

From Prompt to Query

The real cost of agents on Snowflake: attribution, budgets, guardrails and model choice, from the prompt to the warehouse

Amir Peres

Co-founder and CTO, Yuki



**Everything here exists in our product.
But tonight I show you how to build it yourself**

OPTION 1

Buy it

Everything in this talk is in our platform.
Running by tomorrow morning.



OPTION 2

Build it

The next 35 minutes: what Snowflake gives you, what you build on top, where you hit a wall.

ABOUT ME



Amir Peres

Co-founder and CTO, Yuki

Backend Developer

Data Product

Data Architect

amir@yukidata.com

 amir-peres



we're building the future of Data Platforms;

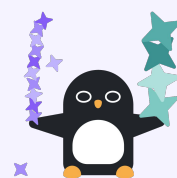
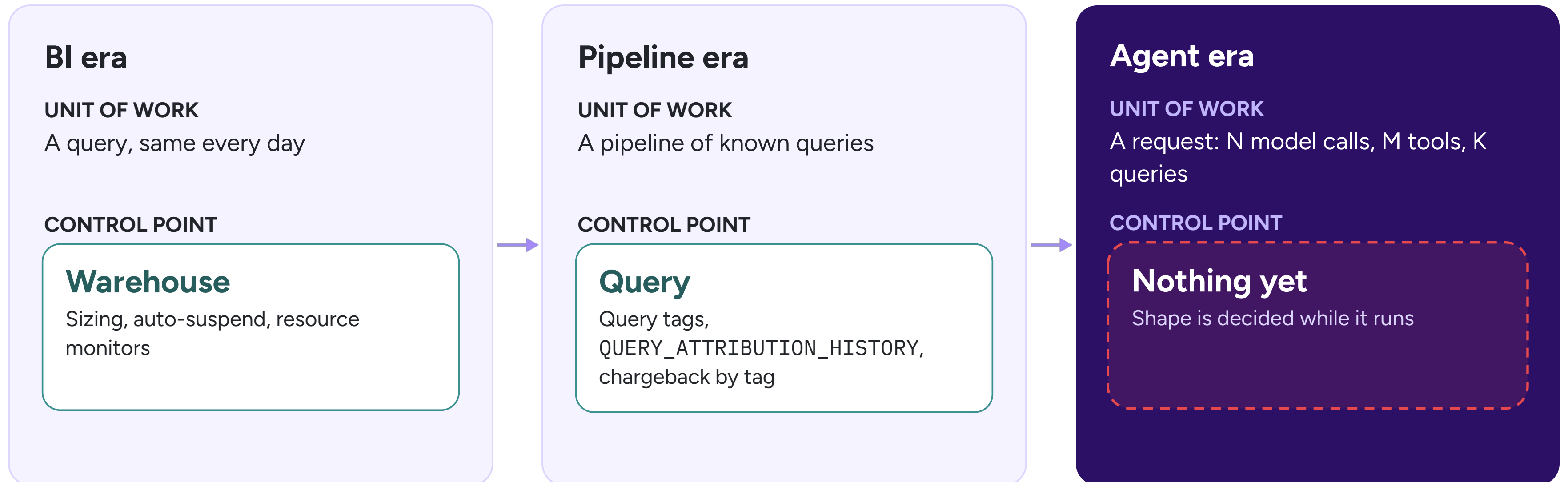


AGENDA

Five parts, 35 minutes, then questions

1	What agents changed	The unit of work is a request. Every run explores a different path.	
2	Cost and attribution	AI meters plus regular meters. Why end to end attribution breaks, and how to build it.	
3	Budgets and guardrails	What covers the whole journey. How to set a limit, adjust it, and handle exceptions.	
4	Models	Why the wrong model costs more. How to track, benchmark and roll out a new one.	
5	Putting it together	The reference architecture. What to build, and where the wall is.	

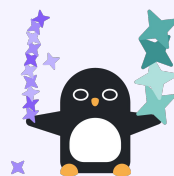
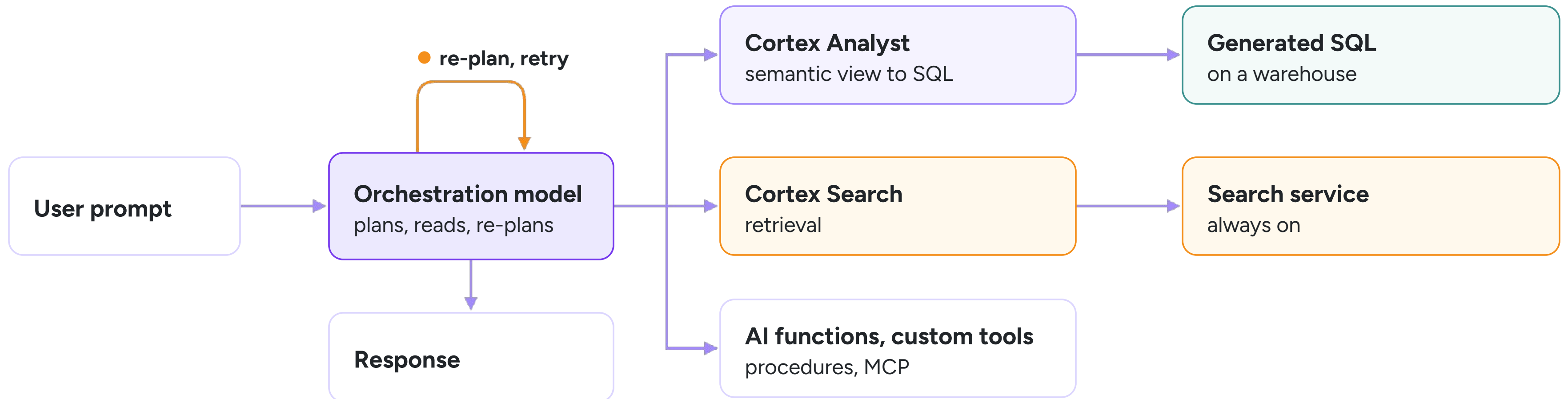
The control point moved from the warehouse, to the query, to the request



Warehouse controls and query tags were built for work you know in advance.

An agent decides how much work to do while it runs. Your tools do not have a control point there yet.

