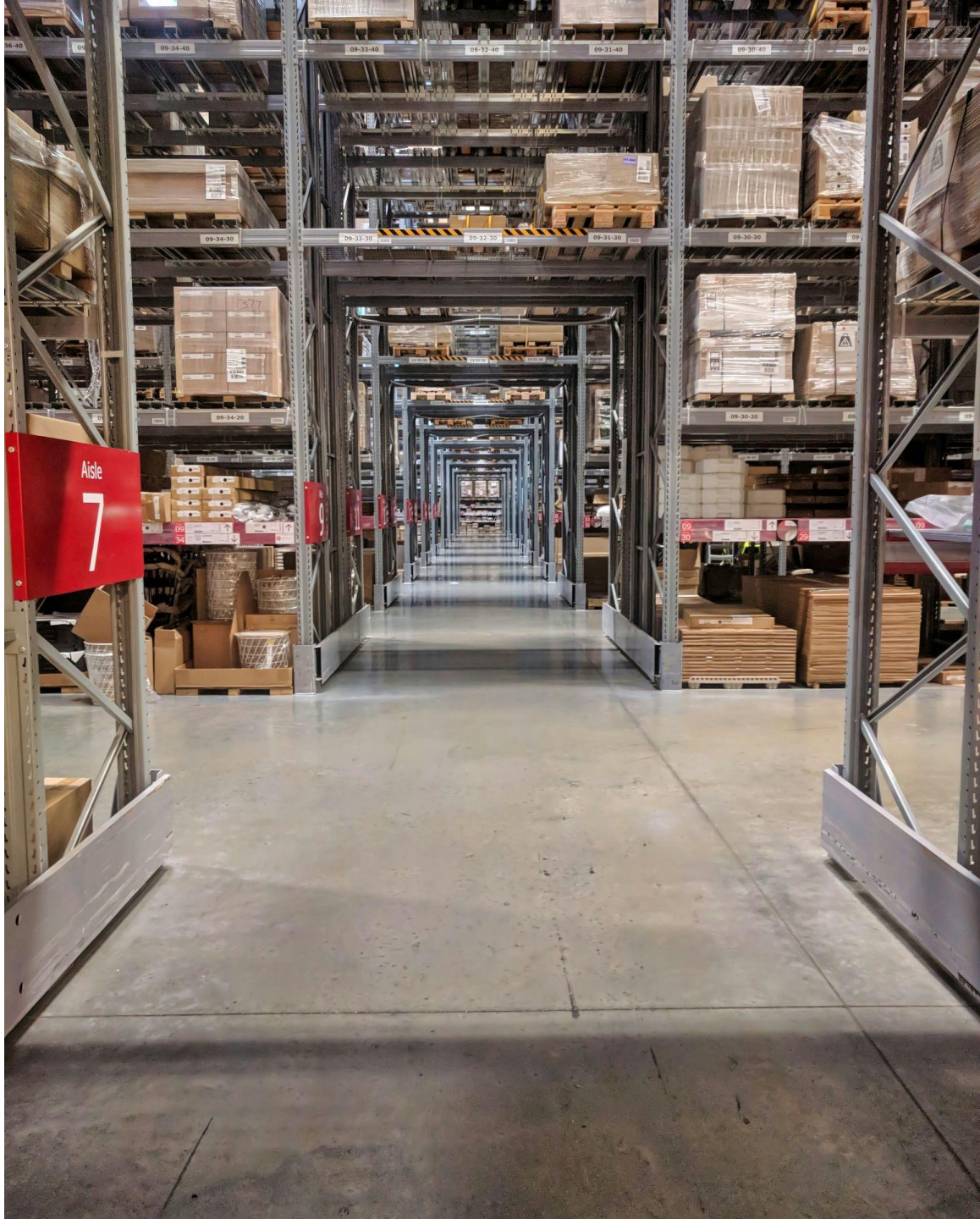
Batch Intelligence at Scale

Cost-Efficient Multi-Agent LLM Workflows with Built-In Resilience



Hi 👋

I’m Aditya Mulik

SENIOR SWE @ WALMART GLOBAL TECH

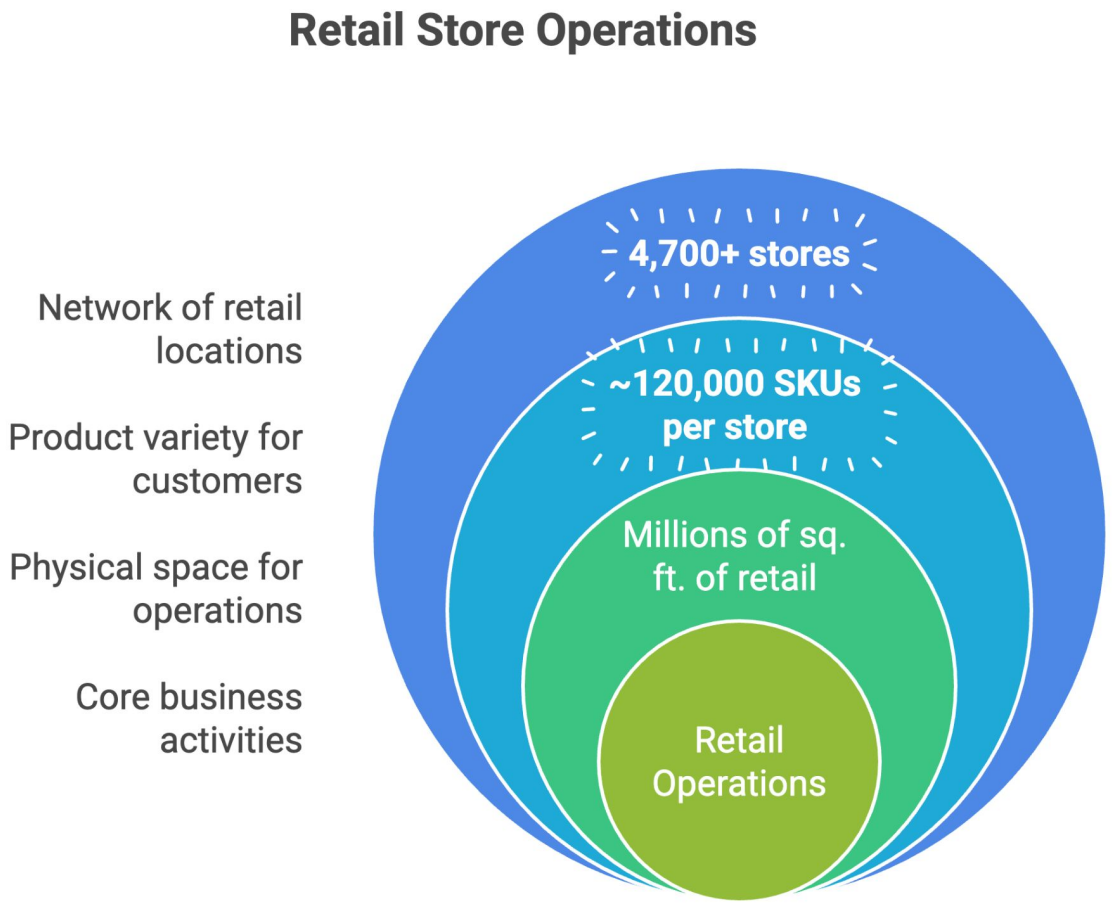
Disclaimer!!!

