

WHERE THE MONEY IS

# You've tuned the warehouse.

## *The bigger number is still hiding in the code*

Every enterprise Snowflake estate carries recoverable spend, and infrastructure work is where most teams start — it's visible, it's finite, and in most estates it's already close to tuned. That's only one part of the game. The larger pool sits below it, in the code itself: the pool that native tooling shows you least. And it isn't only a cost problem — the same inefficient code is what makes pipelines slow and unpredictable.

### 01 Where the inefficiency actually lives

A PATTERN ACROSS HUNDREDS OF ESTATES

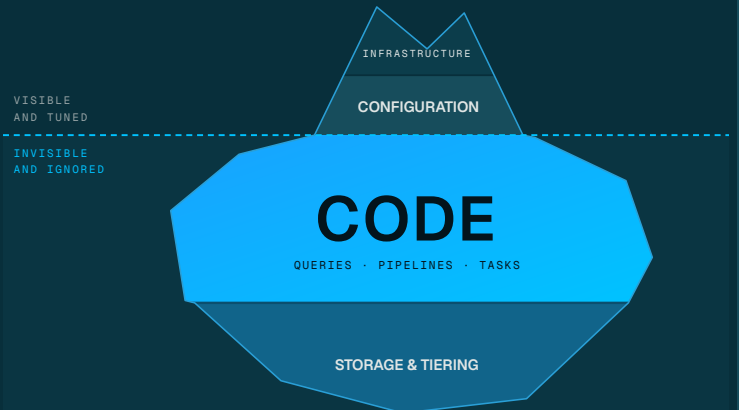
WHAT HUNDREDS OF ESTATES HAVE IN COMMON

**100s** of customer Snowflake estates Unravel has studied, across industries and estate sizes.

**40–60%** of the recoverable spend in a typical estate sits in the **code** itself — not infrastructure, not warehouse sizing.

That makes code the single largest — and least visible — category of recoverable spend in most estates, ahead of infrastructure, configuration, and storage & tiering. They are not remotely equal in size — and they are not equally visible.

Work the biggest, least-visible pool first — *not the one your dashboard happens to show you.*



The four places inefficiency hides. Proportions are illustrative, but the shape is consistent across estates: the two you can see are the two that are already close to tuned.

#### It shows up as cost

You are billed for every wasted scan, every spill to local storage, and every failed run that retried three times before it gave up.

#### It shows up as slowness

The same inefficiencies are the reason a query that should take minutes takes hours — and the reason adding compute stops helping.

#### It shows up as unreliability

Inefficient code is fragile code. It misses windows, breaks under volume growth, and fails in ways that look green on a dashboard.

## 02 What code inefficiency *actually looks like*

None of these require deep engine internals to recognize. Each one is a normal engineering decision that was reasonable when it was made and expensive at today's volume. The technical names are there for your platform team — the description is the part that matters to everyone else.

### 1 Scanning way more data than needed

OVER-SCANNING · POOR PRUNING · EXPENSIVE JOINS

COST

SPEED

RELIABILITY

Small question. Whole table gets scanned. Nothing looks broken, so nothing gets flagged.

#### IN PRACTICE

Two 20M-row tables joined for two months of data. Every run still scans all 20 million rows.

### 2 Work the warehouse can't split evenly

SKREW · SPILL · POOR PARALLELISM

COST

SPEED

RELIABILITY

One partition holds up the whole query. The rest of the warehouse sits idle. Doesn't fit in memory? It spills to local storage.

#### IN PRACTICE

Query Profile shows bytes spilled to local or remote storage right on the operator — persistent spill under normal load is the tell.

### 3 Running more than anyone needs

REFRESH CHURN · RUNAWAY QUERIES

COST

SPEED

RELIABILITY

Tasks and views refresh on a schedule nobody downstream asked for. No one compares the two, so no one notices.

#### IN PRACTICE

Runs 100×/day. Read twice. 98 runs pay for nothing.

### 4 Fails, retries, fails again

FAILED & RETRIED WORK · SILENT SKIPS

COST

SPEED

RELIABILITY

You pay for the compute either way. A task can even go green while skipping everything downstream.

#### IN PRACTICE

A failed task doesn't retry itself unless you've told it to — and a broken step downstream can still leave the graph looking successful.

### 5 Code nobody owns

UNATTRIBUTABLE SPEND

COST

SPEED

RELIABILITY

No name on it, so no one fixes it. Usually the biggest waste class of all.

#### IN PRACTICE

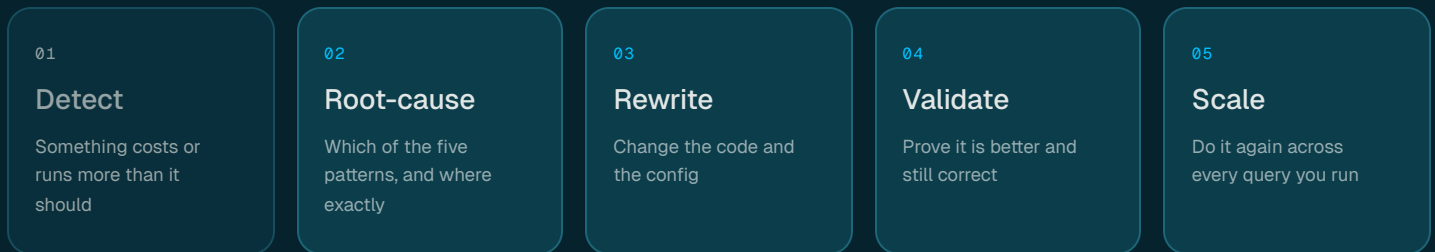
Shared-warehouse cost pools across every query that ran on it — not whoever's query actually needed it.

