

Two queries can agree today and disagree tomorrow

That is the difference between output equivalence and real query equivalence.

Query equivalence

Same result for every valid input.

```
WHERE x BETWEEN 1 AND 5
```

```
WHERE x >= 1 AND x <= 5
```

Any x, any table: the two are identical.

Output equivalence

Same result for this data only.

```
WHERE rating > 4.7
```

```
WHERE rating >= 4.9
```

Tomorrow a row rates 4.8:

> 4.7 keeps it, >= 4.9 drops it.

Do These Two Queries Mean the Same Thing?

One valid counterexample disproves equivalence. To prove it instead, explain why both queries return the same result on every valid input, because the table can hold any of them tomorrow.

To find a counterexample, compare Query A and Query B in the three places below. Test them on the inputs where two versions of a query most often disagree: a tie, a NULL, a repeated value, or a missing match.

1. **Row values:** What does each row represent, and do both compute the same values? Follow joins, groups, counts, and averages.
 2. **Filters:** Do both keep the same rows or groups? Compare the conditions, including NULLs and thresholds.
 3. **Output:** Do both return the same columns, values, and duplicate counts? Check `DISTINCT` and any required ordering.
-

Query-equivalent: users who span at least 2 genres

Both count distinct genres per user, then keep counts of at least two.

GROUP BY + HAVING

```
-- 3. Output: one name per qualifying user
SELECT u.name
FROM Users u
-- Attach each listen to its user and song
JOIN Listens l ON u.user_id = l.user_id
JOIN Songs s ON l.song_id = s.song_id
-- Form a group of listen rows for each user
GROUP BY u.user_id, u.name
-- 1. Row values: COUNT
-- distinct non-NULL genres
-- 2. Filters: keep groups
-- with that count >= 2
HAVING COUNT(DISTINCT s.genre) >= 2
```

CTE + WHERE

```
-- Listens has user_id; no Users needed yet
WITH UserGenreCounts AS (
  SELECT l.user_id,
    -- 1. Row values: count
    -- distinct non-NULL genres
    COUNT(DISTINCT s.genre) AS genre_count
  FROM Listens l
  -- Attach each listen's song and genre
  JOIN Songs s ON l.song_id = s.song_id
  -- Form a group for each user_id
  GROUP BY l.user_id)
-- 3. Output: join Users for each name
SELECT u.name
FROM Users u
JOIN UserGenreCounts ugc
  ON ugc.user_id = u.user_id
-- 2. Filters: keep groups
-- with that count >= 2
WHERE ugc.genre_count >= 2
```

■ same distinct-genre count

■ same threshold

Output on today's data

name
Mickey
Minnie
Daffy

Why they are query-equivalent

1. Row values: count distinct non-NULL genres for the same user.
2. Filters: HAVING and outer WHERE both keep counts ≥ 2 .
3. Output: one name per qualifying user; shared names can repeat.

Query-equivalent: ratings above each user's personal average

Both return name-rating pairs where the rating exceeds that user's average.

Correlated subquery

```
-- 3. Output: each name-rating pair once
SELECT DISTINCT u.name, l.rating
FROM Users u
JOIN Listens l ON u.user_id = l.user_id
-- 2. Filters: rating above
-- the user's average
WHERE l.rating > (
  -- 1. Row values: average
  -- the selected ratings
  SELECT AVG(l2.rating) FROM Listens l2
  -- Select only this user's listens
  WHERE l2.user_id = u.user_id)
```

CTE + JOIN

```
WITH UserAvs AS (
  -- 1. Row values: average
  -- ratings for each user
  SELECT user_id, AVG(rating) AS avg
  FROM Listens GROUP BY user_id)
-- 3. Output: each name-rating pair once
SELECT DISTINCT u.name, l.rating
FROM Users u JOIN Listens l
  ON u.user_id = l.user_id
-- Match each listen to its user's average
JOIN UserAvs ua ON ua.user_id = u.user_id
-- 2. Filters: rating above
-- the user's average
WHERE l.rating > ua.avg
```

■ same user average ■ same comparison

Output on today's data

name	rating
Mickey	4.5
Minnie	4.7
Minnie	4.6
Daffy	4.9

Why they are query-equivalent

1. Row values: same user ratings and average; AVG ignores NULLs.
2. Filters: both test rating > that user average.
3. Output: DISTINCT name-rating pairs; each pair appears once.

