extending Snowflake's Agent Controls with Intent-Aware Governance

CONTENTS


Executive summary 2

Purpose of this white paper 3

What Snowflake's agent identity does 3

How Snowflake's agent identity is used 4

The value of dual attribution 4

Where Snowflake's agent controls stop 5

How TrustLogix enhances Snowflake Dual Attribution 6

Supporting Modern AI Agent Security with TrustLogix and Snowflake 11

Appendix 13

# Executive summary

When a user asks an AI agent to access Snowflake data, the agent runs with their role and inherits everything they can see. No one decides whether the agent should have that access. It simply inherits it.

Snowflake's new agent identity capabilities change that. A session can now carry the user's identity as well as an agent flag. Masking and row access policies can act on that flag, Restricted Session Scope can cap agent privileges, and audit views record agent activity. This is the right foundation for agents running inside Snowflake.

However, Snowflake can only govern what reaches Snowflake. Its agent flag has no notion of which agent is asking or why, only indicating whether there is an agent involved. When external agents connect through a shared service user, Snowflake sees an agent but not the user behind it. Each policy is SQL written by hand for Snowflake alone, and an agent that also uses Databricks, Salesforce or GitHub needs the same work repeated in each.

TrustLogix builds on Snowflake's dual attribution in four layers, covering what enterprises need as they secure AI agent access (as shown in **Image 1**):

**LAYER 1**

**TrustAI** decides each tool call on the agent, the user and whether the request fits the agent's declared purpose.

**LAYER 2**

The **TrustLogix Policy Decision Point (PDP)** signs in to Snowflake on behalf of the user, so Snowflake sees the real user in an agent-active session and dual attribution works for external agents too.

**LAYER 3**

Create agent-aware masking into native Snowflake policy with **TrustAccess**.

**LAYER 4**

**Guardian Agent** turns Snowflake's own agent logs into risk findings, mapped to the compliance frameworks they affect. Read alongside the TrustAI decision record, it shows which external agent asked and what Snowflake did.

By combining these controls, enterprises gain fine-grained intent and context governance. In practice, TrustAI checks intent at the tool call, so it can refuse a request that doesn't fit the agent's purpose even when the user is entitled to the data, then allow the next request from the same agent and user. Inside Snowflake, TrustAccess applies the user's row scope and masks columns when that same user's query runs through an agent, including Snowflake's own agents.

## Purpose of this white paper

Most enterprises will run a mix of agents: Snowflake's own Cortex agents, and external agents built in Foundry, Copilot Studio or their own code that reach Snowflake alongside other systems. This white paper explains Snowflake's new agent identity capabilities and what they give you, then shows how TrustLogix builds on them for the cases enterprises hit next: which external agent is acting, who it acts for, whether each request fits its declared purpose, and what data any agent sees. The test results come from our own Snowflake and TrustLogix tenants.

## What Snowflake's agent identity does

Snowflake calls it agent identity: the ability to recognize when an AI agent is acting in a session, so agent traffic can be governed separately from human traffic. Its core is **dual attribution**. A single session carries two pieces of information:

- **The user (who authorized it).** The session belongs to the user. The agent runs with the user's identity, role and permissions. In Snowflake's words, "The session belongs to the user; Snowflake marks it agent-active so policies and audit views can treat the traffic as agentic."
- **The agent flag (that an agent carried it out).** Snowflake marks the session agent-active, and `IS_AGENT_ACTIVATED` returns TRUE. Audit views record the agent type. For Snowflake's own agents, they also record which agent ran and which objects it touched.

### The user stays the user, and the agent flag comes along with them.

## How Snowflake's agent identity is used

Teams put agent identity to work in four ways:

- **Marking agent sessions.** Snowflake marks a session agent-active based on how it arrived:
  - Cortex Agents and Cortex Code clients
  - OAuth integrations set with `IS_AGENTIC = TRUE`
  - any call through a Snowflake-managed MCP server
  - users created with the `SERVICE_AGENT` type

  Snowflake's own agents and the managed MCP server need no extra setup.
- **Writing agent-aware policies.** `SYS_CONTEXT('SNOWFLAKE$CURRENT', 'IS_AGENT_ACTIVATED')` returns TRUE in an agent-active session. Teams use it in masking, row access, projection, aggregation and join policies. For example, a policy can show a contact name to the user and mask it when an agent runs the same query.
- **Setting a privilege ceiling.** Restricted Session Scope, set on a session policy, narrows what the user's roles allow while an agent is active. It never grants anything new.
- **Auditing agent activity.** `QUERY_HISTORY.agent_type` and `ACCESS_HISTORY.agents_info` show which queries an agent ran and which objects it touched.

