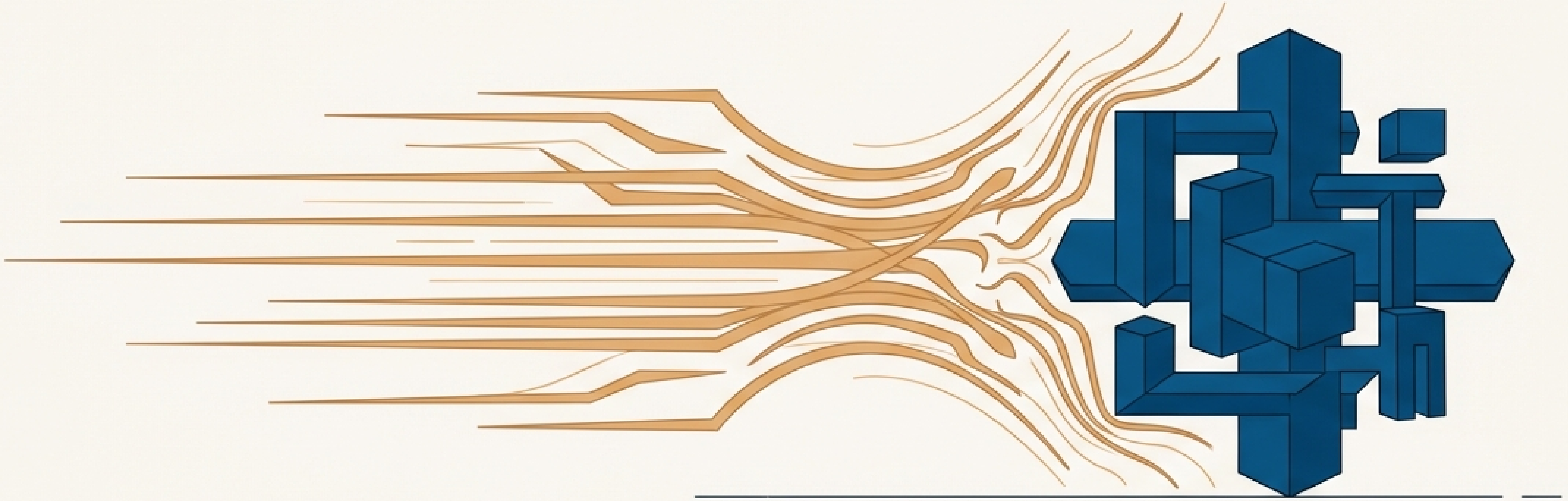
The Core Tension in Big Data: Speed vs. Truth



The Dilemma

Every data platform must reconcile two competing demands:

Speed (Velocity)

The business needs insights now. Real-time dashboards, immediate fraud detection, and instant recommendations demand millisecond latency. This is the world of streaming.

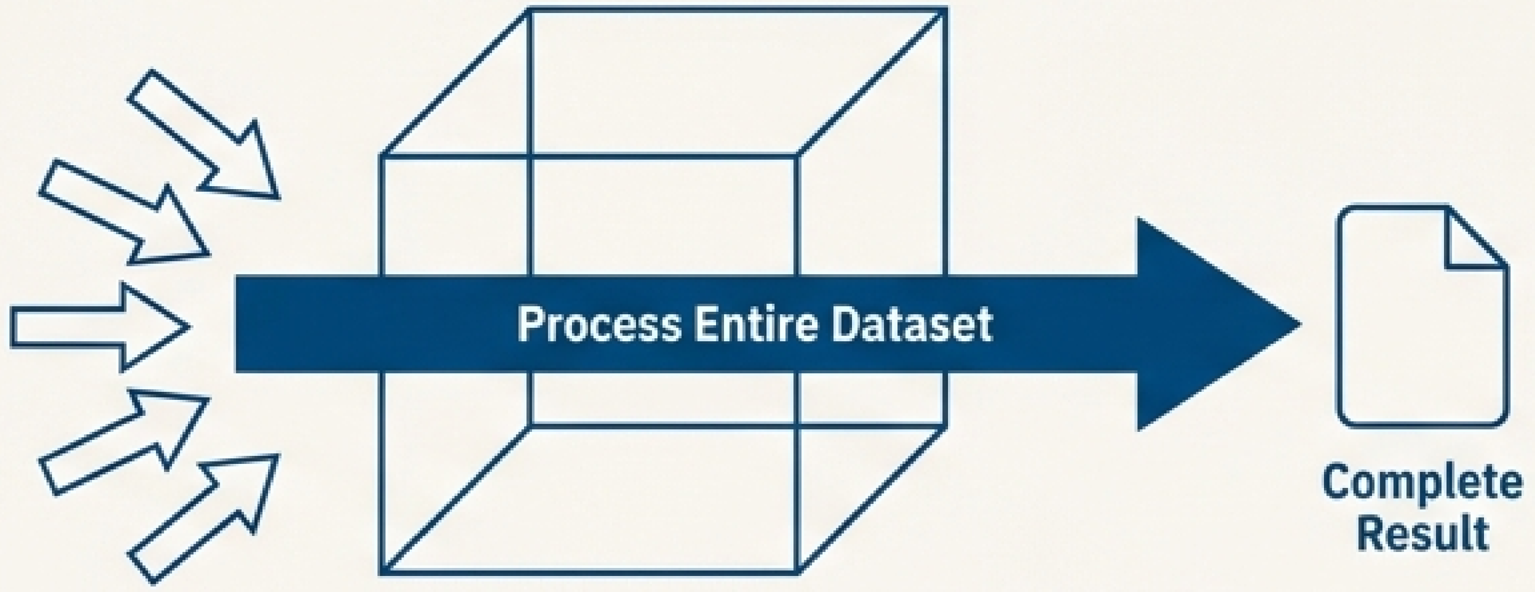
Truth (Veracity)