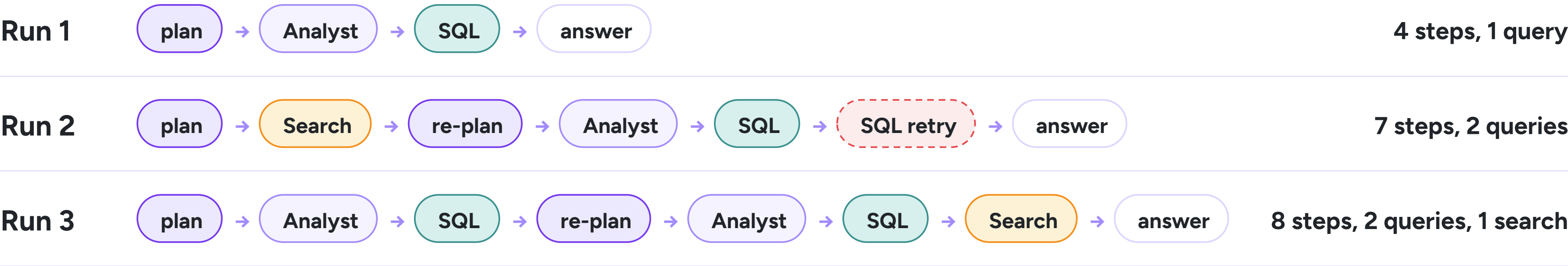
What happens inside one agent request



The model plans, calls tools, reads results, and decides if it needs another round.

The same user, same prompt, tomorrow: a different path. This is not a bug. This is what an agent is.

Agents explore: the same prompt can take a different path on every run



Same prompt, same user, three runs. Illustration.

● plan ● SQL ● search ● retry



You cannot size a warehouse, a token budget or a timeout for a path you do not know in advance.

Any static setup is right for the average run and wrong for the tail. The tail is where the money goes.

One request is billed by five AI meters and three regular meters

AI METERS, BILLED IN AI CREDITS

Orchestration tokens

per token, per model

Input, cached input and output priced apart

Cortex Analyst

per message or tokens

Per message standalone. Token based inside an agent

Cortex Search serving

per GB-month

Runs while the service is up, with zero queries too. Plus embedding tokens

AI functions, tools

per token

Only when the agent calls them

Cross-region inference

per token

When the model is not in your region

REGULAR METERS, BILLED AS ALWAYS

Warehouse compute

per second

The generated SQL, on whatever warehouse the tool points to

Search index refresh

warehouse compute

Materializing and refreshing the search data

Storage & cloud

per TB, per request

The index, the results, the metadata calls



Four different billing models in one request: per token, per message, per GB-month, per second.

This is why the finance question 'why did Cortex go up 40 percent' has no simple answer.

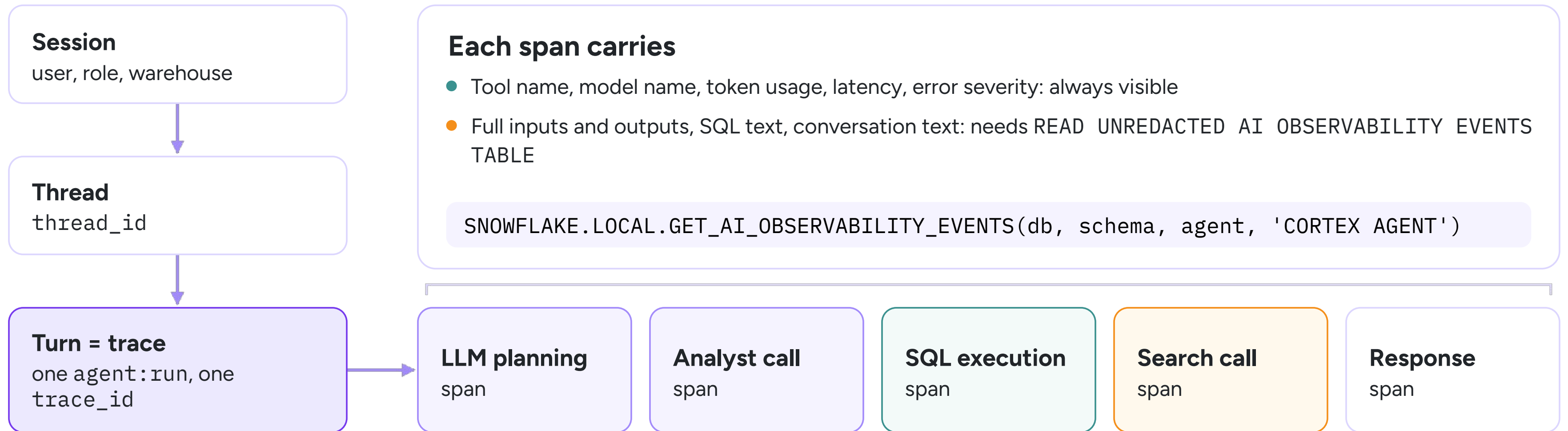
What each usage view knows about a request

VIEW	GRAIN	REQUEST ID	QUERY ID	LAG
CORTEX_AGENT_USAGE_HISTORY	One agent request	Yes, plus parent	Warehouse names, credit sum	1 h. SQL cost fields up to 8 h
AI_OBSERVABILITY_EVENTS	One turn = one trace, spans per step	trace_id, thread_id	SQL text in the span. Check for query_id	Minutes
CORTEX_ANALYST_USAGE_HISTORY	One Analyst message	Yes	No	Up to 2 h
CORTEX_SEARCH_SERVING_USAGE_HISTORY	One service, one hour	No	No	Hourly
QUERY_HISTORY, QUERY_ATTRIBUTION_HISTORY	One query, compute credits	Only if you set QUERY_TAG	Yes	45 min, 6 h
METERING_HISTORY	One hour, one service type	No	No	Up to 3 h



The agent view has start and end time per request, so latency per agent per model is a GROUP BY.
Lag values are the documented maximums. Adaptive Warehouse credits are not in the agent view's SQL cost fields.

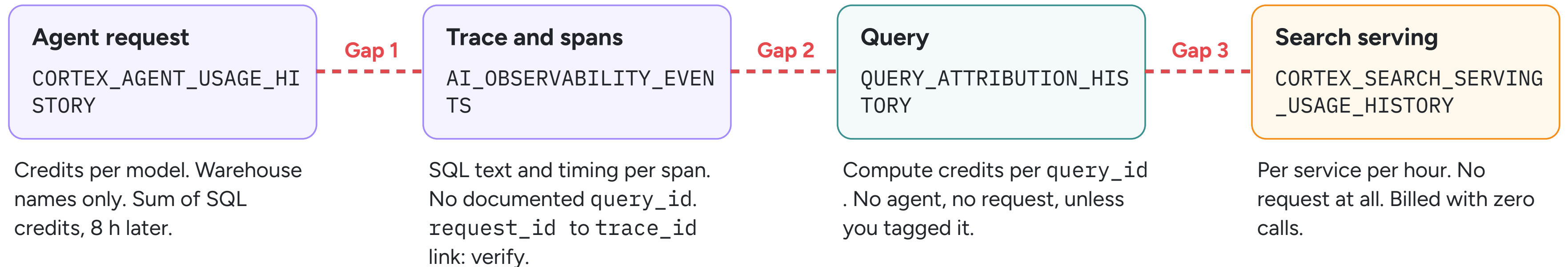
AI_OBSERVABILITY_EVENTS: one trace per turn, one span per step



This is where per-step latency and per-step tokens live. Grant MONITOR on the agent to read it.