## The value of dual attribution

With dual attribution, Snowflake can apply different controls to a user's direct access and to an agent acting on their behalf, while keeping the user's identity in the session. It provides three benefits:

- **More precise access controls.** Masking and row access policies can act on the agent flag. A user can see customer names directly, while an agent acting for them sees masked names. Restricted Session Scope can also cap what an agent can do below the user's own privileges.
- **Preserved user permissions.** When the connection runs on behalf of the user, their existing access boundaries still apply to the agent. For example, someone who can see only North America accounts keeps that limit when an agent acts for them.
- **Clearer auditing.** Query and access history record the session user and flag agent activity, so teams can tell direct queries from agent-driven ones.

The key value is that a user's permission to access data no longer has to mean their agent gets the same visibility. But the agent flag only makes this possible. Someone still has to write the policies that impose the additional restrictions.

## Where Snowflake's agent controls stop

Snowflake governs the session it can see, and it does that well. Modern AI agent scenarios, however, encompass a much broader scope.

1 **The agent flag is yes or no.**
It tells a policy that an agent is present. It doesn't say which agent, what it was asked to do, or whether that fits what the agent is for.

2 **External agents arrive without a name.**
For external agents, `agents_info` carries no agent ID. When we sent calls through a Snowflake-managed MCP server, Snowflake recorded them as its own lite agent or as the Cortex agent behind the tool. The external agent that actually made the call didn't appear anywhere.

3 **Snowflake only knows the user when the session runs as the user.**
If an agent connects with a shared service user, Snowflake sees an agent but not the user behind it, so any policy keyed on the user stops working. A key pair or programmatic access token (PAT) session outside the managed MCP server isn't treated as an agent at all, unless the user is a `SERVICE_AGENT`.

4 **Restricted Session Scope sits on the account or the user, not the agent.**
Every agent acting for that user gets the same ceiling, and it's fixed for the life of the session.

5 **It covers Snowflake only, and it's written by hand.**
Each policy is SQL you write and attach yourself. An agent that also uses Databricks, Salesforce or GitHub needs the same thinking repeated everywhere else.

None of this is a criticism. Snowflake can only govern what reaches Snowflake. The questions it can't answer come before the session starts and span the other systems an agent uses. That's where TrustLogix comes in.