My presentation, comments and opinions are provided in my personal capacity and not as a representative of my employer. They do not reflect the views of my employer and are not endorsed by my employer.

“The problem I want to walk you through today is one every retailer is wrestling with …

Inventory Accuracy”

Retail at Massive Scale



> Can we predict what inventory a department needs for the next N days accurately, automatically and at scale?

The deceptively simple question driving the architecture.

The Accuracy Imperative

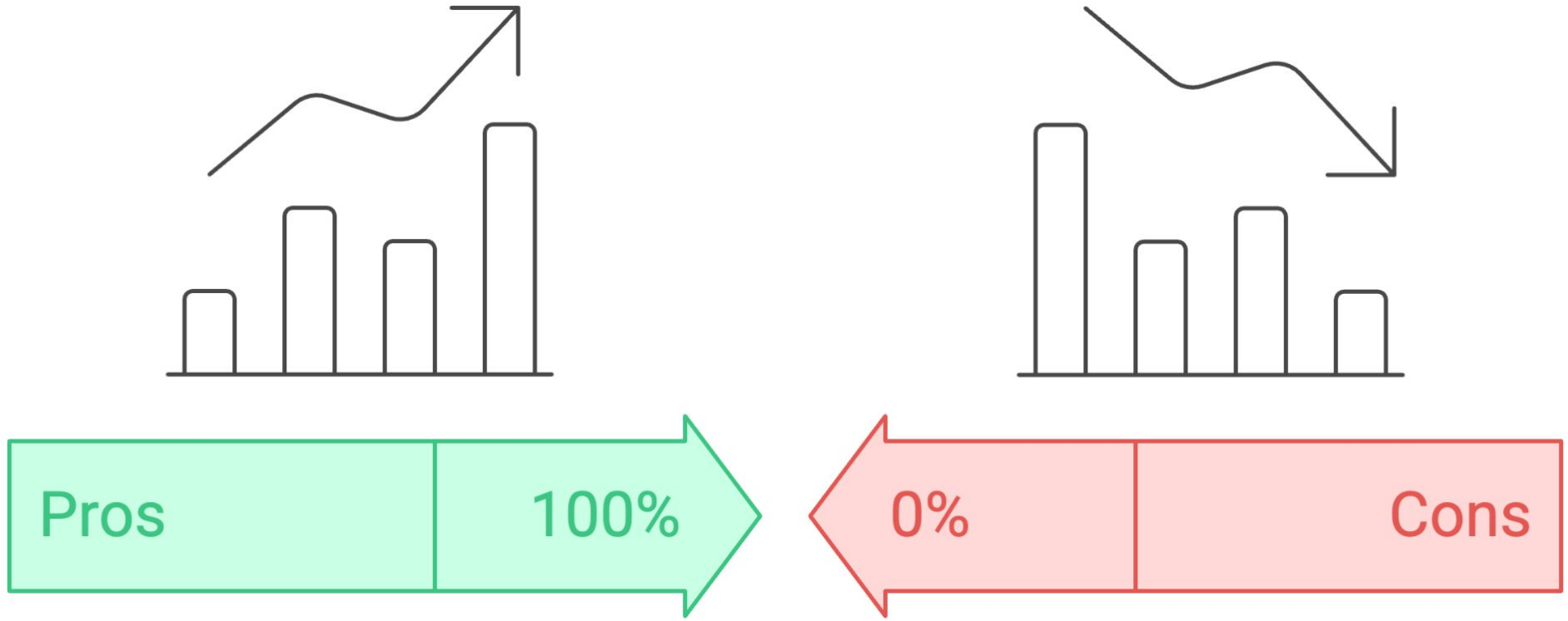
- Accuracy isn't a nice-to-have - it's **mission-critical**
- The line between a stocked shelf and an empty rack
- Stockouts and overstock both hit the bottom line
- Compounded across millions of sq. ft. of retail

“So how do we keep inventory accurate at this scale? Our first answer was the obvious one, forecast demand from history.”

Backward Looking Machine Learning Model

- Trained on historical sales data to forecast future demand
- Reliable for steady, recurring and seasonal trends
- Fully automated and scales across SKUs and stores

Sales Forecasting Accuracy



Learns from historical sales patterns

Blind to unprecedented events

> So the question became: how do we add real-time, real-world context on top of the signals we already trust?

Agents, Not Monoliths: The Sequential Workflow

Agent 1

Gathers context from various data sources for a particular department

Agent 2

Synthesizes past sales data prepared by our data engineering pipeline.

Agent 3

Analyzes the context and output from the previous agent, validates it with grounding mechanism

Agent 4

Produces the final natural-language recommendation directive.

We designed a multi-agent system where each agent maintains a single, well-defined responsibility, consuming the output of the previous step as context.