Snowflake also records the agent version that served each turn. You will need that for the model rollout later.

Where end to end attribution breaks: three gaps and a lag



Result: nobody can answer "what did this prompt cost, end to end" from the views alone



Gap 1 and 2 need a key you create. Gap 3 can only be allocated, never attributed. And Adaptive Warehouse credits fall out of the agent view.

Three ways to close the request to query gap

A	Match by time window Same user, role, warehouse. Query start inside request start and end	Cheap. Breaks when two requests from the same user overlap	Control you keep
B	Match on the trace Span SQL text plus warehouse plus timestamp against QUERY_HISTORY	Much better. Still a match, not a key. Needs the unredacted privilege	Control you keep
C	Own the execution path Agent plans and writes SQL. Your layer runs it with QUERY_TAG = request_id	A real key. Every query carries request, agent, model, user. You pick the warehouse	Control you keep



Option C only works when the agent returns the SQL and your application executes it (REST API pattern).

When Snowflake hosts the interface (Intelligence, Teams), the agent runs the SQL itself. There you get A or B, not C.

The attribution query: credits per agent per model, with warehouse cost and P95

```
WITH agent AS (SELECT request_id, agent_name, token_credits,
                     DATEDIFF('ms', start_time, end_time) AS latency_ms
                FROM snowflake.account_usage.cortex_agent_usage_history),
    per_model AS (SELECT a.request_id, g.value:model::STRING AS model,
                        g.value:input::NUMBER + g.value:output::NUMBER AS
model credits
                FROM snowflake.account_usage.cortex_agent_usage_history a,
                     LATERAL FLATTEN(INPUT => a.credits_granular) f,
                     LATERAL FLATTEN(INPUT => f.value) g),
    sql_cost AS (SELECT query_tag AS request_id,
                       SUM(credits_attributed_compute) AS wh_credits
                FROM snowflake.account_usage.query_attribution_history
                GROUP BY 1)
SELECT a.agent_name, m.model, COUNT(*) AS requests,
       SUM(a.token_credits + COALESCE(s.wh_credits, 0)) AS credits,
       APPROX_PERCENTILE(a.latency_ms, 0.95) AS p95_ms
FROM agent a
     JOIN per_model m
         USING (request_id)
     LEFT JOIN sql_cost s
         USING (request_id)
GROUP BY 1, 2;
```



Needs option C

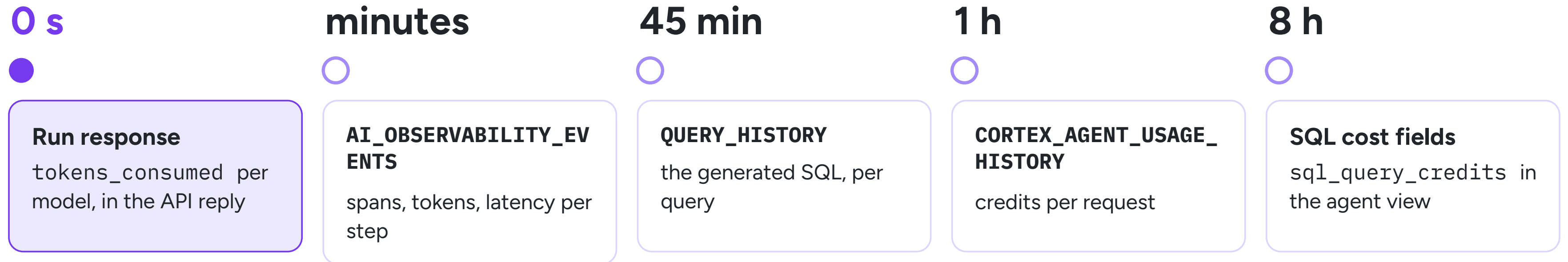
(QUERY_TAG = request_id).

With option B, replace sql_cost with the trace match.

Search serving is allocated per hour on top, not attributed. Verify the CREDITS_GRANULAR paths on your account.

You can know the cost at request time instead of waiting for the views

When do you know what a request cost?



The run response already returns tokens per model. At the application boundary you can count credits before the user sees the answer.
Everything to the right is for reconciliation and finance. Control needs the left side.

Four shapes of agent waste and the signal that shows each one

Retries and re-plan loops

NATIVE SIGNAL

spans per trace in `AI_OBSERVABILITY_EVENTS`.
`PARENT_REQUEST_ID` chains

WHAT TO DO

Count steps per turn per agent. A rising step count is the earliest cost signal you have

Failed generated SQL

NATIVE SIGNAL

`error_code` on the SQL execution span and on the matched query

WHAT TO DO

Usually the semantic view, not the model. Fix the view, the retries disappear

Oversized warehouse for agent SQL

NATIVE SIGNAL

`sql_query_warehouses` vs `bytes_scanned` per matched query

WHAT TO DO

Agent SQL is mostly small aggregates. Give the Analyst tool its own small warehouse

Search serving with no callers

NATIVE SIGNAL

serving hours with zero requests in
`GET_AI_OBSERVABILITY_EVENTS` for the service

WHAT TO DO

Suspend the service off hours or use Batch Search for jobs. Easiest win in this room

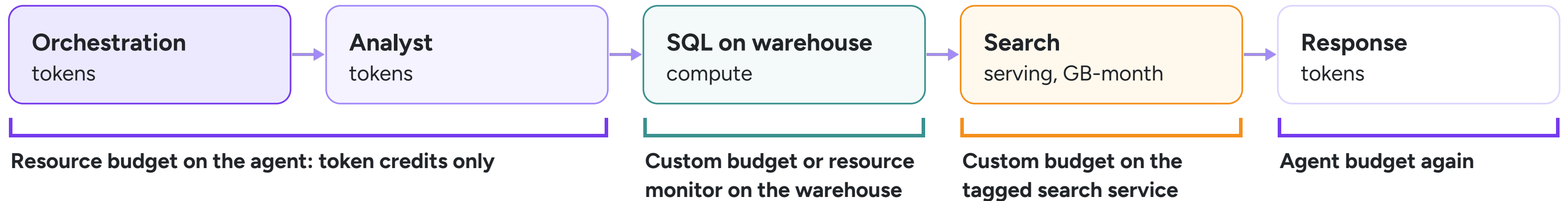
Native budget objects: what each one covers, and what it does not