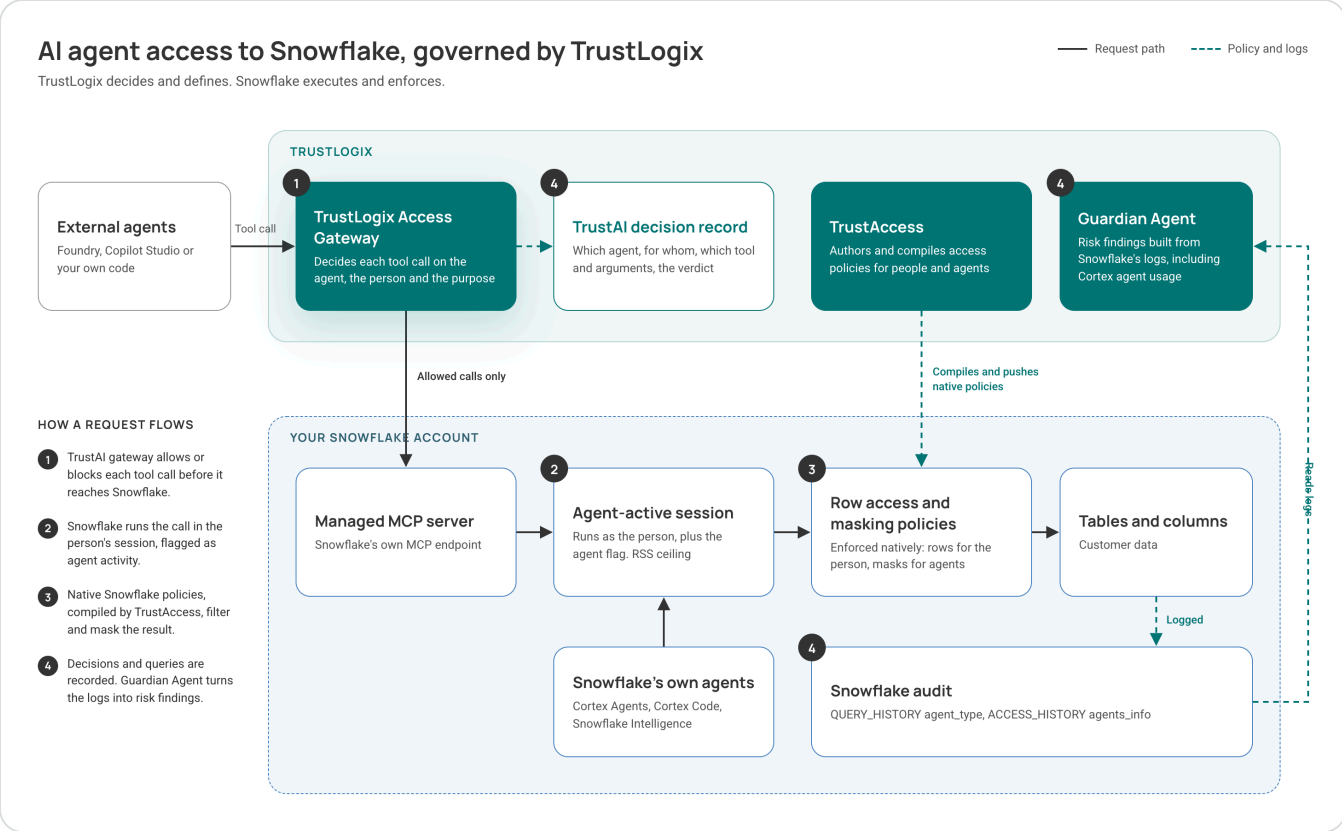



**Image 1:** The four layers on one agent request. External agents come through the TrustAI gateway; Snowflake's own agents start inside Snowflake, so layers 2 to 4 still apply to them.

## How TrustLogix enhances Snowflake Dual Attribution

TrustLogix builds on Snowflake's agent identity in four layers, from the tool call to the column. External agents come through the TrustLogix access gateway. Snowflake's own agents, Cortex Code and Snowflake Intelligence, start inside Snowflake, so they skip Layer 1, but Layers 2 to 4 still apply to them.

## Layer 1: TrustAI decides the tool call

Addresses: the yes-or-no flag (#1), unnamed external agents (#2), and one ceiling for every agent (#4)

An external agent reaches Snowflake through a tool call, so that's where the first decision belongs. The TrustAI gateway sits in front of the Snowflake-managed MCP server. It decides every call before Snowflake is queried, on which agent is asking, which user it's acting for, and whether the request's intent fits the agent's declared purpose. Snowflake can't see the first or the last.

In practice:

- **Each agent sees only the tools it qualifies for.** A tool without a policy allowing it is refused. Controls apply per agent, not one ceiling per user.
- **The intent check reads the tool's arguments, not just its name,** and compares them with the agent's declared purpose.
- **One action cuts an agent off every server it uses,** not just Snowflake.
- **Every decision is recorded** with the user, the agent, the tool, the arguments and the verdict. That names the external agent that Snowflake's audit doesn't.

**IN TESTING:**

a North America client management agent with a declared analytics purpose asked for recent customer feedback with the contact who submitted each entry. TrustAI refused the call before Snowflake was queried. The same agent then asked for a count of feedback entries by sentiment, and TrustAI allowed it. Same agent, same user, same tool. Only the intent differed: an aggregate count fits an analytics purpose, and a list of named contacts doesn't.

The decision record shows the reasoning (see **Image 2**). It includes the user's entitlements from Entra ID (market, account, groups), the agent's attributes (user, purpose, whether human context is required), the tool's attributes, and the purpose check's verdict.

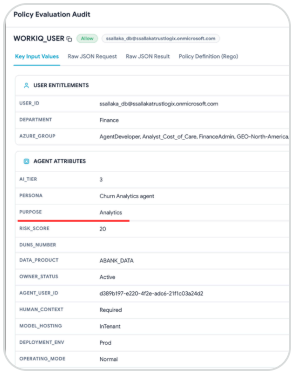



**Image 2:** The TrustAI decision record for the allowed call: the person's entitlements (market, account, groups) and the agent's attributes (persona, purpose, human context required).

## Layer 2: the user stays in the Snowflake session

Addresses: the user lost behind a shared service user (#3)

What Snowflake sees depends on how the gateway signs in. With **on-behalf-of**, the gateway exchanges the user's token for one Snowflake accepts, so the query runs as the user. Snowflake already marks every session through its managed MCP server as agent-active, so both halves of dual attribution arrive with no extra setup.

We checked what Snowflake recorded for two gateway registrations:

| Gateway registration | Session user | User type | Agent-active | `agent_type` |
|---|---|---|---|---|
| **On-behalf-of** | The signed-in user | `PERSON` | Yes | `CORTEX_AGENT` |
| **Shared key pair** | Service user | `SERVICE` | Yes | `CORTEX_LITE_AGENT` |

With on-behalf-of, Snowflake has the user and the agent flag in the same session, which is exactly what agent-aware policies need. With a shared key pair, Snowflake sees an agent but not the user, so policies keyed on the user stop working. **For dual attribution, the registration has to run as the user.**

Neither row names the external agent that made the call. That name exists only in the TrustAI decision record.

## Layer 3: TrustAccess writes the agent branch into native policy

Addresses: hand-written, one-off SQL (#5), and Snowflake's own agents that never pass the gateway

Layer 1 covers only agents that come through the gateway. Snowflake's own agents run inside Snowflake, so the last control has to sit in the data itself.

TrustAccess already compiles policy into native Snowflake row access and masking policies, driven by the user's attributes from your identity provider. It can also branch on Snowflake's agent flag today, using its existing support for custom functions. A one-line function, `TLX_IS_AGENT_SESSION()`, reads `IS_AGENT_ACTIVATED`; once it's added under Functions in TrustLogix, it appears in the condition builder like any other condition.