The business needs insights that are correct. Comprehensive historical reports, accurate financial models, and regulatory compliance demand completeness and absolute accuracy. This is the world of batch processing.

The Question: How do we build systems that deliver both without compromise? This challenge gave rise to the Lambda and Kappa architectures.

The Two Foundational Paradigms: Batch and Streaming

Batch Processing (The Complete Picture)

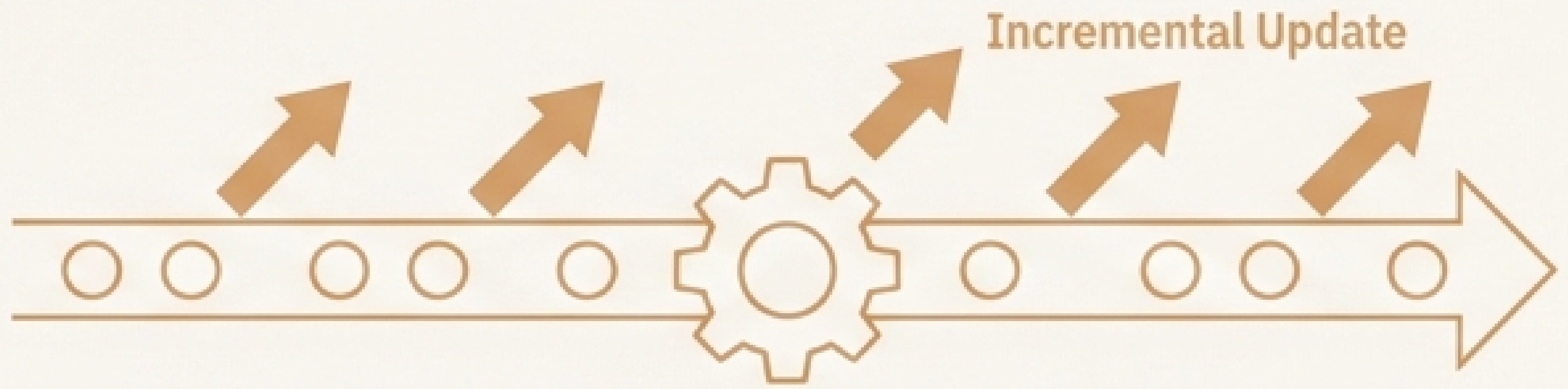


Concept
Processes a fixed, known volume of data. Queries run against the entire dataset to compute a result.

Strengths
- **Complex Logic**: "On SQL, you can do practically anything." Ideal for heavy joins, complex aggregations, and sophisticated models.
- **Error Correction:** Easy to reprocess. If you find a bug, you simply erase the result and recalculate from the full dataset.

Weaknesses
- **Latency**: High-latency by nature. Data must be collected, loaded, and then processed, which can take hours.

Stream Processing (The Immediate Answer)