OBJECT	SCOPE	CYCLE	ACTION	AGENT TOKENS	WAREHOUSE SQL
Account budget	Whole account	Monthly	Notify	Yes, in total	Yes, in total
Custom budget	Tagged objects, warehouses	Monthly	Notify	If the agent is tagged	If the warehouse is tagged
Resource budget on an agent	One tagged Cortex Agent	Monthly	Stored procedure at 80 / 100 %	Yes	No
Shared resource budget	Users by tag, AI features	Monthly	Stored procedure	Yes, per user group	No
Per-user quota	Users in scope, AI domains	Monthly, weekly, daily	Hard block, minutes later	Yes, per user	Only if warehouses added, no auto-block
Resource monitor	Warehouse or account	Any cycle	Suspend the warehouse	No	Yes, all users of it
Daily per-surface limit	CoCo only, per user	Rolling 24 h	Block	CoCo only	No



No single object covers tokens, the generated SQL, and search serving for one agent. Each one sees a piece.
All of them act after the spend, on a monthly or daily cycle. None sees a request.

A budget that covers the whole journey does not exist natively, and the workaround has a price



Three budgets, three cycles, nobody sums the journey

The agent budget stops at the tokens. The warehouse budget counts every user of that warehouse, not this agent. Search serving is shared by everything that calls the service.

The workaround, and its price

Give the agent its own warehouse and its own search service. Put one cost-center tag on all three. One custom budget on the tag now sees the whole journey.

Price: a warehouse per agent, an index per agent, still monthly, still after the fact.

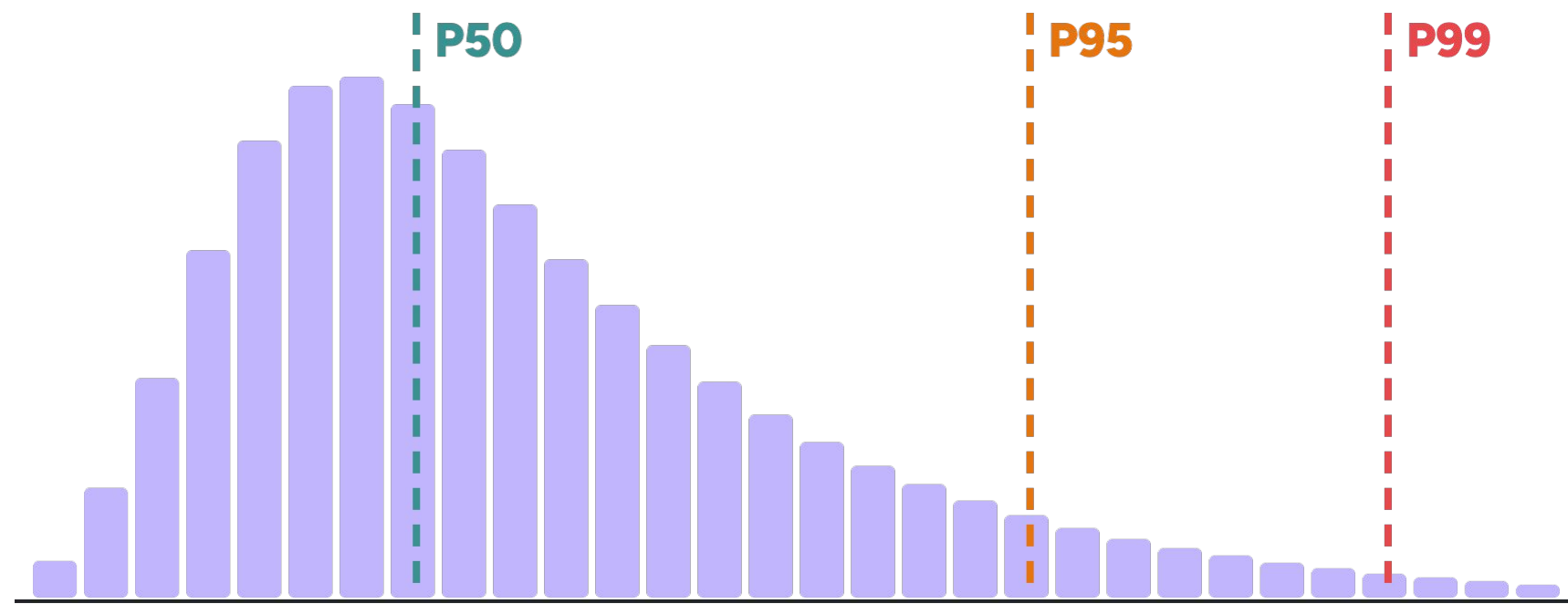
Per request guardrails Snowflake gives you today, and which step each one sees



Whole request: the REST API stops at 15 minutes. Per user: quotas and daily limits, after the spend.

 Each guardrail sees one step. The token budget does not count Analyst or Search tokens. The query timeout does not know how many queries the run will make.

How to choose a limit: measure first, then set it at the tail of the good answers



orchestration tokens per request, one agent, two weeks →

Illustration, not customer data.

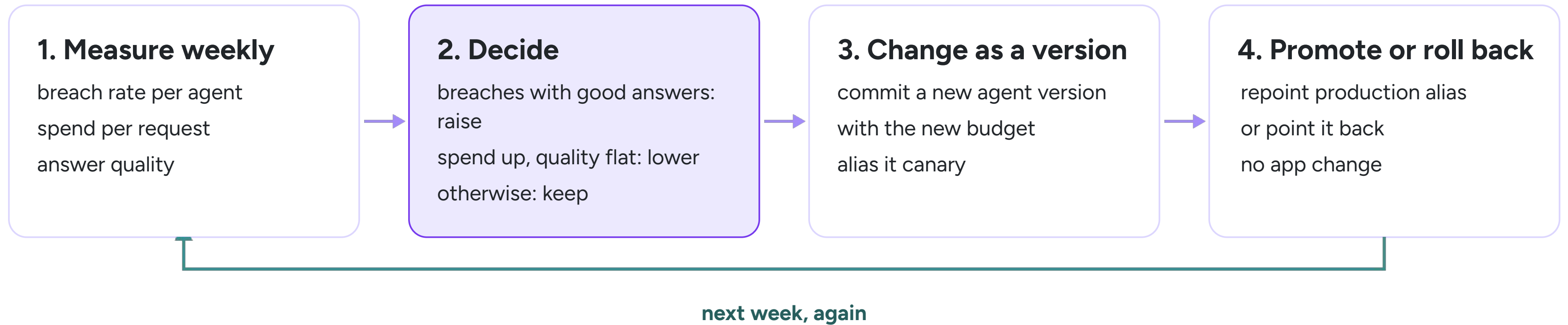
How to pick the number

- 1 Two weeks in shadow: no limit, full attribution. One distribution per agent, tokens and seconds.
- 2 Keep only the runs with a good answer (feedback, evaluation score). Take their P95. That is the starting limit.
- 3 Set it per agent, in the agent spec. Never one number for the account.
- 4 Write down what happens at the limit: a partial answer is a product decision, not a config value.