Output-equivalent today; not query-equivalent

Window query

```
-- Number each user's listens 1, 2, 3, ...
-- A tie still gets two different numbers
WITH ranked AS (
  SELECT *, ROW_NUMBER() OVER (
    -- Restart the numbering for each user
    PARTITION BY user_id
    ORDER BY rating DESC) AS rn
  FROM Listens WHERE rating IS NOT NULL )
-- Keep positions 1 to 3
SELECT * FROM ranked WHERE rn <= 3
```

Count query

```
SELECT * FROM Listens l
WHERE l.rating IS NOT NULL AND (
  -- Count this user's listens rated higher
  SELECT COUNT(*) FROM Listens l2
  WHERE l2.user_id = l.user_id
    -- A tie does not count as higher
    AND l2.rating > l.rating
  -- Keep it when fewer than 3 rate higher
) < 3
```

Mickey's songs by rating:

rating	ROW_NUMBER	songs rated higher	Window query keeps?	Count query keeps?
4.5	1	0	keep	keep
4.2	2	1	keep	keep
3.9	3	2	keep	keep
3.9	4	2	drop	keep

new: a second 3.9

Output equivalence and query equivalence

Output-equivalent

Same result on this database.

Query-equivalent

Same result on every database allowed by the schema.

One valid input with different results disproves query equivalence.

M1

Schema Design: One Fact, One Place

Schema design decides which tables hold which columns.

A Franken-table (or wide table) repeats itself

Put every column in one table, and each fact is stored on many rows.

Listens as one Franken-table

listen_id	name	email	title	artist	rating
1	Mickey	mickey@ex.com	Evermore	Taylor Swift	4.5
2	Mickey	mickey@ex.com	Willow	Taylor Swift	4.2
3	Mickey	mickey@ex.com	Yesterday	Beatles	3.9
7	Daffy	daffy@ex.com	Evermore	Taylor Swift	2.9
8	Daffy	daffy@ex.com	Willow	Taylor Swift	4.9

Mickey's email, copied on all 3 rows one artist, copied again and again

What repetition costs you

- 1. Update** Mickey changes his email. Fix every Mickey row, miss one, and now one person has two emails.
- 2. Insert** Add a brand-new song nobody has played yet. There is no listen row to hang it on.
- 3. Delete** Delete Mickey's last listen and his email disappears with it. The fact had nowhere else to live.

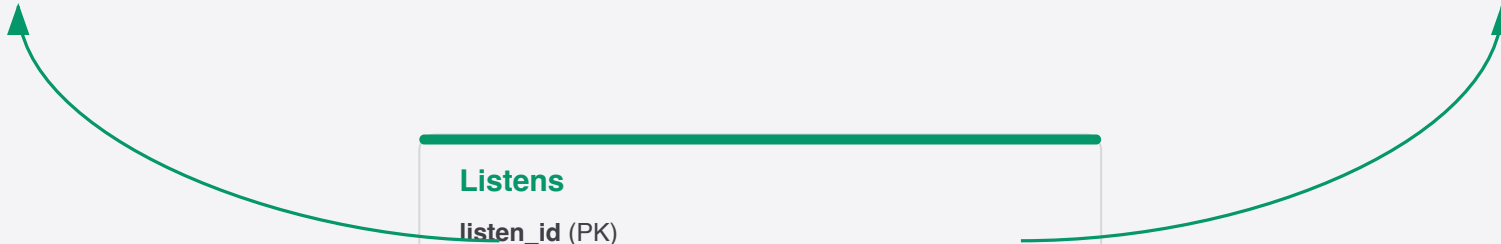
Give each fact one home

Each fact is stored once. Listens points to it by key.

Users	
user_id (PK)	· name · email
1 Mickey	mickey@ex.com
3 Daffy	daffy@ex.com
<i>Mickey's email lives here once</i>	

Songs	
song_id (PK)	· title · artist · genre
1 Evermore	Taylor Swift
2 Willow	Taylor Swift
<i>Taylor Swift lives here once</i>	

Listens	
listen_id (PK)	
user_id (FK)	· song_id (FK) · rating
<i>just the events, pointing by key</i>	