It fails safe: if the flag can't be read, the function treats the session as an agent, so data is masked rather than revealed. The policy is written in TrustAccess like any other: unmask the contact name only when the caller is the entitled user **and** the session isn't an agent session.

**IN TESTING:**

the same user, with the same role, ran the same SQL twice.

| Where the query ran | Agent flag | Contact name |
|---|---|---|
| **Snowsight worksheet** | FALSE | Bob Taylor |
| **Cortex Code** | TRUE | B**** |

Both returned the same single row, because the user's North America scope applied on both paths. The only thing that changed was who ran the query, and Snowflake knew.

**The user keeps their access, and the agent gets less.** Because it's a TrustAccess policy, it's managed with the rest of your policies instead of living as one-off SQL on a table (as illustrated in **Image 3**). TrustDSPM helps decide where to apply it by reading your existing classification tags to show which columns hold personal or regulated data.

(Implementation details for `TLX_IS_AGENT_SESSION()` are in the appendix.)

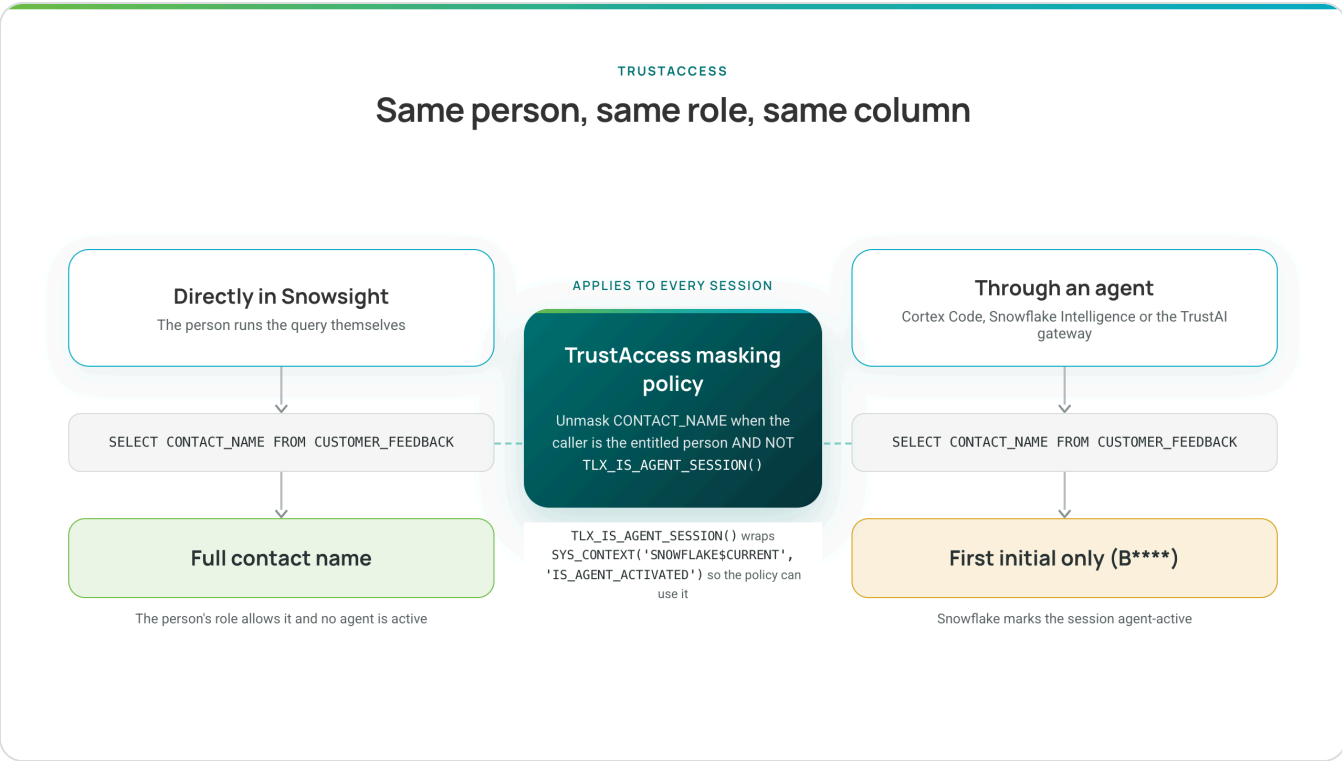



**Image 3:** One person, two answers. The same role sees contact names directly and masked names through an agent.

## Layer 4: Guardian Agent watches what agents actually do

Addresses: gaps between the audit records

Prevention is half the job. The other half is knowing what agents did. Three records each answer a different question:

| Record | What it tells you |
|---|---|
| **TrustAI decision record** | Which external agent asked, for which user, for what, and what was decided |
| **Snowflake audit** | What ran inside Snowflake: session user and role, agent type and, for Snowflake's own agents, which agent ran and which objects and columns it touched |
| **Guardian Agent** | Risk findings built from Snowflake's own logs, including Cortex agent usage history, agent query tags and AI observability events |

Read together, they cover each other's gaps. Snowflake can tell you a Cortex agent touched a column, but not which external agent asked it to. The gateway can tell you which external agent asked, but not which columns Snowflake read to answer.