The Output & The Human-in-the-Loop

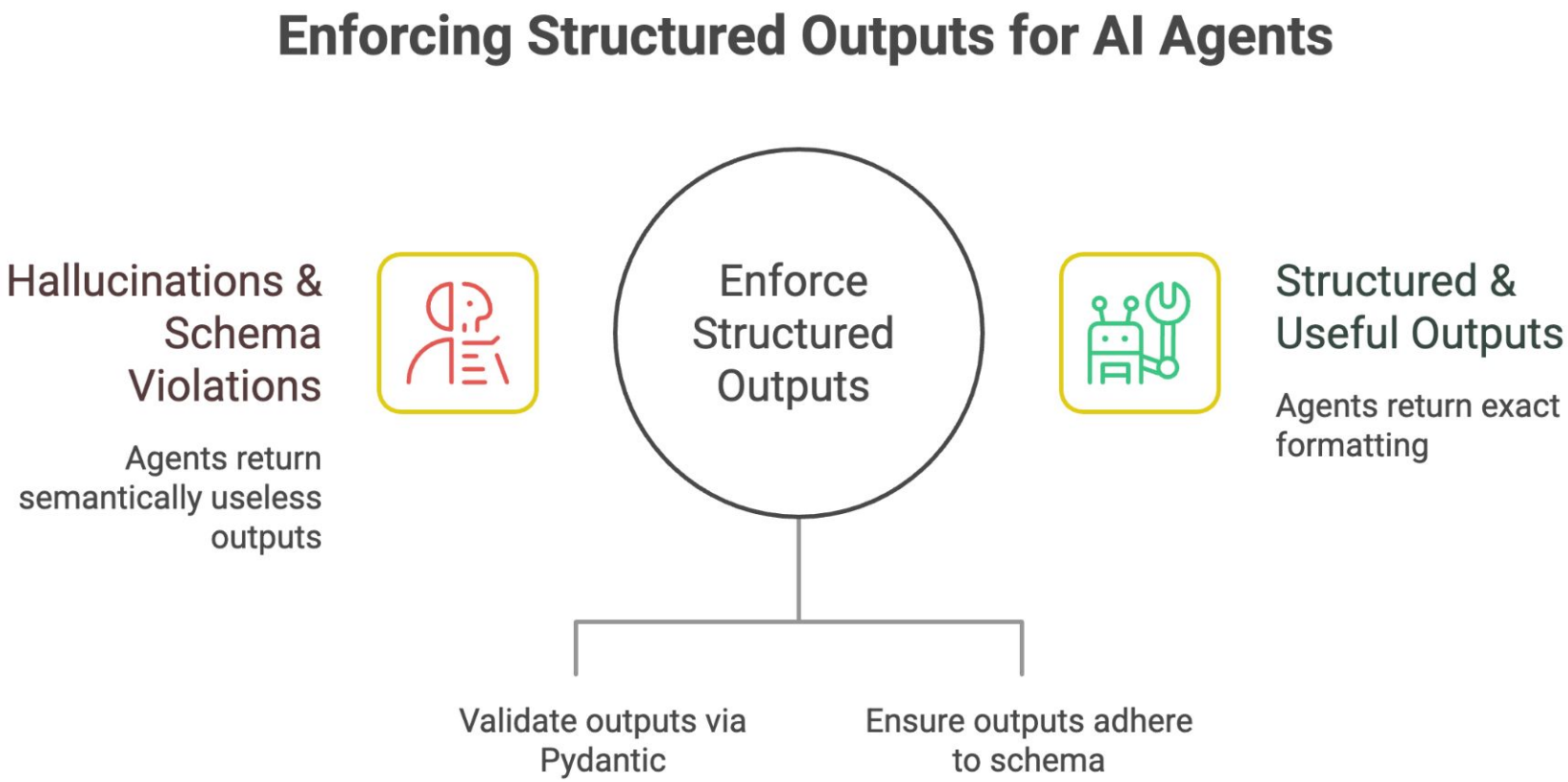
- **Plain-English Directives** — Not a dashboard, just a clear command: "Increase apple stock 15% in produce ahead of the weekend."
- **Real-Time Data Integration** — MCP lets agents pull live data sources during reasoning.
- **The Critical Checkpoint** — Store manager reviews each recommendation and can accept or override.
- **A Feature, Not a Fallback** — Human oversight is the final validation layer, catching local context data can't.

Then We Went to Production.

- The pilot was magic.
- Clean outputs, coherent recommendations & happy stakeholders.
- Then we flipped the switch to full-scale nightly runs and reality showed up.

“What works at the scale of one rarely survives the scale of thousands. Here are the **four failures** that taught us why!!”

Failure #1: Hallucinations & Schema Violations



Failure #1: Example of a bad output

```
{
  "recommendation": "I think you should probably increase
   some stock for the weekend, maybe around 15% or so for
   produce items that sell well.",
  "confidence": "pretty high"
}
```

Take this output we got, it reads fine to a human, but there's no item code, no number, no timeframe an associate could actually execute on.

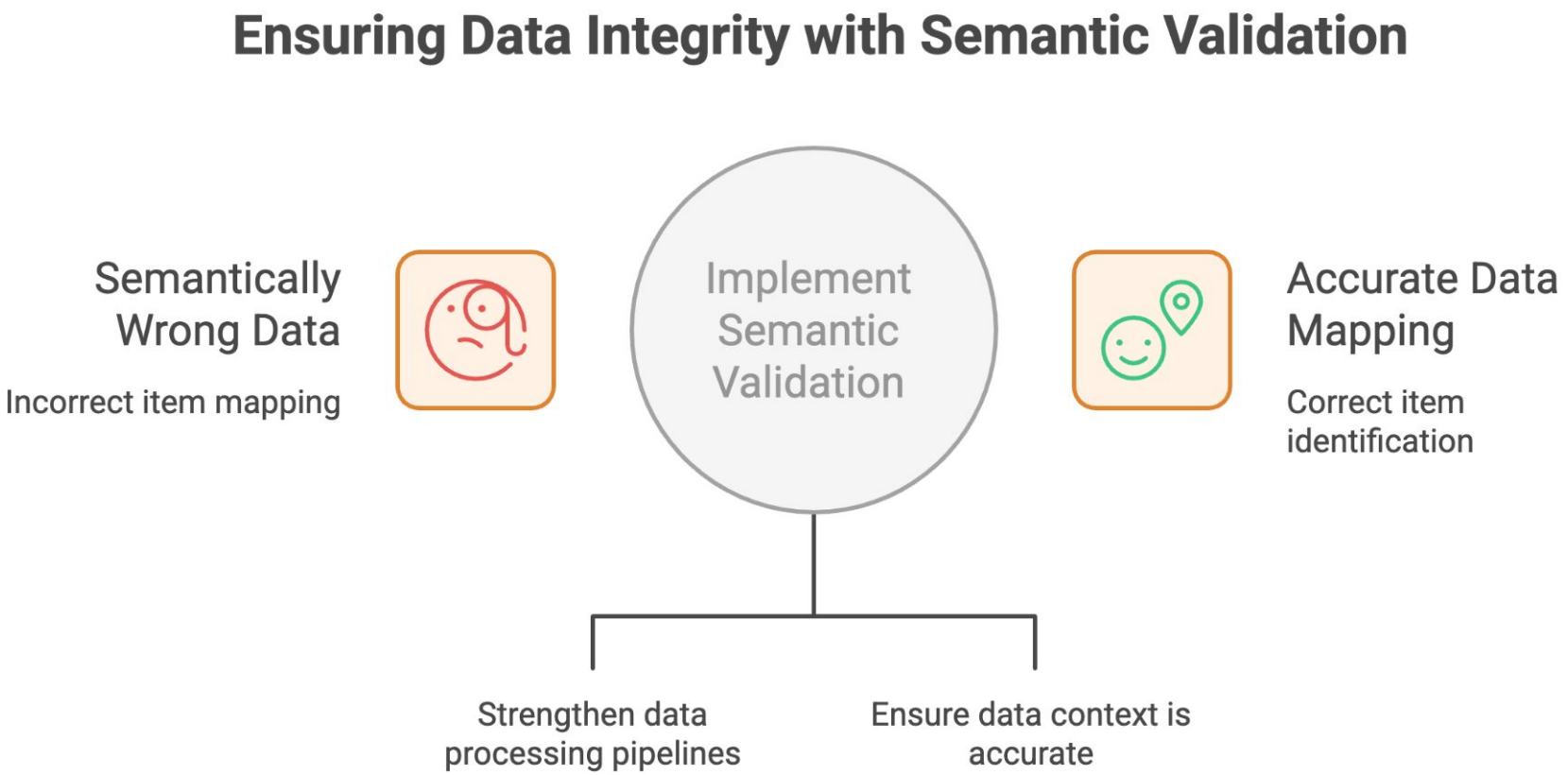
Failure #1: Fix it with structured output and schema enforcement

```python
from pydantic import BaseModel, Field
from enum import Enum

class Department(str, Enum):
    PRODUCE = "produce"
    BAKERY = "bakery"

class StockDirective(BaseModel):
    sku: str = Field(..., pattern=r"^\d{8,12}$")
    item_name: str = Field(..., min_length=1)
    department: Department
    adjustment_pct: float = Field(..., ge=-100, le=100)
    confidence: float = Field(..., ge=0.0, le=1.0)
```

```json
{
  "sku": "004011",
  "item_name": "Honeycrisp Apple",
  "department": "produce",
  "adjustment_pct": 15.0,
  "confidence": 0.87
}
```