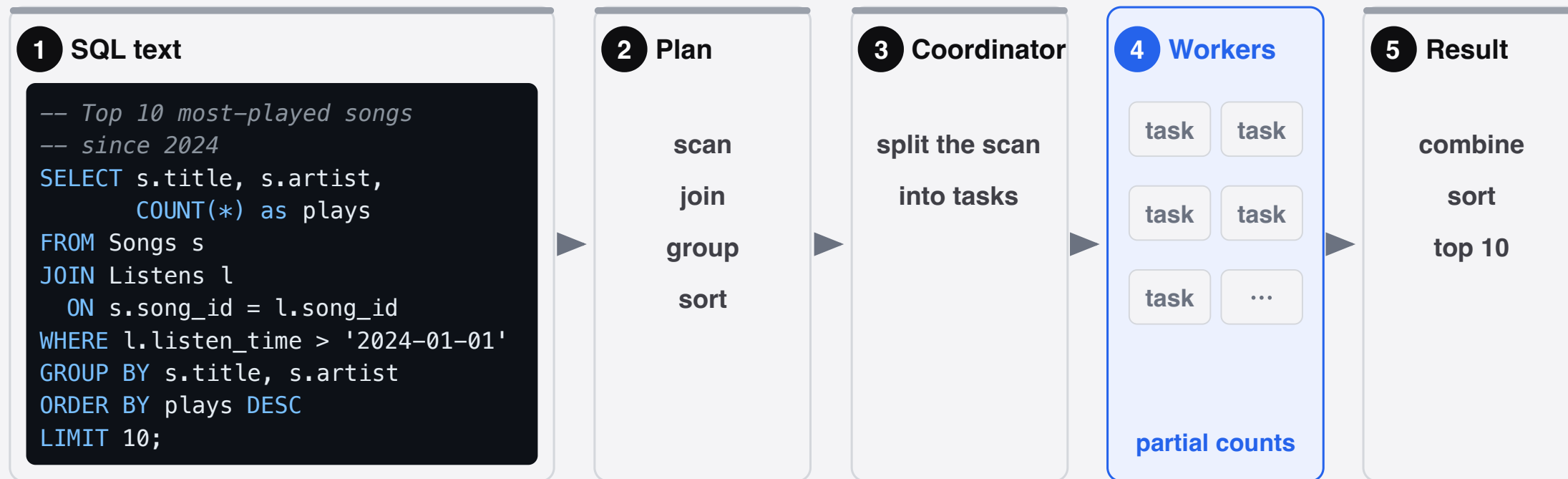


M1

How a Query Runs

A database engine turns SQL text into a physical plan of operators, such as scans, joins, aggregates, and sorts, and runs it, often split across many workers.

How one query runs: five stages



- 1 Parse and analyze.** Parse the SQL and resolve every table, column, type, and expression.
- 2 Choose a plan.** Pick algorithms and operator order from statistics and cost estimates.
- 3 Partition the work.** Divide the scan into tasks; route matching join and group keys to the same worker.
- 4 Execute in parallel.** Each worker scans its share, joins songs to listens, and computes partial counts.
- 5 Merge the result.** Combine partial counts, sort them globally, and return 10 rows.

M1

SQL Libraries: Define Once, Compose

A library packages the SQL you repeat as named, parameterized queries, so you define a term once and compose it everywhere.

A small vocabulary spans a large answerable domain

Define a few parameterized primitives. Compose them to answer the rest.

DEFINED

the vocabulary you write

`listener_of(artist)`

`active_since(days)`

`liked(min_rating)`

`power_user(min)`

each pins a choice

UTC midnight, not 24h;
a NULL rating is not a like

dimensions: artist, genre,
window, rating

ANSWERABLE

compose with $n - u$, none of it written

`All - listener_of('Taylor')`

`active_since(7) n listener_of('Beatles')`

`listener_of('Taylor') n liked(4.0)`

add `power_user`, and a new question composes:

`power_user(50) n listener_of('Taylor')`

OFF THE TABLE

no definition reaches here

which songs are good?
subjective

why did Pluto quit?
data you don't have

will Daffy churn?
a prediction, not a lookup

Curation grows the green by adding to the blue. The grey is the edge of what any definition can reach.