A limit at the P50 blocks half of the honest deep questions. A limit at the P99 lets most of the waste through.

How to adjust a limit over time without breaking production



Breach rate is the number to watch: how many runs hit the limit, and whether those runs would have been good answers.

Agent versions give you a rollback for a budget change, the same way they do for a model change.

Special users and special weeks without rebuilding the whole setup: three layers of policy

LAYER 1

Default per agent

NATIVE

`orchestration.budget` and tool timeouts
in the agent spec. One number per agent.

THE CATCH

One number for every user of the agent.

LAYER 2

Tier per user group

NATIVE

Two quotas: Standard and Power, scoped by
user tag. Same for shared resource budgets.

THE CATCH

Tag changes are DDL. No per-request
context, only who.

LAYER 3

Time-boxed exception

NATIVE

Earnings week: budget stored procedure
reinstates access. Or a version with a bigger
budget behind a temporary alias.

THE CATCH

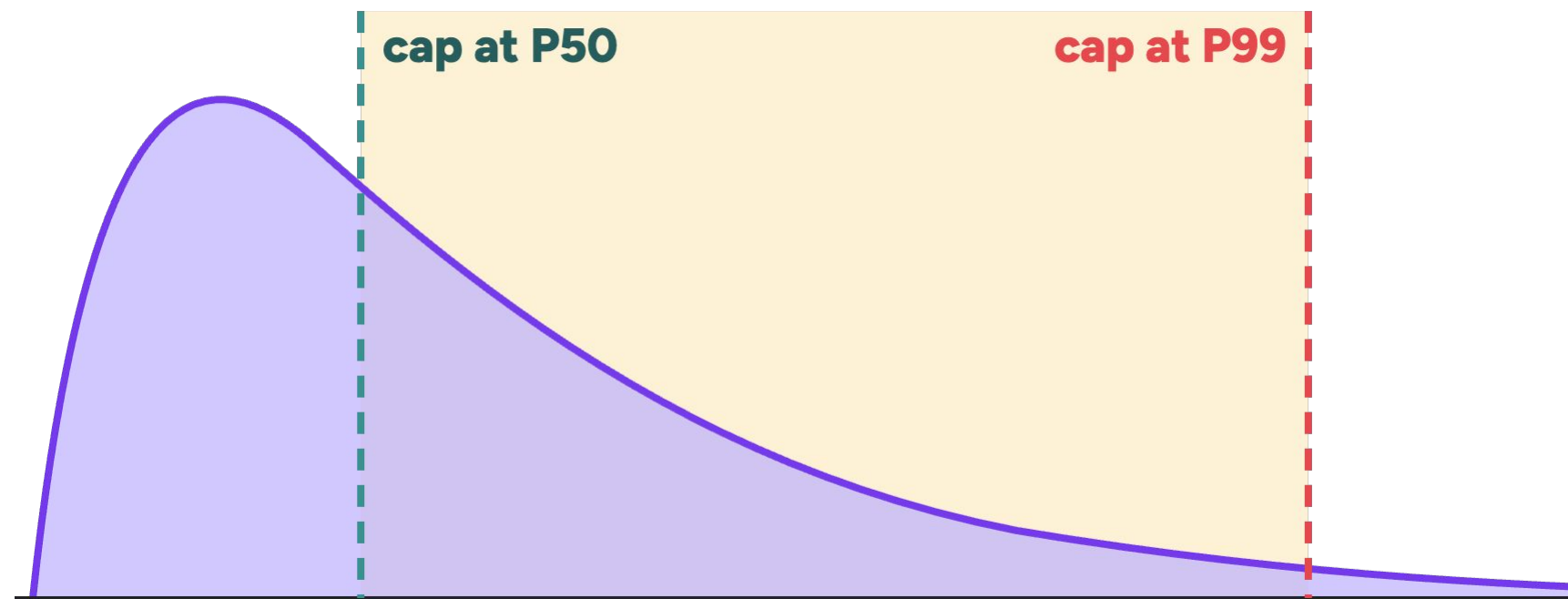
Nothing expires by itself. Someone must
remember to turn it off.



Design the layers first, then map each to a native object. If you start from the objects you end up with one limit for everybody.

What none of the layers can do: look at this request, this user, this hour, and decide.

Why a static limit fights exploration, and what a limit needs to know to be right



Honest deep questions and waste live in the same zone. A number cannot tell them apart.

Illustration.

What a limit needs to know to be right

Who asks, what they asked, which agent, how much budget is left this hour, what SLA this caller has.

And what it can do instead of blocking

Use a smaller model. Use a smaller warehouse. Cap the steps. Ask the user before the expensive branch.



This is a decision per request. Snowflake has no object that sits there.

The model changes the price per token, the number of steps, and how often the SQL is wrong

WHAT CHANGES	SMALL MODEL	LARGE MODEL
Price per token	Low	10x or more higher, see the Service Consumption Table
Planning quality	More re-plans, more tool calls	Fewer steps to a good plan
Tool call accuracy	More wrong tool, more bad SQL	Better SQL, fewer retries
Latency per step	Faster	Slower
Latency per answer	Often slower: more steps	Often faster: fewer steps
Cost per correct answer	Can be higher	Can be lower



Price per token is the number everybody looks at. Cost per correct answer is the number that matters, and it moves the other way more often than you think.

The orchestration model is one line in the agent spec. Changing it changes the whole path.