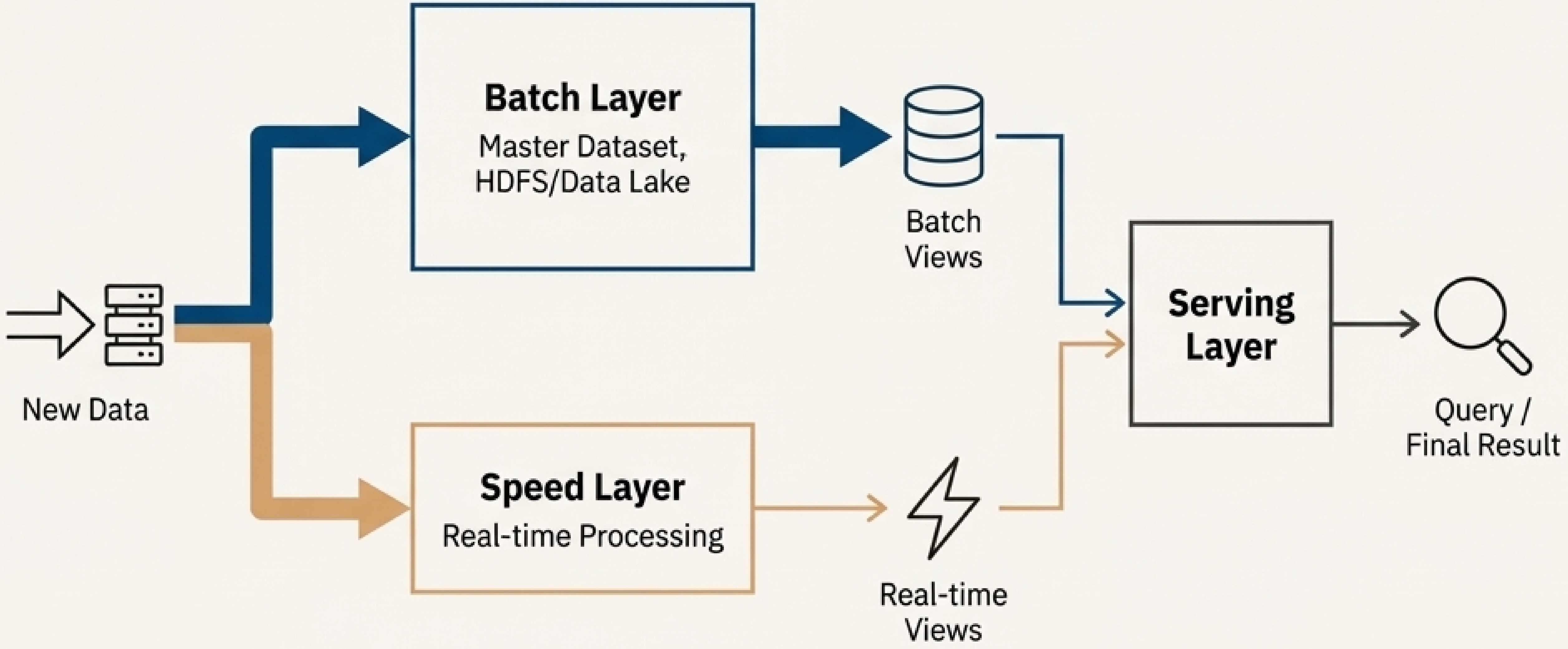
Concept
Processes a continuous, unbounded flow of events in real-time. State is updated incrementally.

Strengths
- **Speed**: Extremely low latency (milliseconds). Results are available almost instantly.
- **"Boundless Data"**: Can handle infinite data streams as it doesn't need to store the full history for processing. Events are processed and can be discarded.

Weaknesses
- **Simple Logic**: Struggled with complex, non-additive measures (e.g., `COUNT DISTINCT`) and logic that requires historical context.
- **Hard to Correct**: "Events have already gone to the trash." Fixing a past mistake is incredibly difficult as the original data is gone.