M1

Validating SQL: Prove the Result

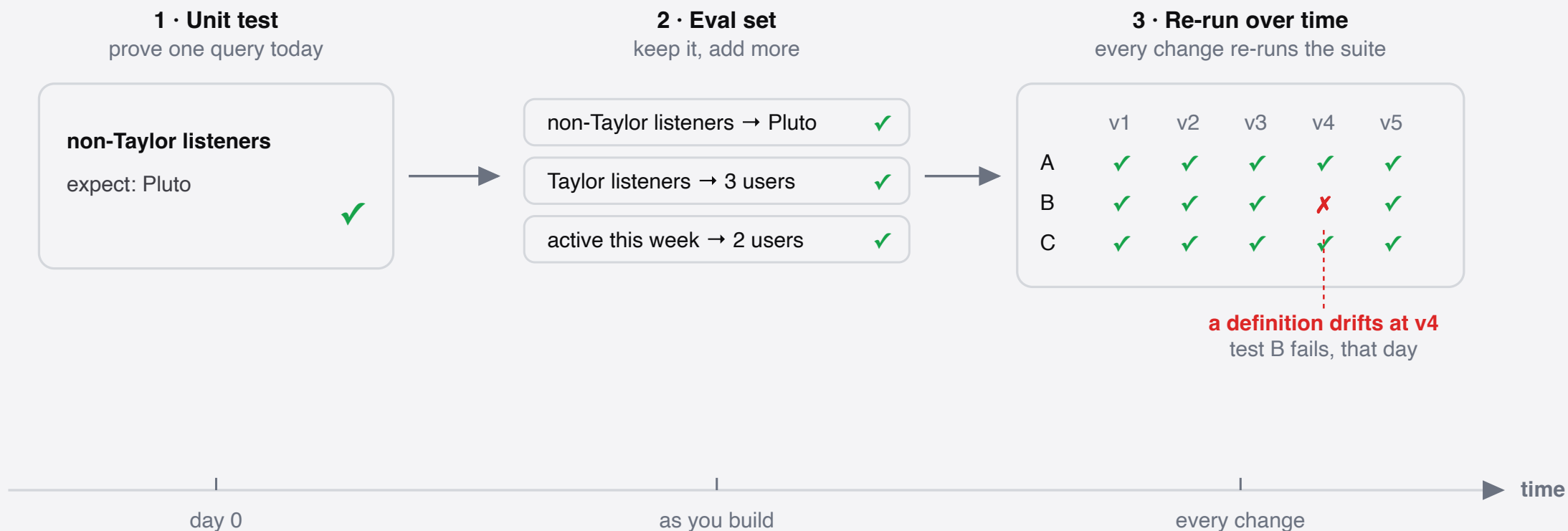
A query that runs is not necessarily right.

Unit test: listener_of

```
-- Test: who listens to Taylor Swift, on the course data?  
SELECT name FROM Users  
WHERE user_id IN (SELECT user_id FROM listener_of('Taylor Swift'))  
ORDER BY name;  
-- Expected: Daffy, Mickey, Minnie
```

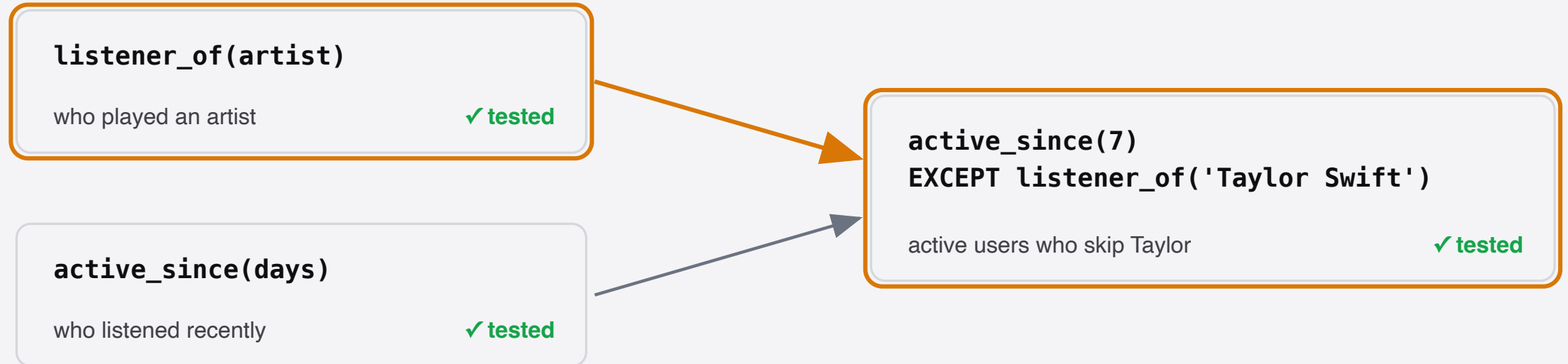
One test today, a suite that holds over time

A unit test checks one case now. Keep it, grow the set, and rerun it after each change.