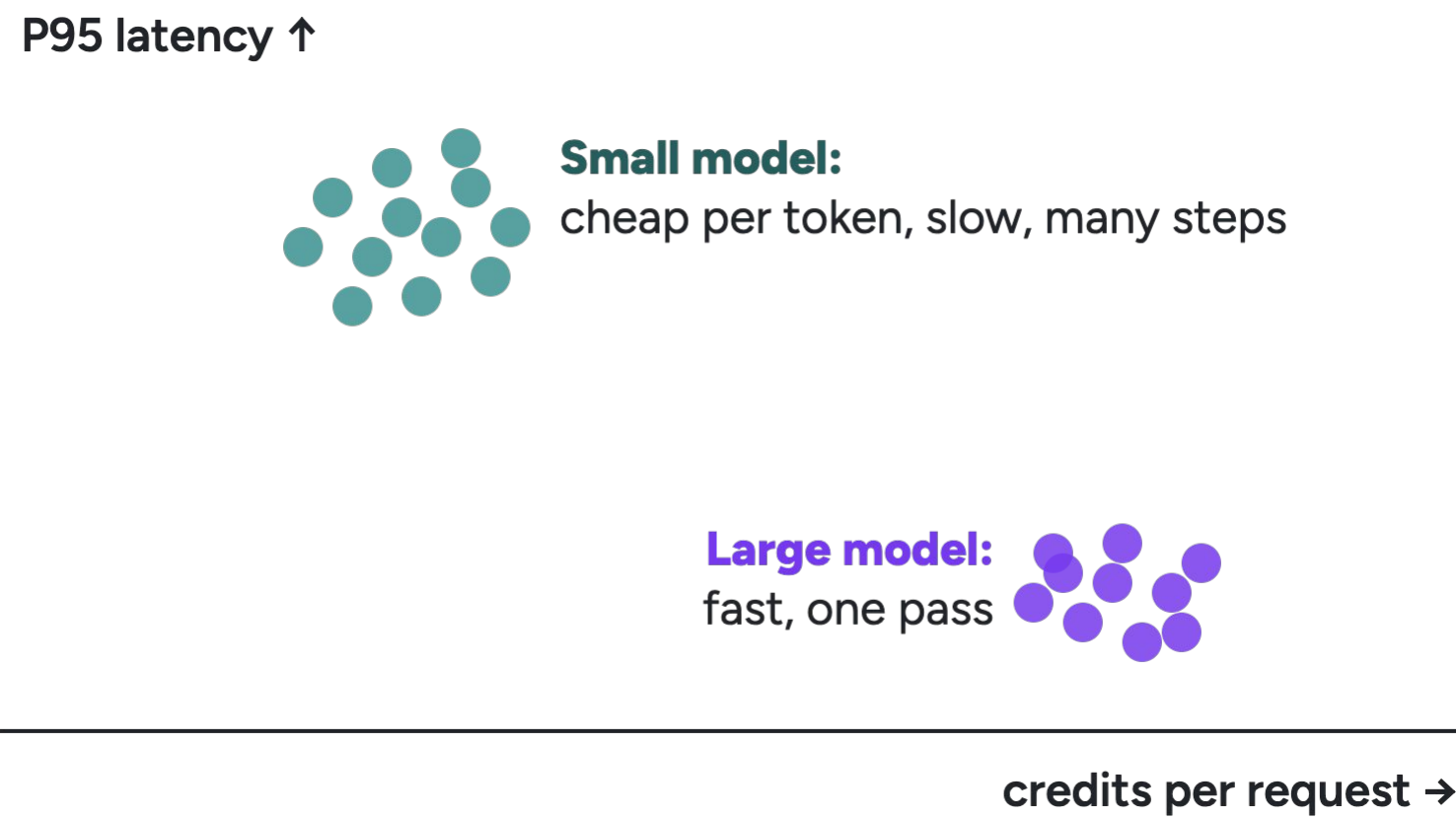
Symptoms of the wrong model and where you see each one

SYMPTOM	SIGNAL	WHERE
Re-plan loops, long turns	Spans per trace rising. PARENT_REQUEST_ID chains	AI_OBSERVABILITY_EVENTS, agent usage view
Wrong tool chosen	Search span where Analyst was needed, or the reverse	Trace spans, tool name
Bad SQL, retries	error_code on the SQL span and the matched query	Trace spans, QUERY_HISTORY
Slow answers	Latency per span, P95 per model	Trace spans, START_TIME to END_TIME
Users unhappy	Thumbs down rate per agent version	CORTEX_AGENT_FEEDBACK events
Quality drops after a change	Evaluation score per version	Cortex Agent evaluations



Every signal here is per agent version. If you do not version, you cannot tell a model change from a semantic view change.

What to track per agent, per model, per version: six numbers on one dashboard



Credits per request agent view + query attribution	P95 latency START_TIME to END_TIME
Steps per turn spans per trace	SQL error rate SQL span error_code
Thumbs down rate CORTEX_AGENT_FEEDBACK	Evaluation score agent evaluations



Left: cost next to latency, per model. Most teams never saw this picture for their own agents. Illustration, not customer data.
Right: the six numbers, and the native source of each.

Benchmark before you switch: a golden set, one agent version per model, one evaluation run each

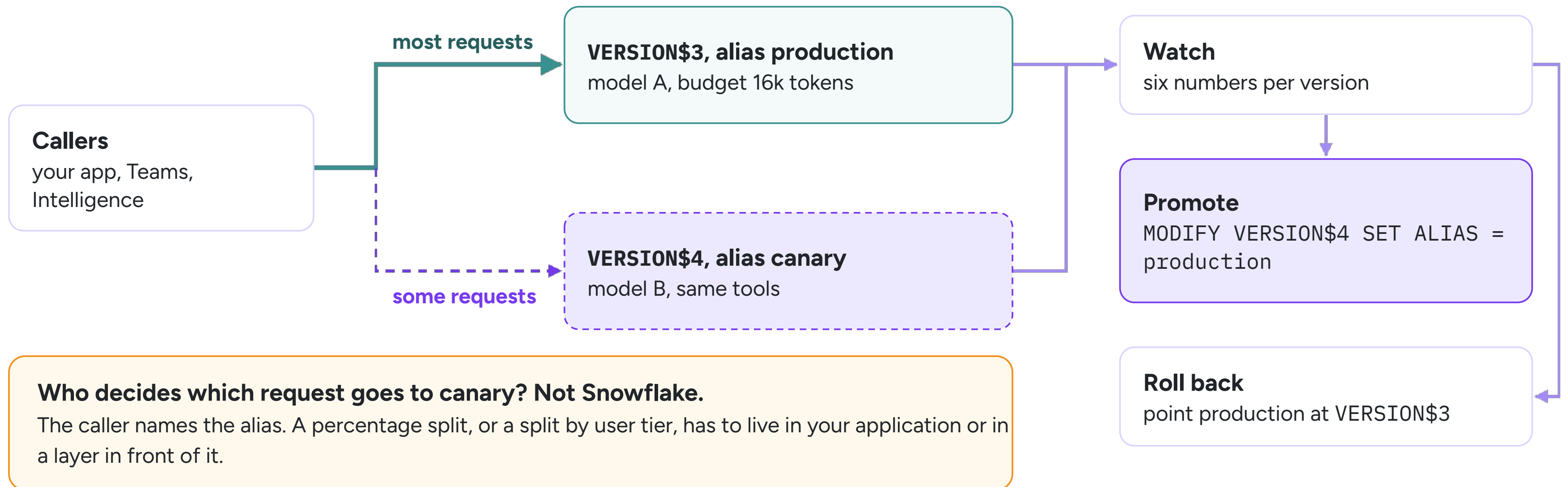


The judge costs credits too: every metric on every response is a model call. Budget the benchmark like any other agent workload.
Pin committed versions (`VERSION$N`) for runs you want to compare over time. An alias can move under you.