The First Grand Compromise: The Lambda Architecture

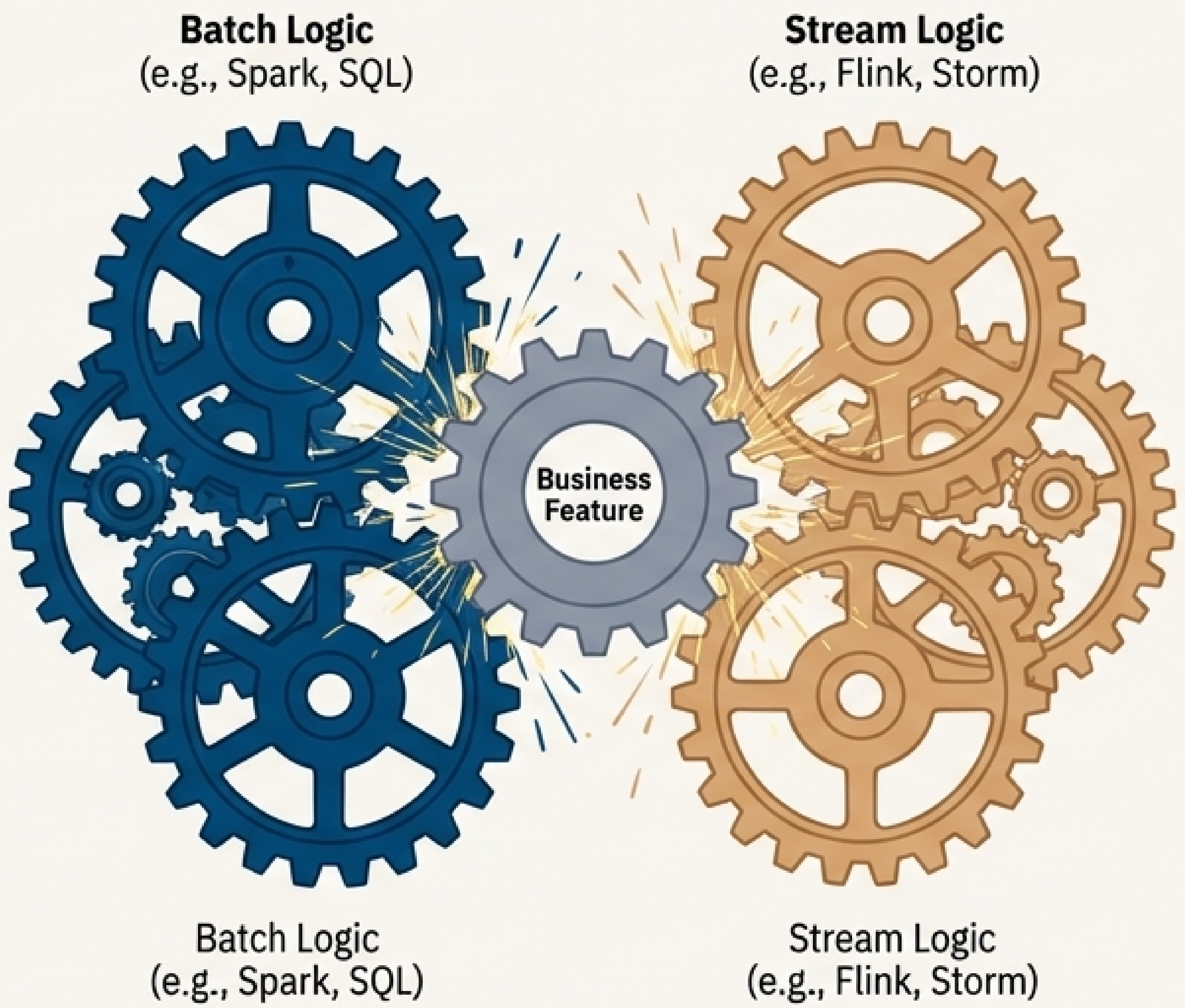
"The advantages of one system are the disadvantages of the other. The obvious solution is to combine them." — Nikolay Golov, Avito



Core Logic

1. **Ingest:** All data is sent to both the batch and speed layers.
2. **Batch Layer:** Stores the complete, immutable master dataset. Periodically recomputes comprehensive and accurate views from the entire history. This is the source of "truth".
3. **Speed Layer:** Processes data in real-time to provide low-latency, approximate updates for the most recent data. This is the source of "speed".
4. **Serving Layer:** Responds to queries by merging results from the batch views and the real-time views, providing a complete picture.

The Complication: The Heavy 'Duplication Tax' of Lambda



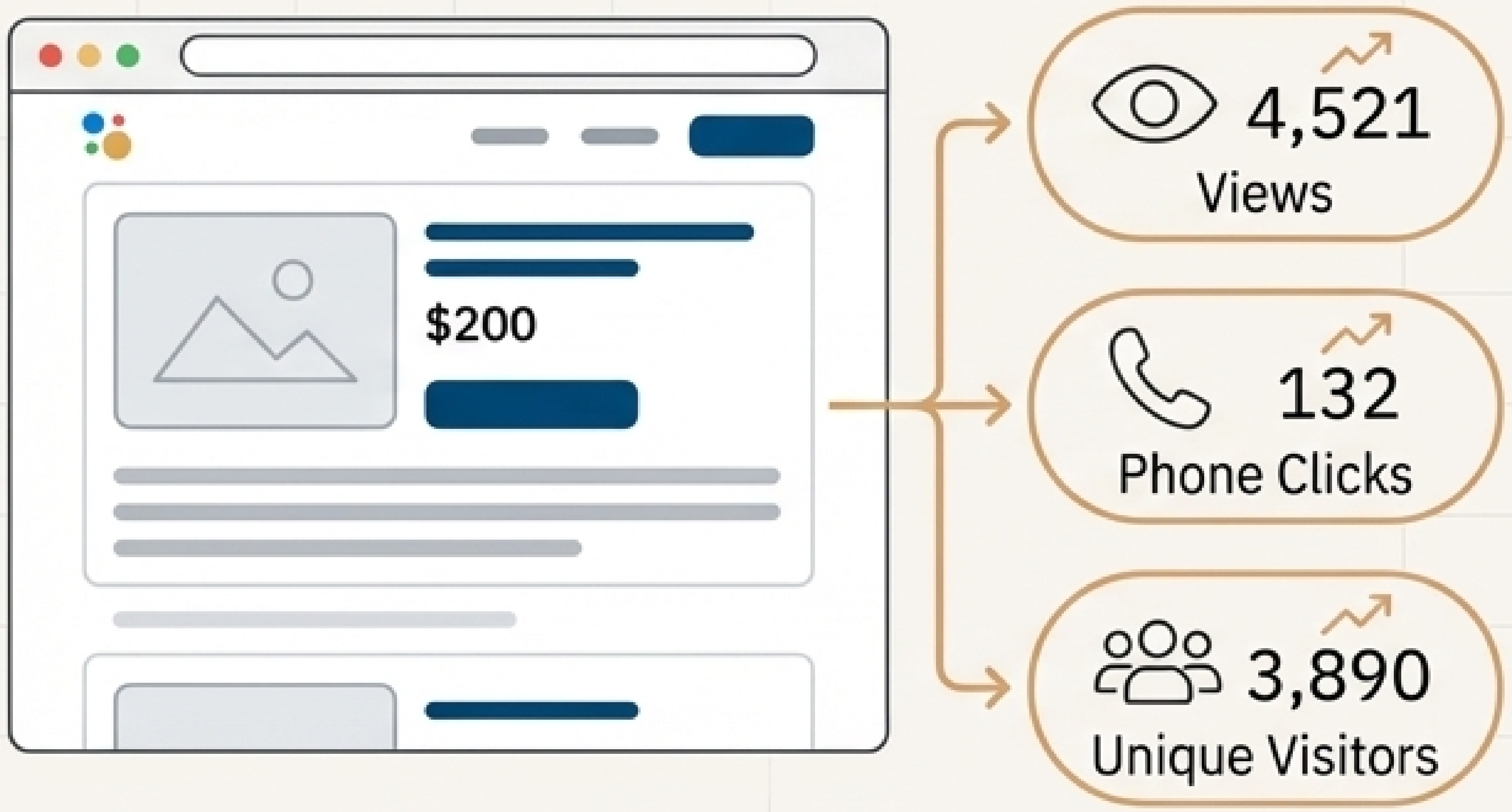
Martin Kleppmann noted a key practical problem: "Having to maintain the same logic to run both in a batch and in a stream processing framework is significant additional effort... the operational complexity of debugging, tuning, and maintaining two different systems remains."