Here's the same output, now schema-enforced, every field typed and validated: the exact SKU, item, department, and adjustment. No ambiguity left

Failure #2: Semantically Wrong



Failure #2: Example of a bad output with valid schema

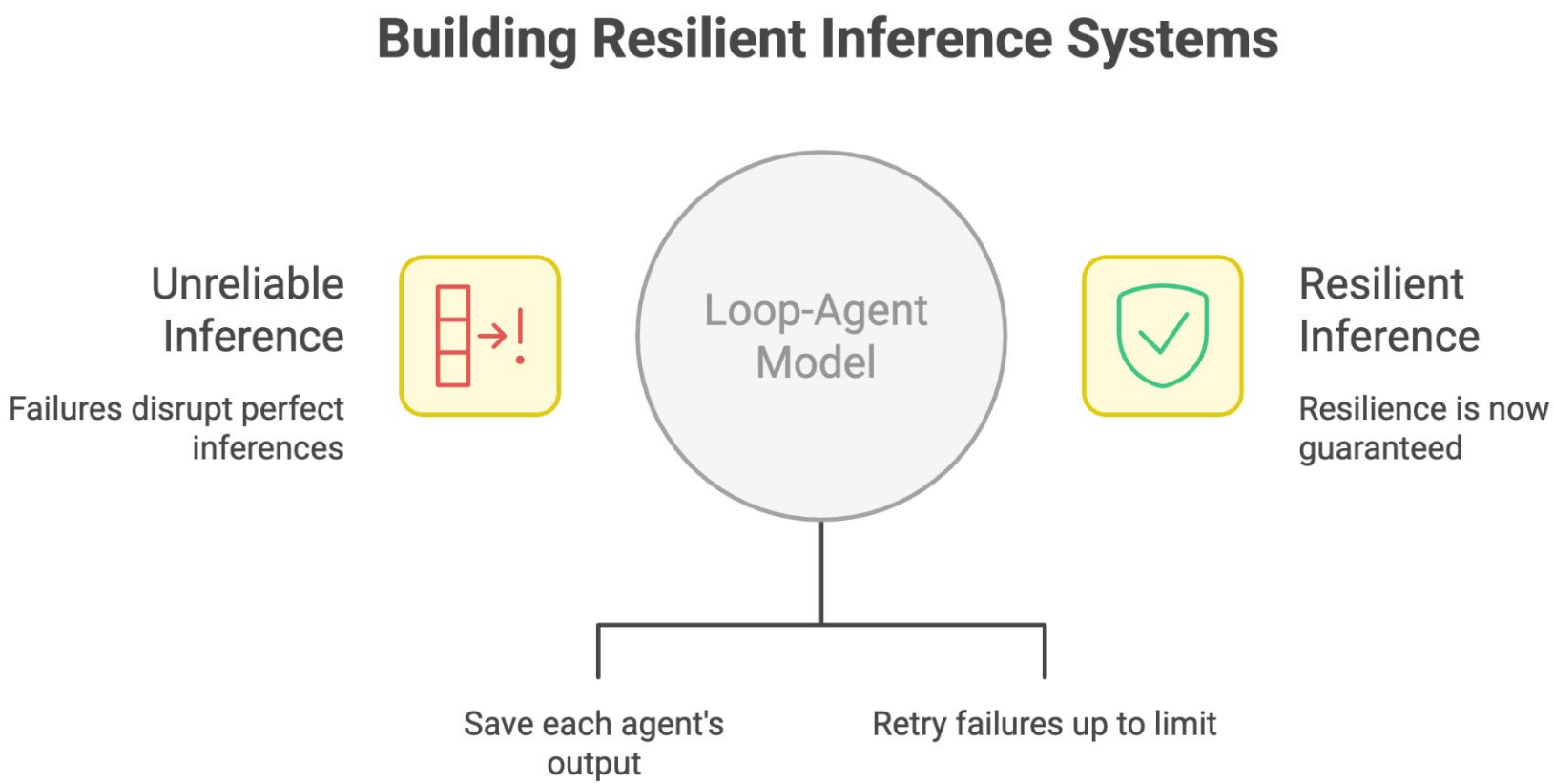
Here's the dangerous kind of failure, perfectly valid JSON, but it maps Honeycrisp Apples to auto parts because the SKUs collided. The schema can't catch this; only semantic validation can.

Failure #2: Fix the context or ground response with tools

```python
async def validate_coherence(d: StockDirective, mcp: MCPClient) -> None:
    # Query the catalog service through MCP for ground truth
    record = await mcp.call_tool(
        "catalog_lookup",
        {"sku": d.sku}
    )

    if record is None:
        raise SemanticError(f"Unknown SKU {d.sku}")

    if record["department"] != d.department:
        raise SemanticError(
            f"SKU {d.sku} ({d.item_name}) belongs to "
            f"'{record['department']}', not '{d.department}'"
        )
```

We caught semantic errors by validating each SKU's context against an authoritative source before the AI's output reached production

Failure #3: Outputs That Vanished



Failure #3: Example logs of network failures

```
[02:14:07] directive_agent → produced ✓
           SKU 003283, produce, +15%, confidence 0.91
[02:14:08] db_write → TimeoutError: connection reset
[02:14:08] run aborted – directive lost ✗
```