In our tenant, for example, Guardian Agent flagged two Cortex agents running as the same shared service user and reading usable data columns across several schemas. It mapped the finding to the compliance frameworks it affects.

## Supporting Modern AI Agent Security with TrustLogix and Snowflake

Snowflake and TrustLogix don't do the same job. Snowflake governs what happens inside the session. TrustLogix governs what happens before the session starts and across the other systems an agent uses, then puts that decision into policy Snowflake enforces.

| Question | Snowflake answers | TrustLogix adds |
|---|---|---|
| **Is an agent acting?** | Yes: the agent-active flag | Uses the flag in centrally managed policy |
| **Which agent?** | Snowflake's own agents only | Names each external agent that comes through the gateway |
| **For whom?** | The session user | Keeps the user in the session with on-behalf-of |
| **Why, and does it fit?** | No | Checks each request's intent against the agent's declared purpose |
| **What can the agent see?** | Native masking, row access and Restricted Session Scope | Writes agent-aware masking and row access policy, without hand-written SQL |
| **What did the agent do?** | Query and access history | Combines Snowflake's logs with gateway decisions in Guardian Agent findings |
| **Beyond Snowflake?** | No | One gateway and one kill switch across every MCP server the agent reaches through it |

### One request through all four layers:

1. The agent asks the TrustAI gateway for a tool. TrustAI checks the agent, the user and the intent against the declared purpose, and records the decision.
2. The gateway calls the Snowflake-managed MCP server as the user. Snowflake marks the session agent-active.
3. TrustAccess policy inside Snowflake applies the user's row scope and masks the columns agents shouldn't see.
4. Snowflake's audit, the TrustAI decision record and Guardian Agent each keep their part of the record.

Enterprises won't run one kind of agent. They'll run Snowflake's own Cortex agents next to external agents built in Foundry, Copilot Studio or their own code, and those agents will reach Snowflake alongside Databricks, Salesforce and GitHub. Snowflake's agent identity is the right foundation inside Snowflake: it knows when an agent is in the session and lets policy act on it. TrustLogix carries that to the rest of the picture. It decides each external agent's tool calls on who the agent is, who it acts for and why, keeps the real user in the Snowflake session, writes the agent branch into native Snowflake policy, and turns Snowflake's agent logs into risk findings. Snowflake enforces the result."

## Decide at the tool call. **Enforce in the data.**

Snowflake knows when an agent is in the session. TrustLogix adds which agent, for whom and why, and turns that into policy Snowflake enforces.

# Appendix

## Under the hood: the `TLX_IS_AGENT_SESSION()` function

TrustAccess builds a policy from conditions: the person's attributes, the data's attributes, and a list of functions it can call. That list includes Snowflake's own functions, such as `CURRENT_ROLE()` and `IS_ROLE_IN_SESSION()`, and any custom functions you add from your account. It doesn't include `SYS_CONTEXT`, so the agent flag can't be picked directly. The answer is the same one we already use for things like a user's market area: put the fact in a small function, and let TrustAccess use the function.

Here's the whole thing:

SQL

```sql
CREATE OR REPLACE FUNCTION CLIENT_MANAGEMENT_DB.ENTERPRISE_ACCESS.TLX_IS_AGENT_SESSION()
RETURNS BOOLEAN
COMMENT = 'TRUE when Snowflake marks the session agent-active. For TrustAccess agent
branches.'
AS
$$
  COALESCE(TRY_TO_BOOLEAN(TO_VARCHAR(SYS_CONTEXT('SNOWFLAKE$CURRENT',
  'IS_AGENT_ACTIVATED'))), TRUE)
$$;

-- TrustLogix deploys the policy, so its control plane role needs to see the function
GRANT USAGE ON FUNCTION CLIENT_MANAGEMENT_DB.ENTERPRISE_ACCESS.TLX_IS_AGENT_SESSION()
  TO ROLE <trustlogix_control_plane_role>;

-- every role that queries the table evaluates the policy, so it needs the function too
GRANT USAGE ON FUNCTION CLIENT_MANAGEMENT_DB.ENTERPRISE_ACCESS.TLX_IS_AGENT_SESSION()
  TO ROLE <analyst_role>;
```

### What each part does, and why:

1. `SYS_CONTEXT('SNOWFLAKE$CURRENT', 'IS_AGENT_ACTIVATED')` reads Snowflake's agent flag for the session running the query. It's TRUE for Cortex Agents, Cortex Code, Snowflake Intelligence, the managed MCP server, `IS_AGENTIC` OAuth integrations and `SERVICE_AGENT` users.