Dual Codebases: Business logic must be implemented twice: once for the batch system and again for the stream system. This is error-prone and slows down development. "The two codebases drift apart, and the project dies."

Complex Merging: The serving layer's task of merging batch and real-time views is non-trivial, especially for complex outputs beyond simple time-series aggregations (e.g., joins, sessionization).

Expensive Reprocessing: While reprocessing the entire historical dataset is powerful, it's expensive on large datasets. This often leads to incremental batch processing, which adds complexity and blurs the line with the streaming layer.

In the Wild: Avito's Real-Time Counter Challenge



The Business Problem

Build a service to provide real-time counters for user ads (views, phone number reveals, unique visitors, etc.).

The Engineering Requirements

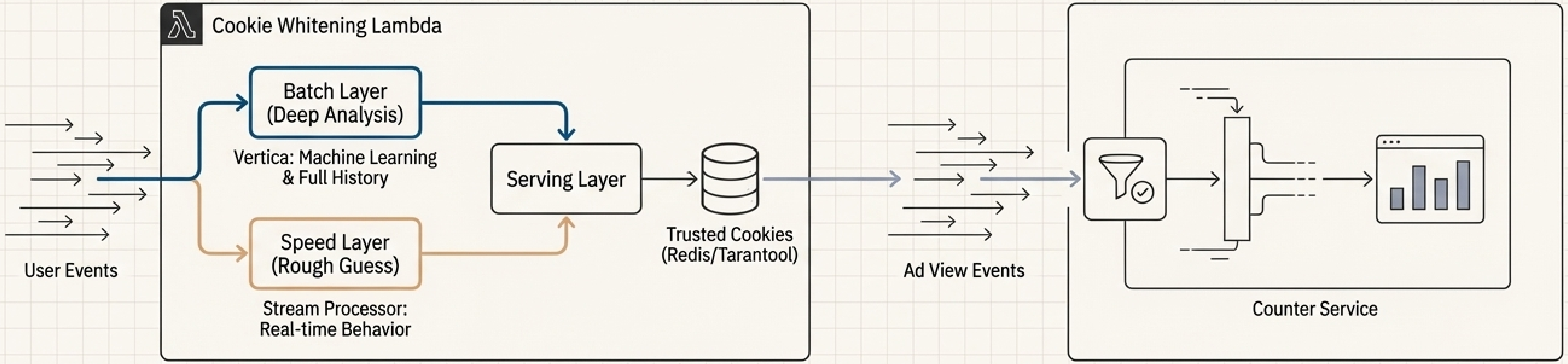
- **Speed**: Users expect to see view counts update instantly after a call or a page refresh. Latency must be in seconds or milliseconds.
- **Accuracy & Stability**: Numbers can't suddenly drop after a 'cleaning' process. This erodes user trust.
- **Complex Filtering**: Must accurately filter out bots and parsers, a task that requires historical context.
- **Evolving Logic**: Must handle new counters and logic changes without losing historical data.