Logs show us that timeout error will result in a failed output or ignored result

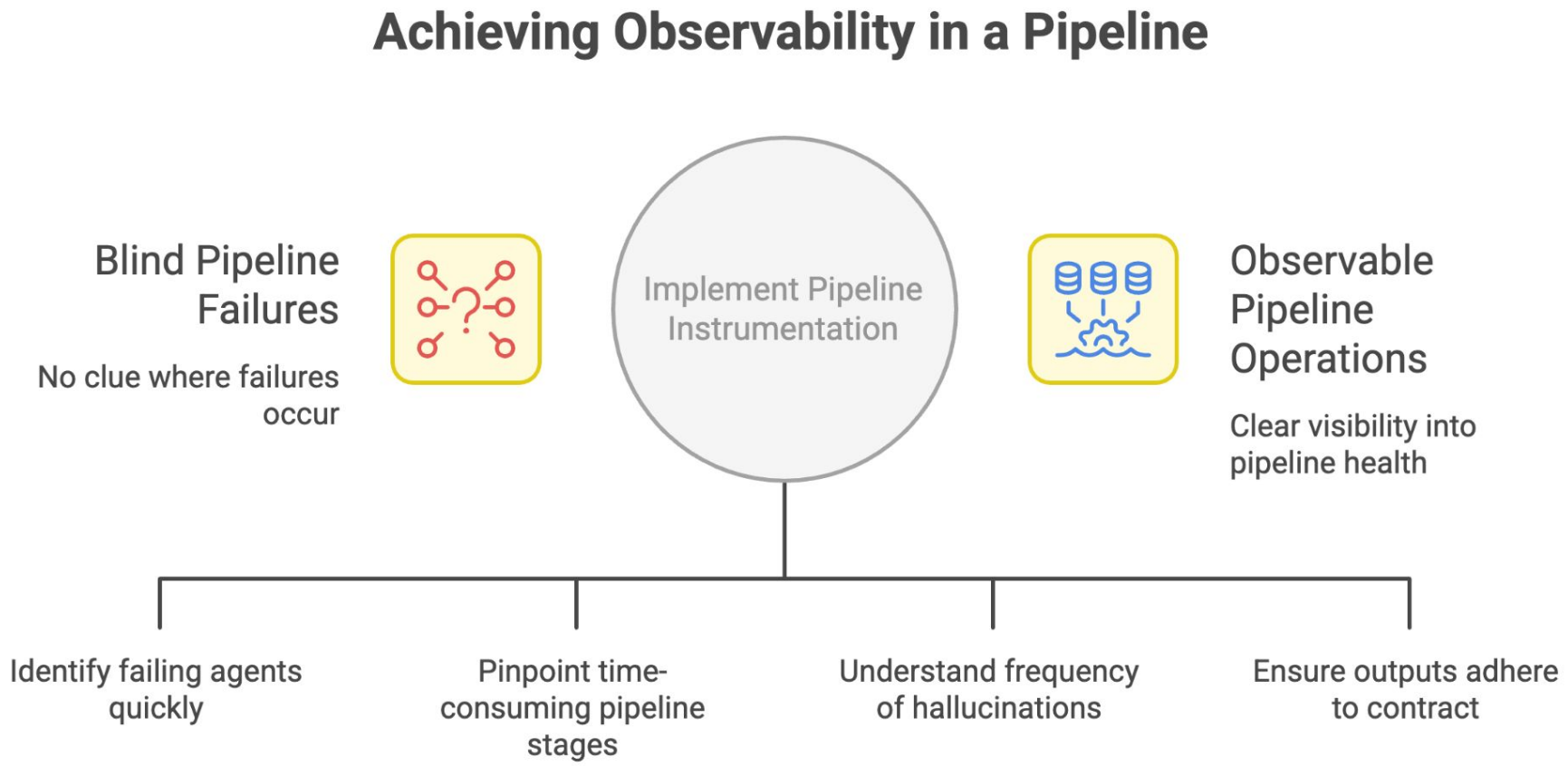
Failure #3: Fix with loop agent with set retries

```
refinement_loop = LoopAgent(
    name="InventoryRefinementLoop",
    # Order is crucial: validate/critique first, then refine or exit
    sub_agents=[
        data_gathering_agent,   # gather data from data pipelines
        synthesis_agent,  # synthesis data for specific department
        grounding_agent, # agent response validator
        structure_agent # validate final agent response
    ],
    max_iterations=5 # bounded retries — fail fast, never loop forever
)
```

```
[02:14:08] db_write failed → retry 1/3 (backoff 2s)
[02:14:10] db_write → success ✓  directive persisted
```

The loop-agent retries on failure — a single agent or the whole chain — up to a bounded limit, then escalates

Failure #4: We Were Flying Blind, Observability is a must



Failure #4: Fix with logging, tracing & dashboards

```
logger.info("agent_complete", extra={
    "agent": "coherence_critic",
    "status": "retry",
    "attempt": 2,
    "reason": "schema_violation",
    "latency_ms": 142,
})
```

Example of structured logging, enables developers to trace workflow easily!

Failure #4: Dashboard Visibility

LOG Level	LOG Message	Timestamp
INFO	LLM Response for user query "xyz"	4/15/26 9:42
INFO	File saved successfully	4/15/26 9:43
ERROR	LLM Response doesn't match the schema	4/15/26 9:43
WARNING	LLM took 4 seconds to respond	4/15/26 9:43
INFO	Successfully retrieved response from get_help mcp tool	4/15/26 9:43
ERROR	Tool response failed, retrying	4/15/26 9:43

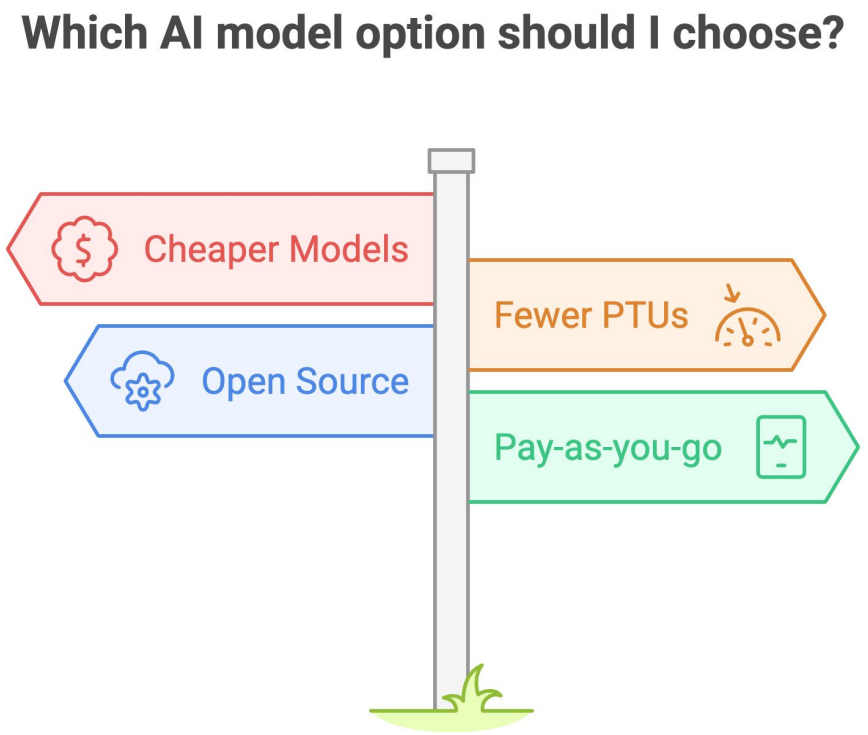
You can't debug what you can't measure - non-deterministic pipelines demand tracing and structured logging.

“With failures behind us, the next question on the table was **cost**!
How much is this whole solution going to run us?”

The PTU problem!