Tested definitions compose

Each block keeps its tests. A change re-runs them, and every test built on top.



Change listener_of: re-run its test and the composed test above it. active_since's test stays green.

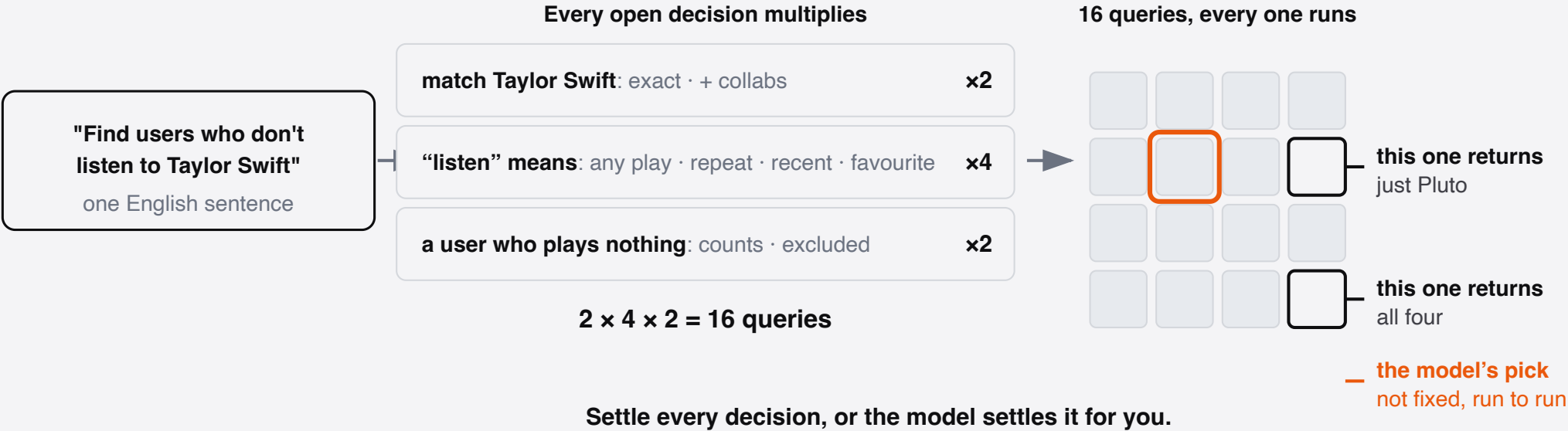
M1

Why Learn SQL When an LLM Can Draft It?

An LLM can draft SQL quickly.

One Question, Sixteen Queries

Each decision you leave open multiplies. Different queries return different answers.



BIRD · the text-to-SQL benchmark

Every serious lab competes on it, on real databases.

Who competes

OpenAI

Anthropic

Google

Databricks

Snowflake

Amazon

Alibaba

Tencent

ByteDance

Huawei

The benchmark

12,751

questions

95 databases

33 GB of real schemas

37 domains

healthcare to blockchain

Execution accuracy, best systems

~80%

BIRD
evidence supplied

10.8%

BEAVER
no hints (standard)