Initial Approach (Streaming-only): A simple streaming counter was built first. It quickly ran into three critical failures:

1. **'Short History' Problem'**: New counters started at zero, creating data inconsistencies.
2. **Poor Filtering'**: Simple stream-based filtering was ineffective against sophisticated bots.
3. **Unfixable Errors'**: Bugs in the counting logic permanently corrupted the data, as past events were already discarded.

Avito's Pragmatic Solution: A 'Lambda for Filtering'

Instead of building one giant, monolithic Lambda, they broke the problem down. **The most complex part—bot detection—**was offloaded to its own dedicated Lambda microservice.



Cookie Whitening Lambda
Batch Layer (Deep Analysis)
Vertica: Machine Learning & Full History
Speed Layer (Rough Guess)
Stream Processor: Real-time Behavior
Serving Layer
Trusted Cookies (Redis/Tarantool)
User Events
Ad View Events
Counter Service

This mini-Lambda isolates the complexity of bot detection.

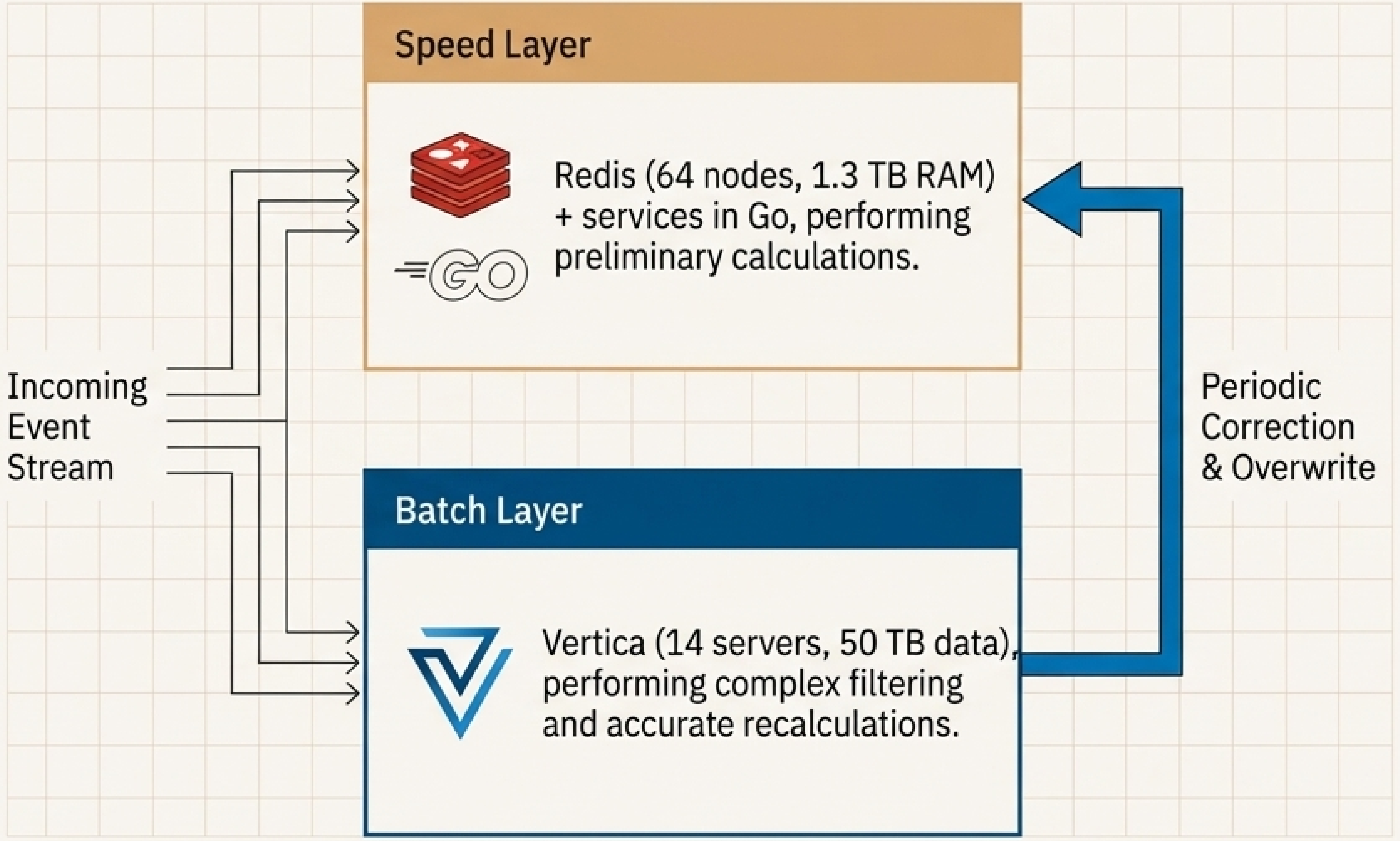
The main counter remains simple, performing a fast key-value lookup to filter events.