- We needed ~400 PTUs to run at our scale
- But PTUs reserve capacity for the entire day
- Our workload was nightly batch jobs - not 24/7
- We were paying for round-the-clock capacity we used for a few hours

The options on the table!

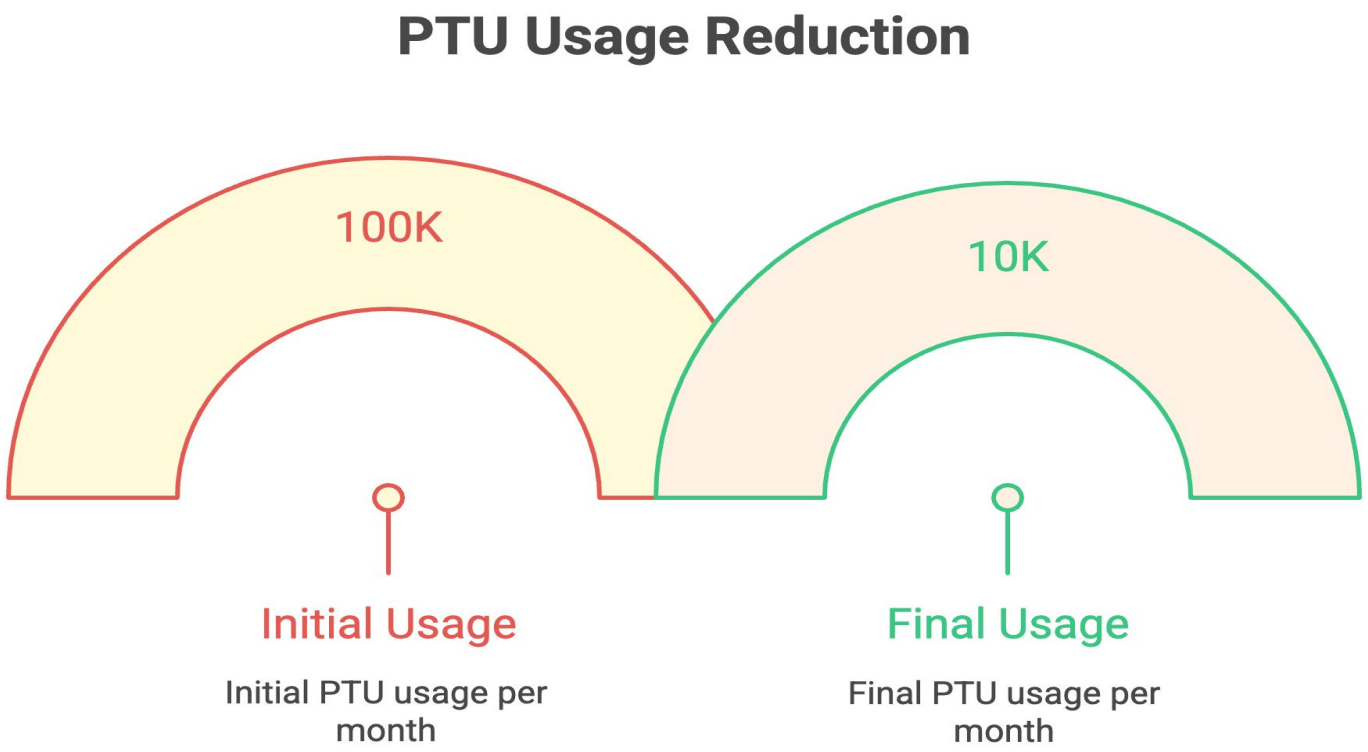


We were debating **Quality v/s Cost**, choosing either was not the best option!

The Decision: Pay-As-You-Go

- Still a frontier, well-trained foundational model quality held
- Trade-off: latency and rate limits
- Our existing retry layer absorbed both resilience we already built
- No reserved capacity sitting idle at 2 AM

The Payoff

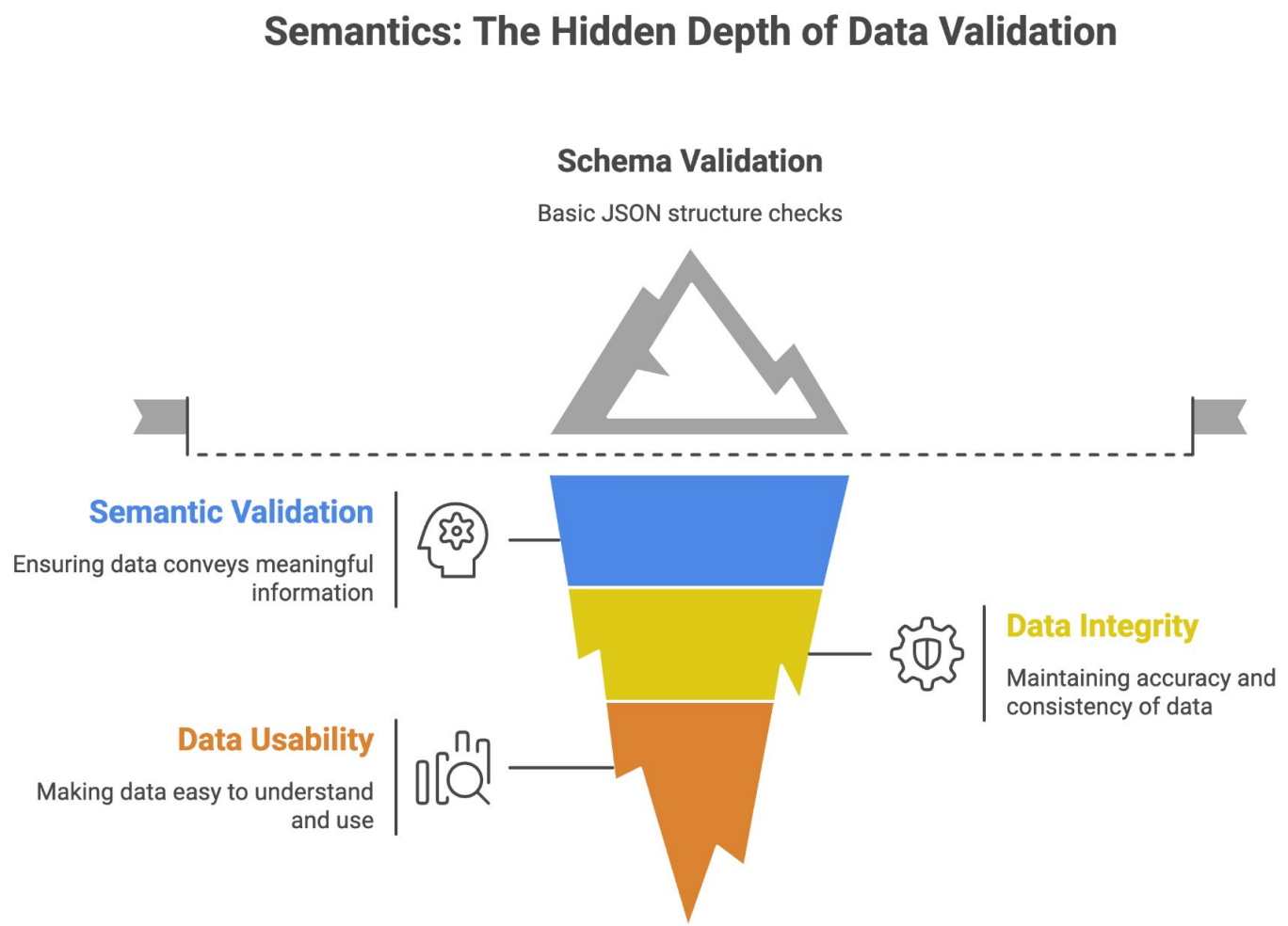


Same tokens, same model, just a pricing-tier switch from PTU to pay-as-you-go, for a drastic drop in cost