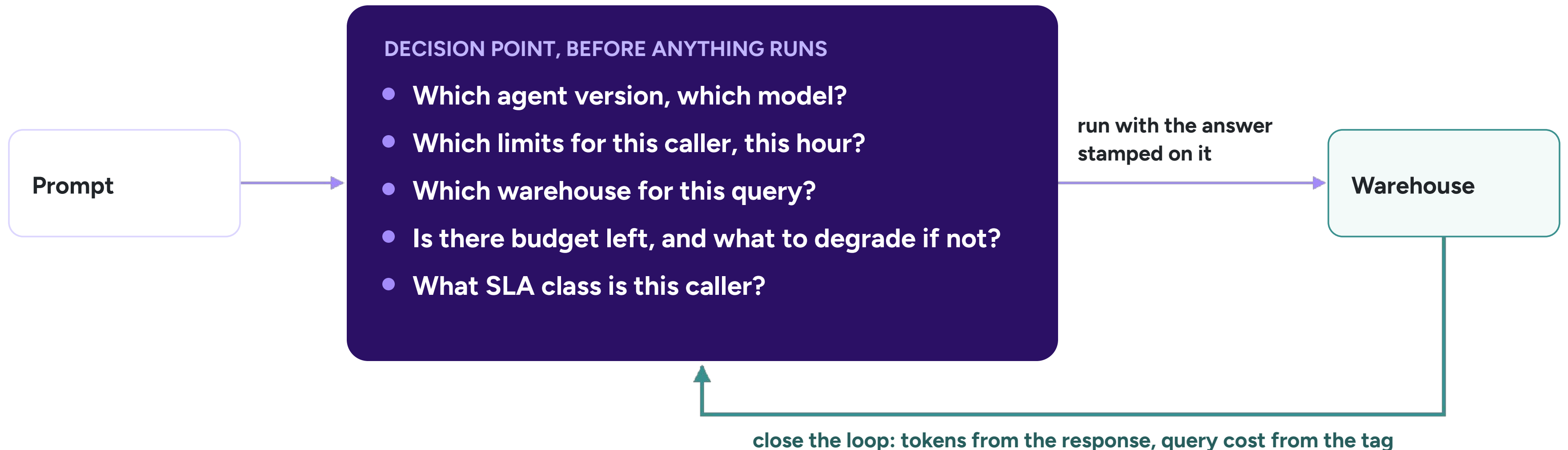


Cost per correct answer = credits per run divided by the share of correct answers. That is the number to compare.
Repeat the same runs whenever a new model or a new semantic view lands.

Roll out a new model gradually with agent versions and aliases, and roll back with one statement

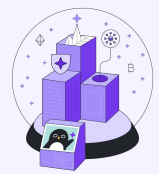
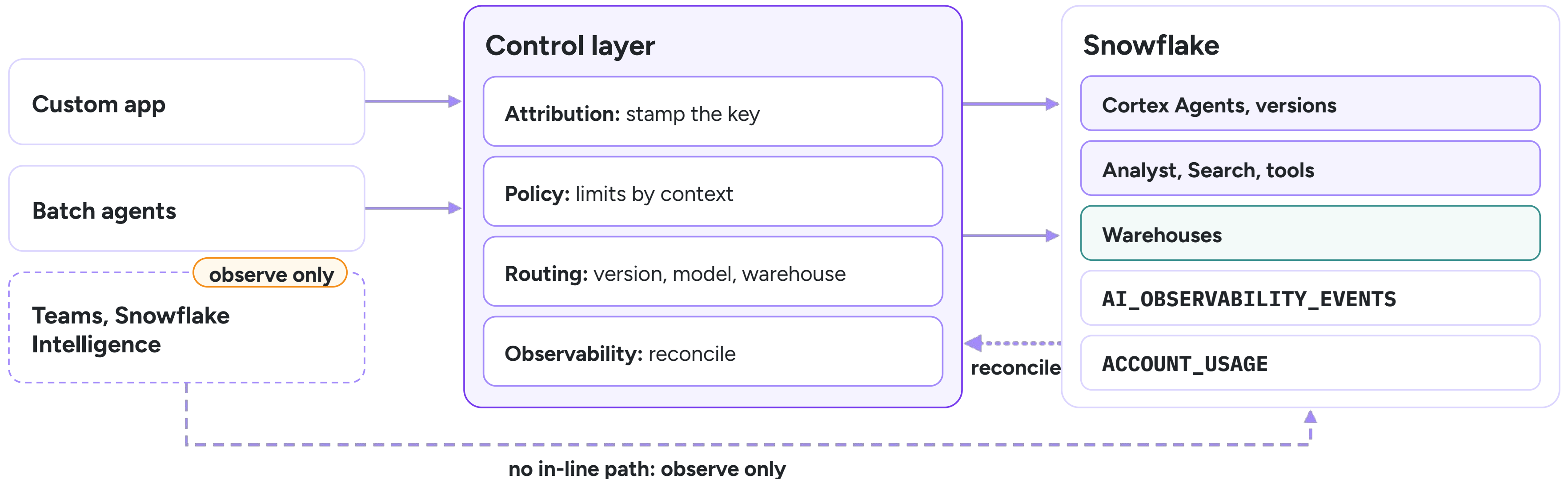


Limits, models, warehouses, exceptions: every open question in this talk is a decision at the request boundary



Attribution needs the key stamped here. Guardrails need the context here. Rollouts need the split here.
Snowflake has no object that sits here. Your application can, if it owns the execution path.

Reference architecture: a control layer between the applications and Snowflake, with telemetry flowing back



Everything on this picture was covered tonight. Most teams build attribution and observability in two or three sprints, then stop before the in-line half.