## WHY THE BACKLOG NEVER CLEARS

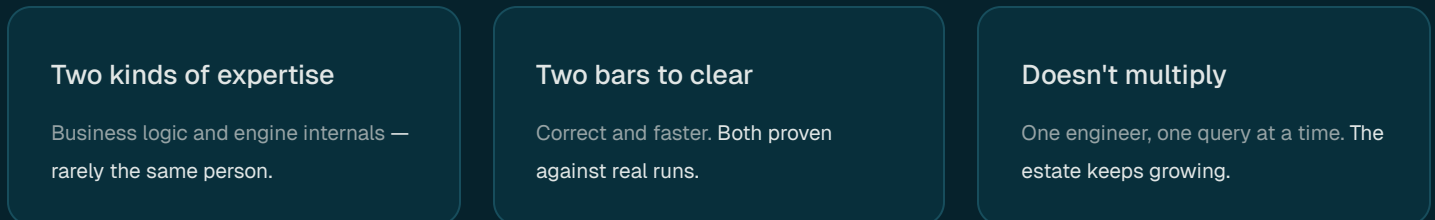
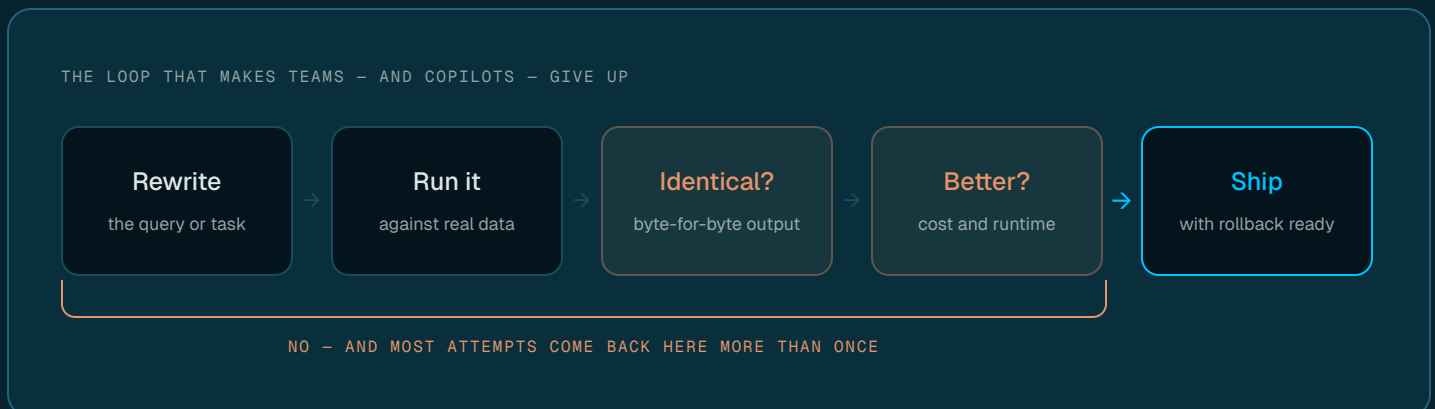
# Finding bad code is the easy part. *Fixing it is why the backlog never clears.*

Everyone can find the waste. Almost no one can fix it fast enough — not a human working alone, and not an LLM.

### 03 The five steps — and where four of them stall



WHERE THE WORK ACTUALLY IS — AND WHERE FEWER THAN 1 IN 10 FINDINGS EVER ARRIVE



## 04 Where native tools — and LLMs — *stop*

Snowflake ships genuinely good detection — that's not a knock on the platform. What happens after detection is where both native tooling and a general-purpose LLM run out of road.

### WHERE IT STOPS

#### Native tools

Give you the metrics. Never the why — or how to fix it.

#### A general-purpose LLM

Can suggest something to try. You still have to run it, check it — and often try again.

#### One suggestion at a time

Doesn't cover millions of queries a day. Or tens of thousands.

### ARVIX BY UNRAVEL

Root-causes every finding automatically, *from what actually ran*.

Writes the fix and validates it against real runs — *before it ships*.

Applies the fix across every query in the estate, *automatically*.

## Your next step FOUR QUESTIONS FOR YOUR PLATFORM TEAM THIS WEEK

- 1 Can we name our ten most expensive queries last week, by team, in under five minutes?
- 2 For those ten queries, can anyone tell you *why* each one costs what it does — not just what it costs?
- 3 How many optimization findings are open right now — and how many actually shipped last quarter?
- 4 If we rewrote one of them today, how would we prove it's faster and still correct before it ships?

### NEXT STEP · NO INSTRUMENTATION REQUIRED

Find your number *before*  
*you decide what it's worth.*

A free Snowflake HealthCheck returns your actual code-caused spend, the five patterns ranked by dollar impact, and what the first 100 rewrites would return — in about two weeks, on your own estate.

Request a HealthCheck