How it Works

- **Speed Layer (Cookie Whitening):** A stream processor makes a 'rough guess' about which cookies are legitimate based on real-time behavior.
- **Batch Layer (Cookie Whitening):** A batch process in Vertica runs a deep analysis using machine learning and full user history to create a definitive list of good/bad cookies.
- **Serving Layer (Result):** The definitive list from the batch layer constantly updates the in-memory database of trusted cookies, correcting the rough guesses from the speed layer.

Benefit: This isolates the complexity. The main counter service remains simple; it just needs to do a fast key-value lookup. The logic duplication is minimized and contained within a single, focused service.

The Avito Counter Architecture in Production


Speed Layer
Redis (64 nodes, 1.3 TB RAM) + services in Go, performing preliminary calculations.
Incoming Event Stream
Periodic Correction & Overwrite
Batch Layer
Vertica (14 servers, 50 TB data), performing complex filtering and accurate recalculations.

Embrace Microservices:

“If you realize simplicity is failing... don't grow a monolith. Build another Lambda contour.”

Simple Logic is Key:

The speed layer logic must be kept extremely simple. All complexity is pushed to the batch layer.

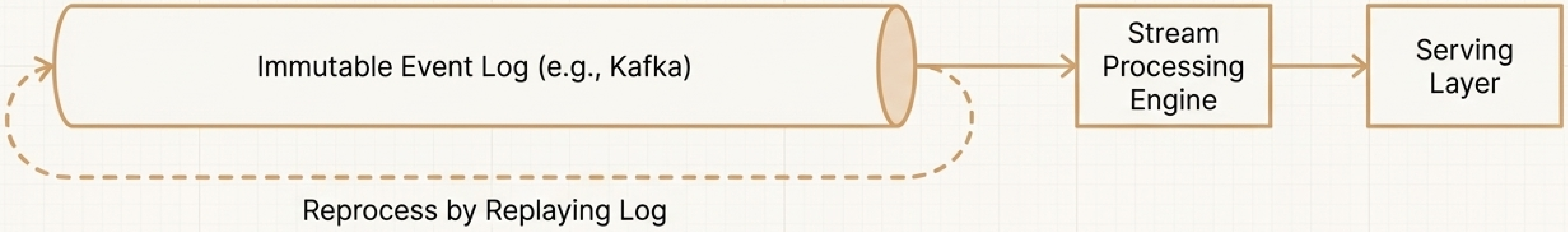
Be Ready to Repopulate:

"You must have all the buttons at your fingertips to re-query the speed layer and overwrite it with data from the batch layer. Errors will happen." The batch layer is the ultimate source of truth and recovery.

A Reaction to Complexity: The Kappa Architecture

Proposed by Jay Kreps (a co-creator of Apache Kafka).

“Why not just use a stream-processing platform as the backbone for all data handling?”



How it Works

- **No Batch Layer:** The batch layer is eliminated entirely.
- **Unified Logic:** There is only one codebase to maintain—the one for the stream processor.
- **Reprocessing via Replay:** To correct errors or compute new views, you deploy new code and simply replay the event log from the beginning (or a relevant checkpoint) through the stream processor.

A Strategic Comparison: Lambda vs. Kappa

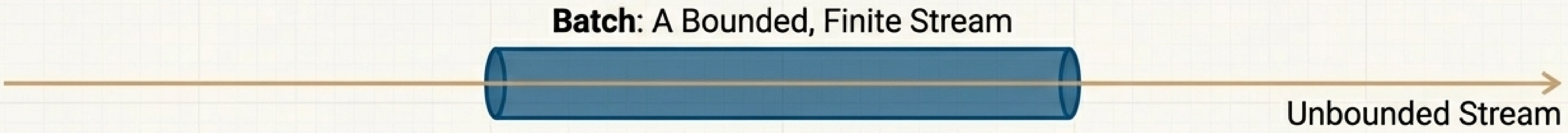
Characteristic	**Lambda Architecture**	**Kappa Architecture**
Core Idea	Separate batch and speed processing paths.	A single, unified stream processing path.
Layers	Batch Layer + Speed Layer + Serving Layer	Stream Processing + Serving Layer
Source of Truth	The master dataset in the batch layer (e.g., HDFS/S3).	The immutable event log (e.g., Kafka, Pulsar).
Complexity	**High**. Two codebases and two systems to maintain and debug.	**Lower**. A single codebase and processing paradigm.
Maintenance	**Difficult**. "Maintaining two different systems is not easy."	**Easier**. A single system to manage and evolve.
Reprocessing	A batch job recalculates aggregates from the master dataset.	The entire log is replayed through the stream processor.
Latency	Milliseconds (from speed layer) + Hours (from batch layer).	Milliseconds to seconds.
Best For	Systems requiring both precise, complex batch reports and real-time dashboards. BI analytics.	Real-time-centric use cases: IoT, monitoring, clickstream analysis, fraud detection.