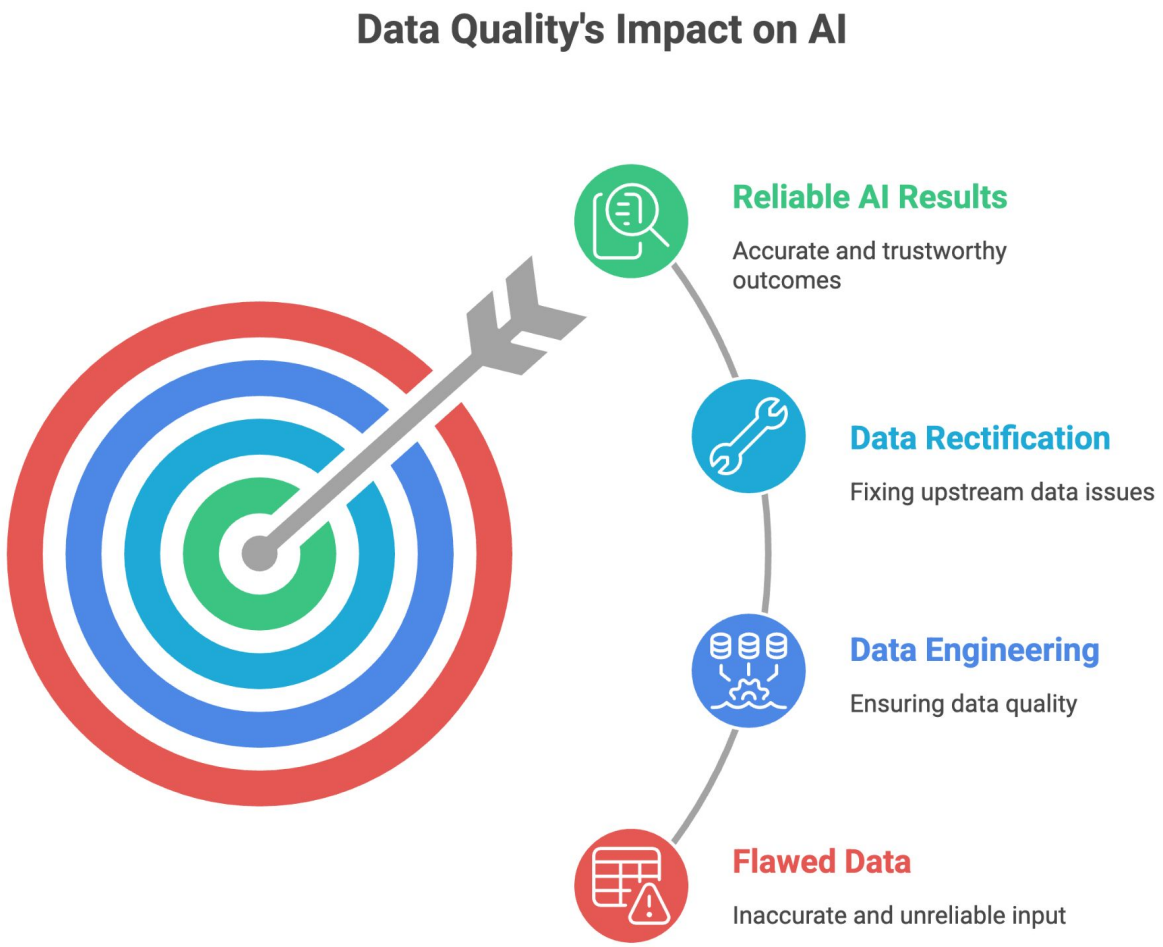
“Every failure and every fix left us with a lesson worth carrying forward

Here's what we **learned building at scale** ”

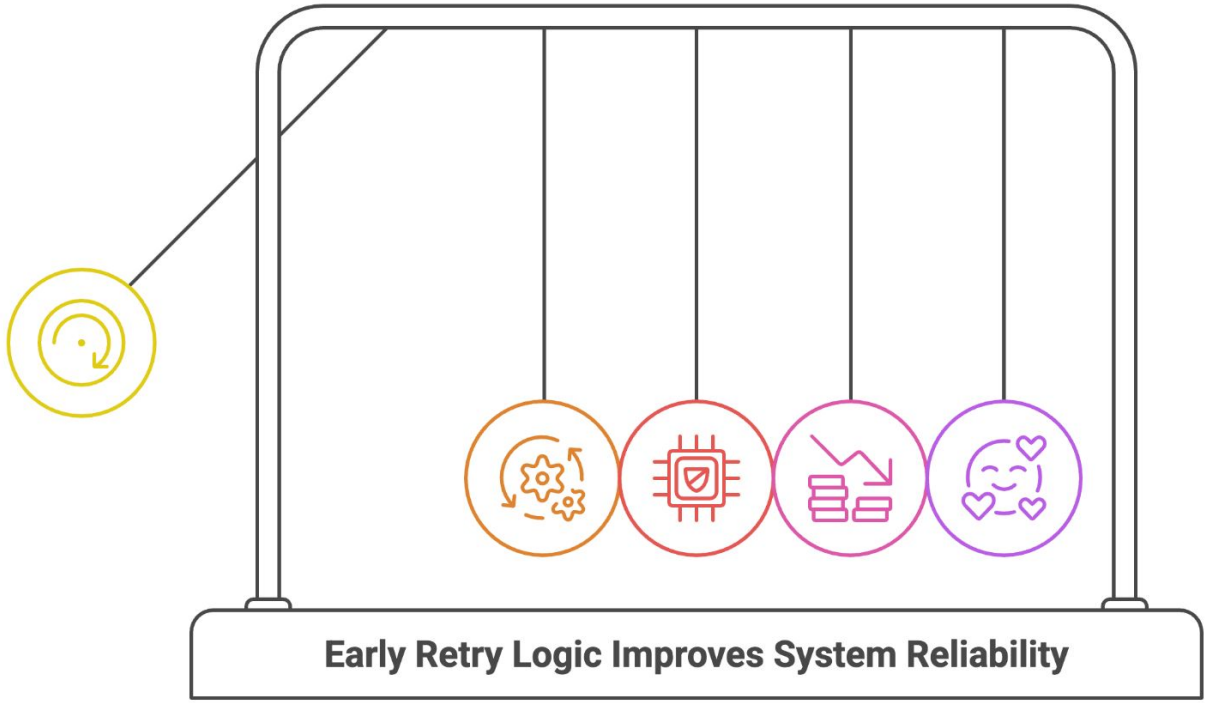
Learnings - Schema is necessary, semantics is the work