What to build natively, and where you hit the wall

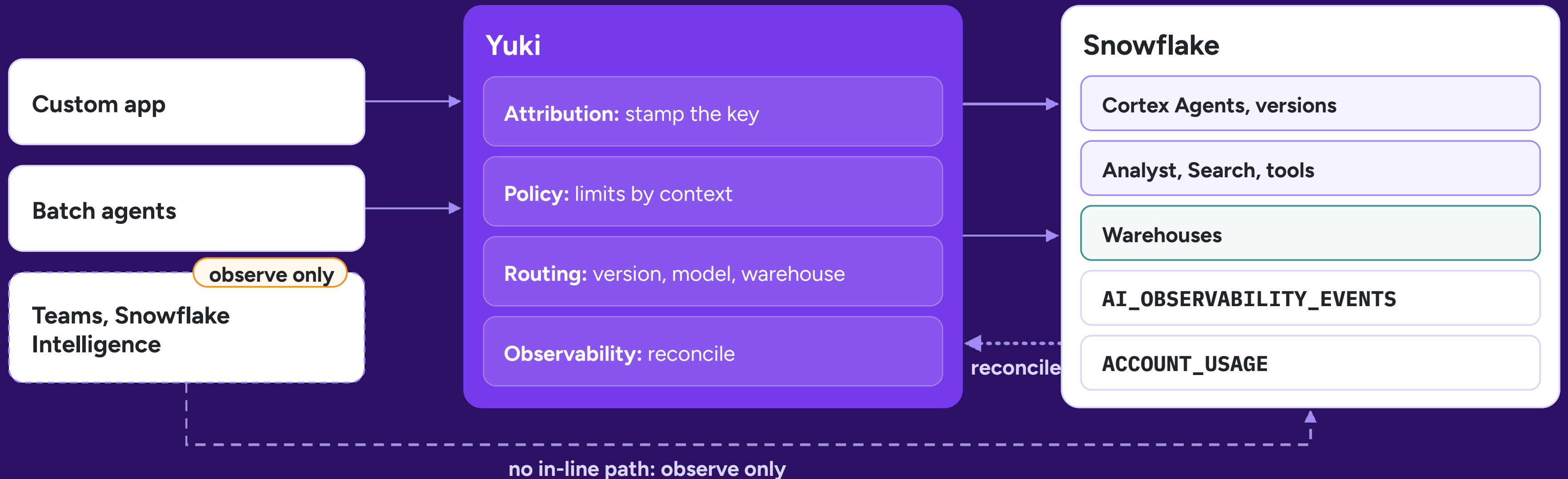
Build it natively. You should.

- ✓ Attribution dashboard: agent view + traces + query attribution
- ✓ Six numbers per agent, per model, per version
- ✓ Waste report: steps, failed SQL, idle search
- ✓ Orchestration budgets, tool timeouts, quotas: turn them on
- ✓ Versions and aliases for every change
- ✓ Golden set and evaluations before every switch

The wall

- ⚠ A budget that sums tokens, SQL and search per request
- ⚠ A limit that knows who, what, and budget left
- ⚠ Degrade instead of block: smaller model, smaller warehouse
- ⚠ Canary by percentage or by user tier
- ⚠ Decisions in milliseconds, not one to eight hours
- ⚠ All of it with zero agent code changes

The control layer between your applications and your data platforms



Sees every request before it runs. Stamps the key, applies the policy, picks the version and the warehouse, reconciles with the views.

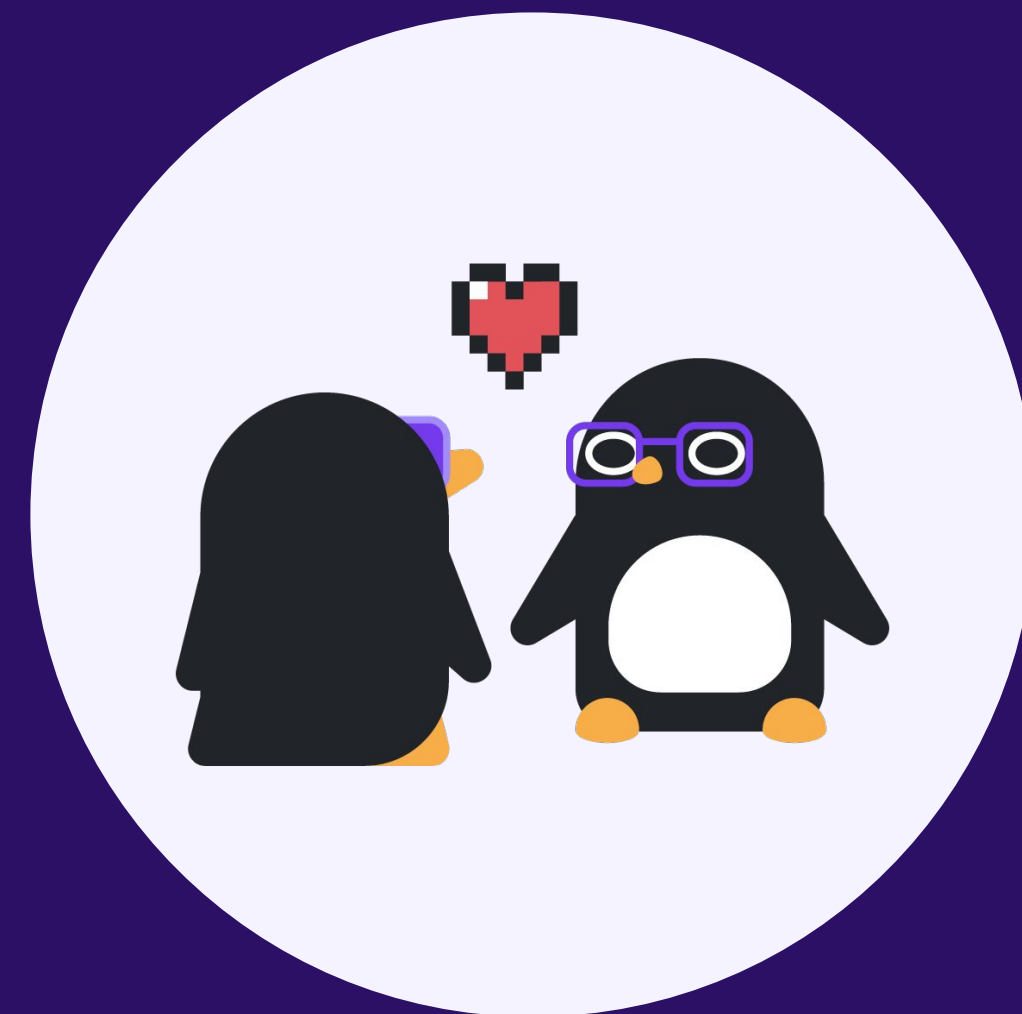
Runs inside your environment or hosted. No changes to the agent code.



Build it, or don't.

If you build it, tell me what broke. If you don't, let's talk.

Cortex Control Center: free for a limited time



amir@yukidata.com

 amir-peres