30.1%

BEAVER
all hints, same model

M1

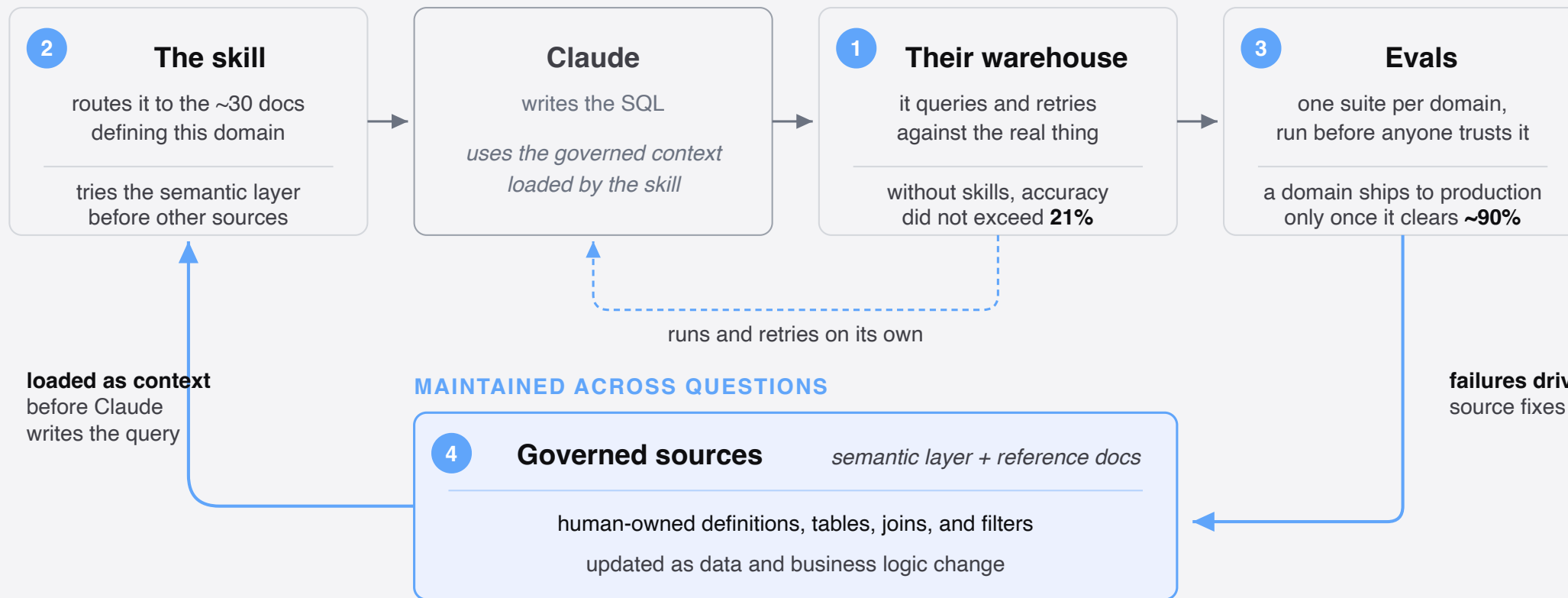
Case Study 1.1: Anthropic's Self-Service Analytics Stack

Anthropic raised Claude's accuracy on its internal analytics evals from at most 21% to above 95% by using skills that route it to governed definitions and verification.

Anthropic's analytics stack

With skills: **at most 21%** → **above 95%**, and around 99% in some domains.

ONCE PER QUESTION



Handing it the raw query history instead moved accuracy less than one point.

Two ways to make it precise

Both of them work. Only one of them you do once.

The question

users who don't listen
to Taylor Swift

three decisions open · **16 queries**

Spell it out in the prompt

answers this one question

```
match: artist = 'Taylor Swift'
listen: any row in Listens
empty: zero listens is NOT a listener
```

then write all of it again for the next question

Operationalize it

answers every question that mentions it

1 Define it once
one SQL function

2 Call it by name
the question composes

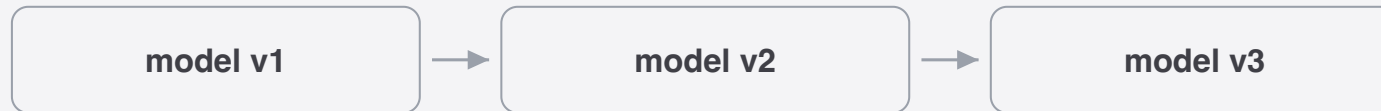
3 Set up a unit test
one per decision

Written once. Every later question about listening calls the same term.

What carries across model changes

Illustrative growth: definitions remain reusable.

MODEL VERSIONS EVALUATED



ILLUSTRATIVE LIBRARY GROWTH

3 functions
7 tests

you start here

9 functions
21 tests

each new term brings its own

22 functions
54 tests

and still growing

Reusable across versions

- the database it runs on
- the specs
- **the tests**
- the named functions

The suite is what holds the meaning still. Every swap above has to pass the same tests.

M1

Case Study 1.2: How ChatGPT Scales PostgreSQL

ChatGPT runs on PostgreSQL with one primary and nearly
50 read replicas.

One chat request, two paths through Postgres

Writes go to one primary. Reads go to nearly 50 replicas. The model runs on its own GPUs.