Learnings - Data engineering limits AI quality



Learnings - Build retry logic early



Retry Logic
Implemented from the outset

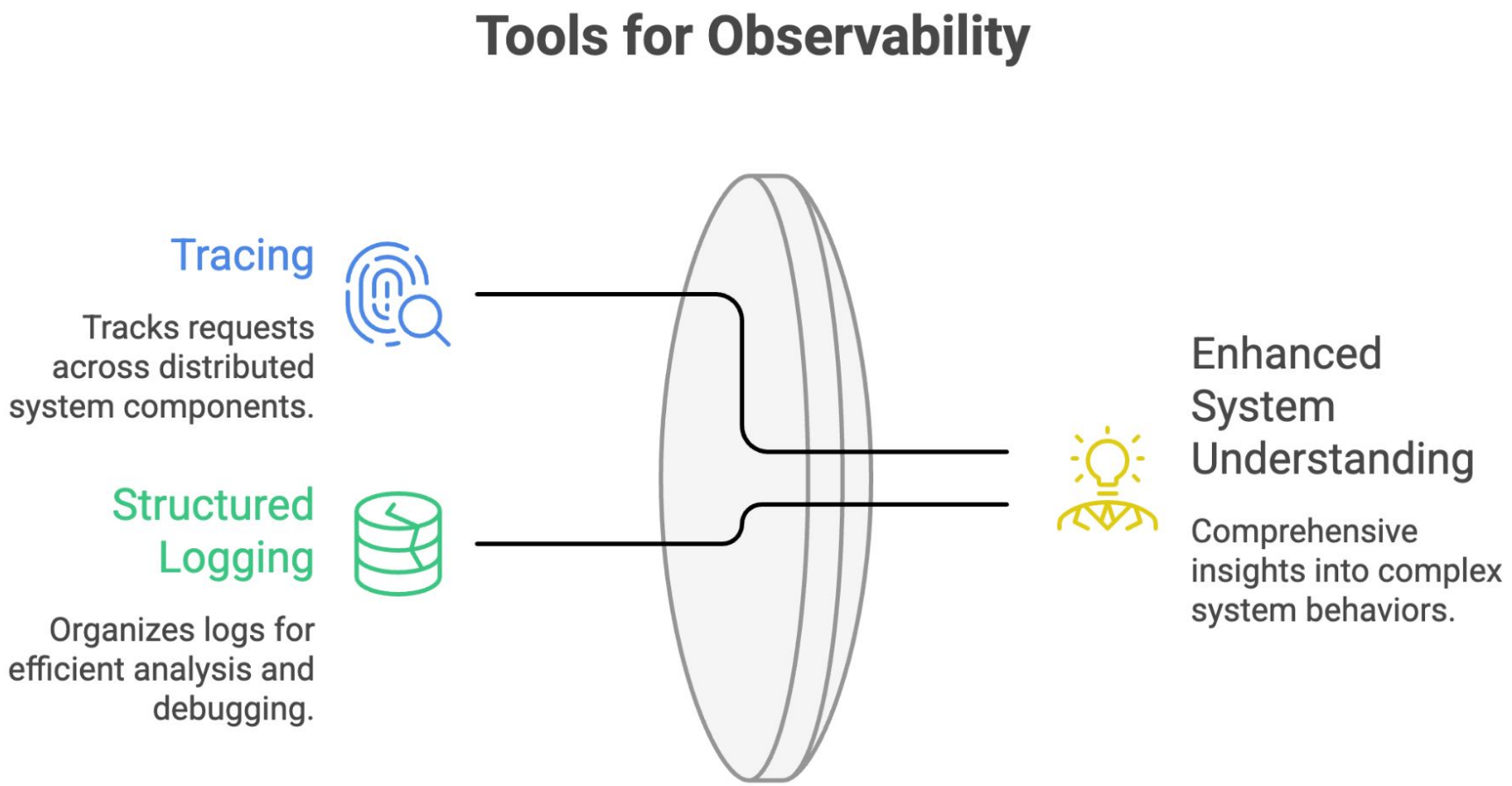
Smoother Operations
Reduced unexpected failures

Enhanced Reliability
Proactive system enhancement

Optimized Resources
Cost savings and efficiency

Improved Experience
Better user satisfaction

Learnings - Observability is mandatory

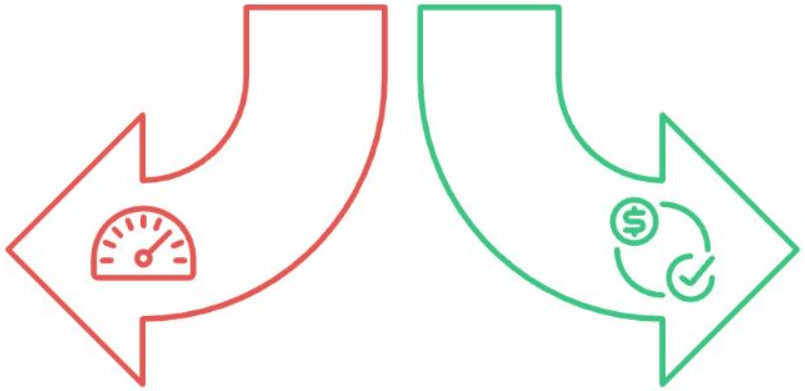


Learnings - Match infrastructure to requirements

Which infrastructure should be used for a nightly batch job?

Real-time Infrastructure

Provides high performance and low latency, but is unnecessary for non-latency-sensitive jobs and can be costly.



Cost-effective Infrastructure

Offers sufficient performance for batch jobs without unnecessary costs.

"Tomorrow at 6:00 AM, a store manager will confidently adjust their inventory. They don't know about our schema validation, our loop-retries or our nightly API optimizations. They don't have to. That is the ultimate metric of ***resilient engineering****."*

Summarizing What We Learned Building at Scale

- **Schema is table stakes, semantics is the real work:** Focus on validation checks that go beyond basic JSON structure.
- **Data engineering limits AI quality:** Garbage in results in confidently-stated garbage out. Fix the upstream data first.
- **Build retry logic early:** It is not an afterthought. It will save you from network drops, hallucinations and high cloud bills.
- **Observability is mandatory:** Tracing and structured logging are the only ways to understand non-deterministic pipelines.
- **Match infrastructure to requirements:** A nightly, non-latency-sensitive batch job does not need real-time PTU guarantees.

Scan the QR Code to know more about me!

Thank You!