2. `TO_VARCHAR` **then** `TRY_TO_BOOLEAN` cope with both return types. The flag comes back as the text 'TRUE' or 'FALSE' today, and as a real BOOLEAN once Snowflake's 2026_06 behaviour change is on. Our account already returns BOOLEAN, so this matters.

3. `COALESCE(..., TRUE)` makes it fail safe. Snowflake documents the flag as always TRUE or FALSE, but if it ever can't be read, the function says "agent", so a masking policy masks rather than reveals.

4. **It isn't** `MEMOIZABLE`**.** A memoizable function can have its result cached, and this answer belongs to each session. Our other custom functions are memoizable because they look up a table; this one mustn't be.

5. **It takes no arguments.** That's the shape TrustAccess offers in its condition builder today.

6. **The two grants** are what make it work in practice. Without them the deploy fails with an unknown function error, or the query fails for the person running it.

Once the function exists, you add it in the TrustLogix console under Functions: pick the database, schema and function, and give it a name and a short description. I spent a day waiting for a scan to pick it up before finding that step, so don't wait, add it. It then appears in the condition builder next to the built-in functions. The agent branch is two condition rows:

| Condition | Operator | Value |
|---|---|---|
| Username | Equals | Current Logged in User |
| `TLX_IS_AGENT_SESSION()` | Not Equals | `true` |

The first row ties the policy to the person running the query. The second is the agent branch. I used "Not Equals true" rather than "Equals false" on purpose, because TrustAccess renders it as `NOT (TLX_IS_AGENT_SESSION())`, which is exactly the logic I want.

You can check the function yourself before any policy uses it. In a Snowsight worksheet it returns FALSE, because you're the one asking:

```sql
SELECT CLIENT_MANAGEMENT_DB.ENTERPRISE_ACCESS.TLX_IS_AGENT_SESSION();
```

Run the same statement through an agent, and it returns TRUE. We sent it through the TrustAI gateway to the Snowflake-managed MCP server, and Snowflake answered TRUE for the function and TRUE for the raw flag in the same row. So the function sees the caller's session, which was the part I wasn't sure about before testing it.

The same is true for Snowflake's own agents. Here's the check in a Snowsight worksheet, then the same query run by Cortex Code, by the same person with the same role:

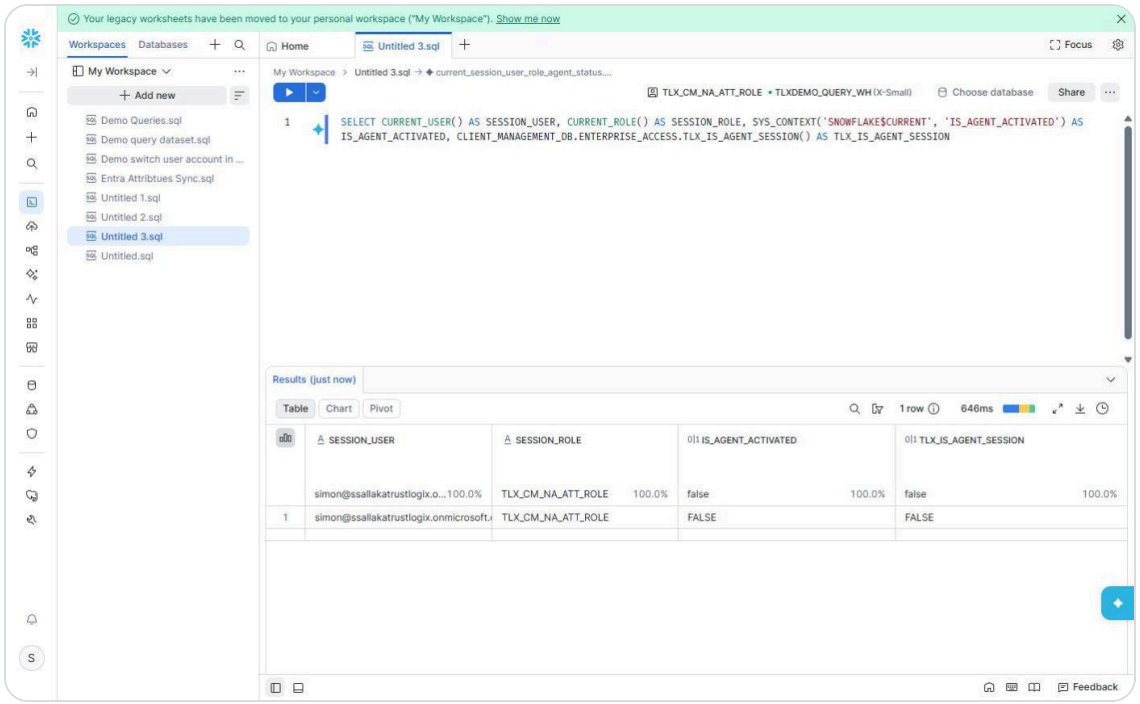



**Image 4:** In a worksheet: the person, their role, and FALSE for both the flag and the function.

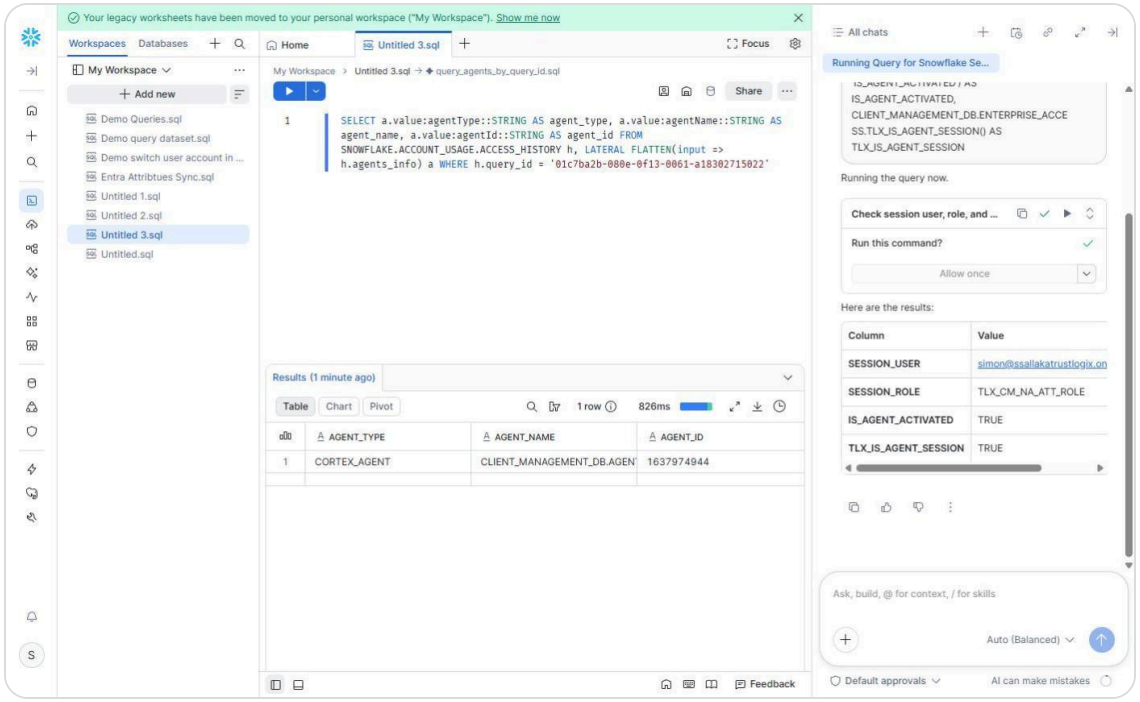



**Image 5:** The same query run by Cortex Code, same person, same role: TRUE for both.

With the function in the builder, I published the policy. TrustAccess compiled it into a native Snowflake masking policy and set it on the column, and it showed as deployed minutes later. This is what it created for the policy in TrustLogix:

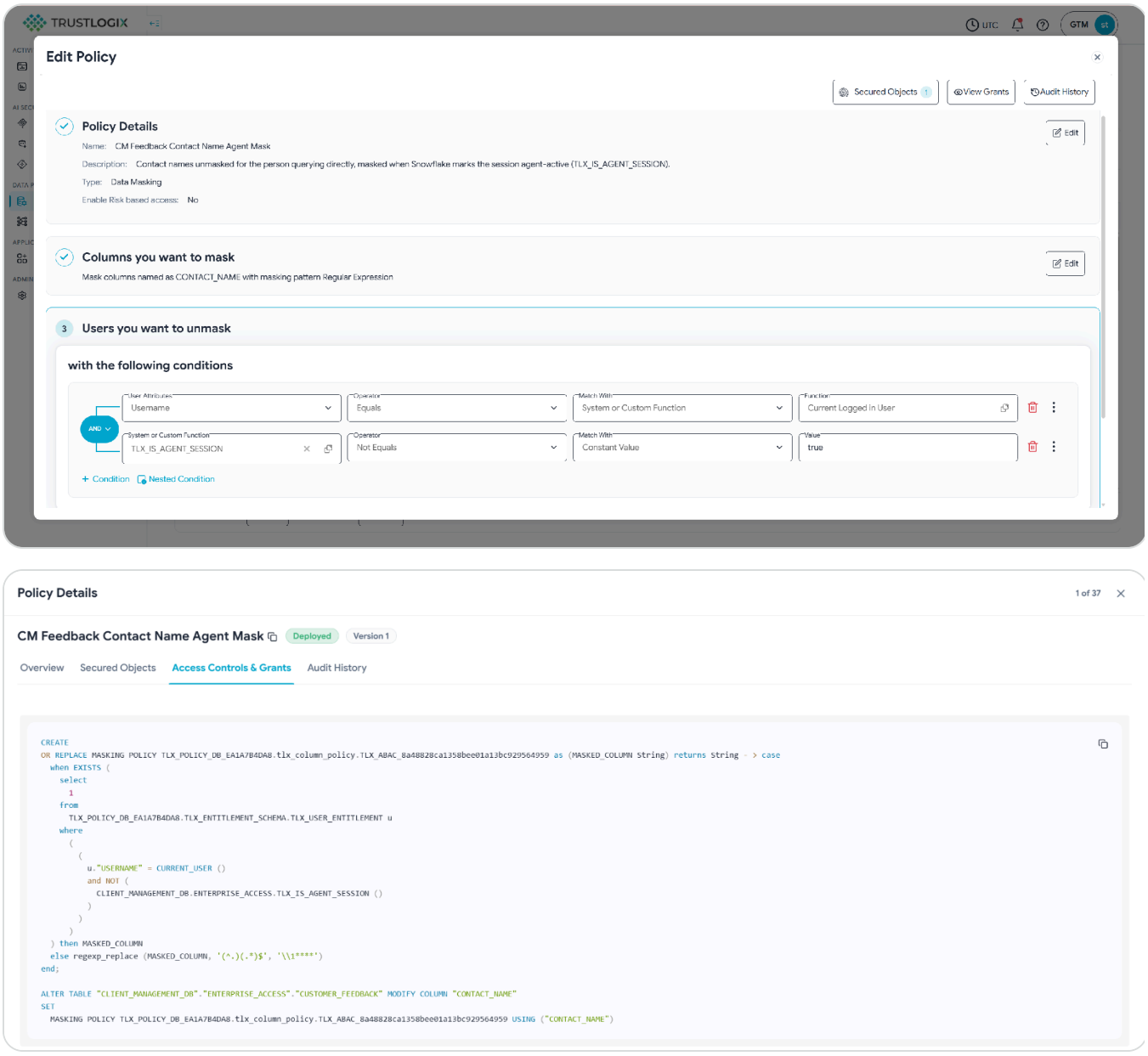



**Image 6:** The policy as TrustAccess deployed it. The real value comes back only when the person's entitlement row matches and the session isn't an agent session. Everyone else gets the first initial.

Then the test: the same person, the same role and the same SQL, once in a Snowsight worksheet and once through Cortex Code.

```sql
SELECT CURRENT_USER() AS session_user, CURRENT_ROLE() AS session_role,
       SYS_CONTEXT('SNOWFLAKE$CURRENT', 'IS_AGENT_ACTIVATED') AS is_agent_activated,
       feedback_id, customer_name, contact_name, feedback_type, sentiment
FROM CLIENT_MANAGEMENT_DB.ENTERPRISE_ACCESS.CUSTOMER_FEEDBACK;
```

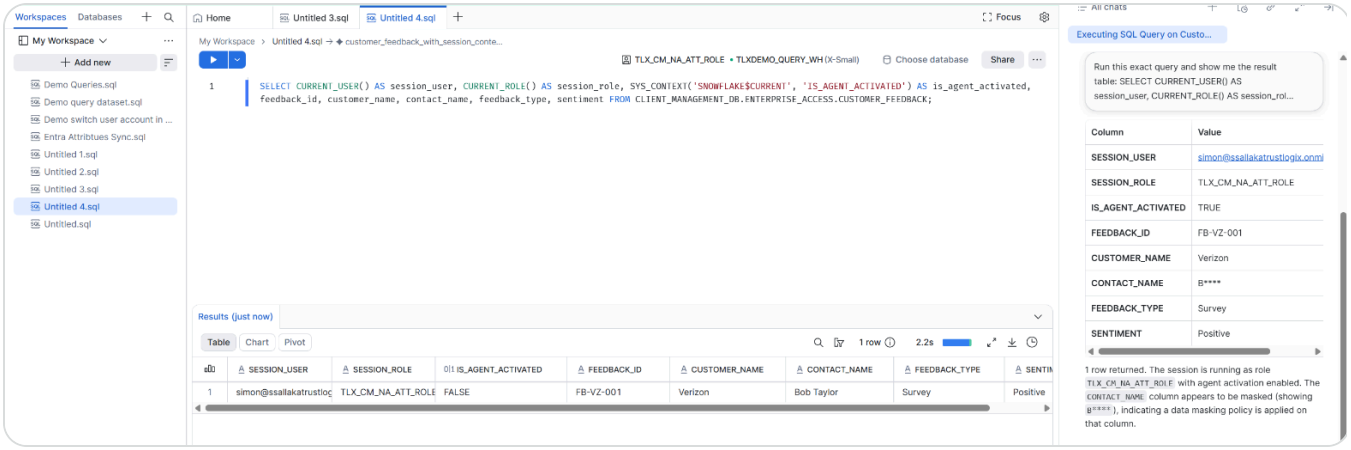



**Image 7:** Left, the worksheet: agent flag FALSE, contact name Bob Taylor. Right, Cortex Code running the same query: agent flag TRUE, contact name `B`.