The Modern Resolution: Unifying Batch and Stream

The Problem with the Debate:
“Both Lambda and Kappa sought to address limitations of the Hadoop ecosystem of the 2010s by trying to duct-tape together complicated tools.” — Fundamentals of Data Engineering

The Philosophical Shift:
More recent work has shown that the distinction between batch and stream processing processing is artificial. The core insight is:

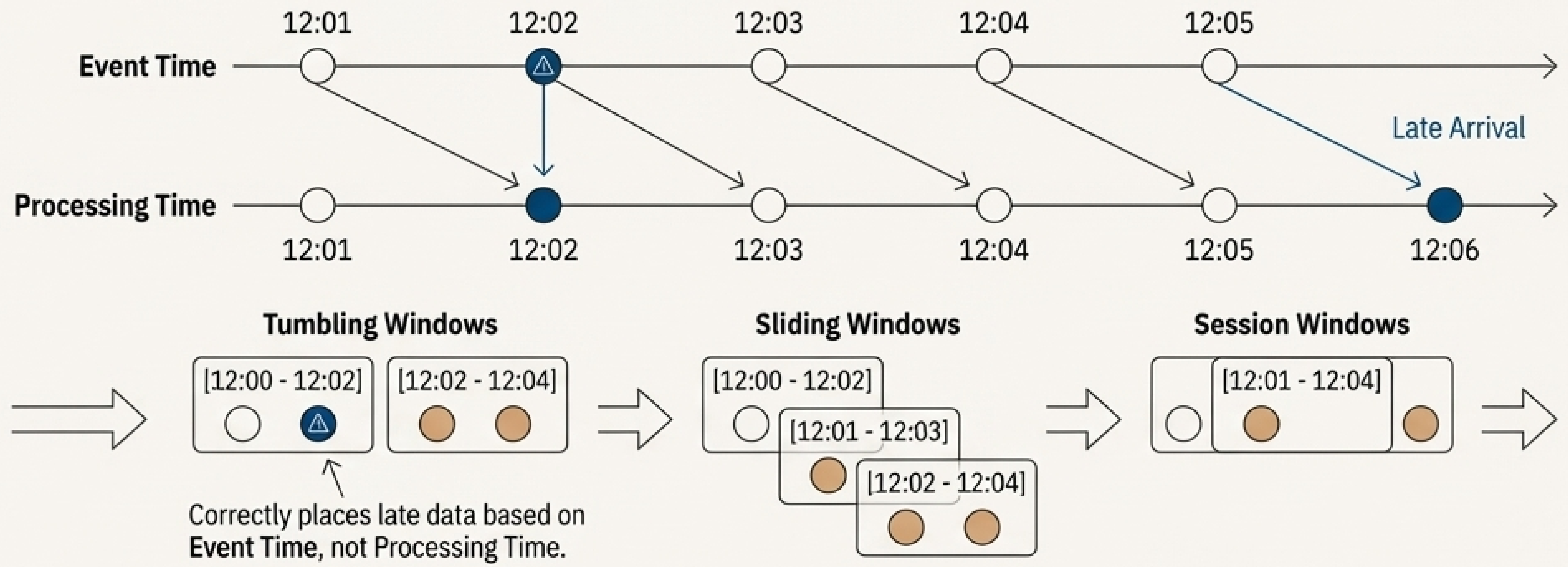
“Batch is just a special case of streaming.”



- A stream is simply an unbounded series of events over time.
- A batch is a bounded, finite series of events.
- Therefore, a unified system can treat all data as an event stream, and batch processing simply becomes processing a stream with a defined start and end.

The Enabling Technology: The Dataflow Model

Pioneered by Google and implemented in open-source as Apache Beam.



How it Works & Impact

- **Unified API:** Provides a single programming model to express data processing pipelines.
- **Event Time Windows:** It decouples *when* data occurred (event time) from *when* it is processed (processing time). This is critical for correctly reprocessing historical data.
- **Unified Engine:** The same code can be executed by a runner (like Flink or Spark) to process both **Unbounded Data** (streams) and **Bounded Data** (batches).

Impact Statement – This approach finally solves the "Duplication Tax" by allowing batch and stream computations to be implemented in the same system with nearly identical code.

Guiding Principles for Modern Data Architecture

Solve Business Problems, Don't Just Implement Patterns

"Always prioritize requirements over building something cool."
Avoid "resume-driven development" where technology choices are made for their novelty rather than their fit.

Favor Simplicity and Modularity

As seen with Avito, complex systems are better built from simple, composable components than as a single monolith. Isolate complexity.

Unify Where Possible

The trend is toward convergence. A unified model for batch and streaming (like Apache Apache Beam) eliminates the primary weakness of the Lambda architecture—the "Duplication Tax."

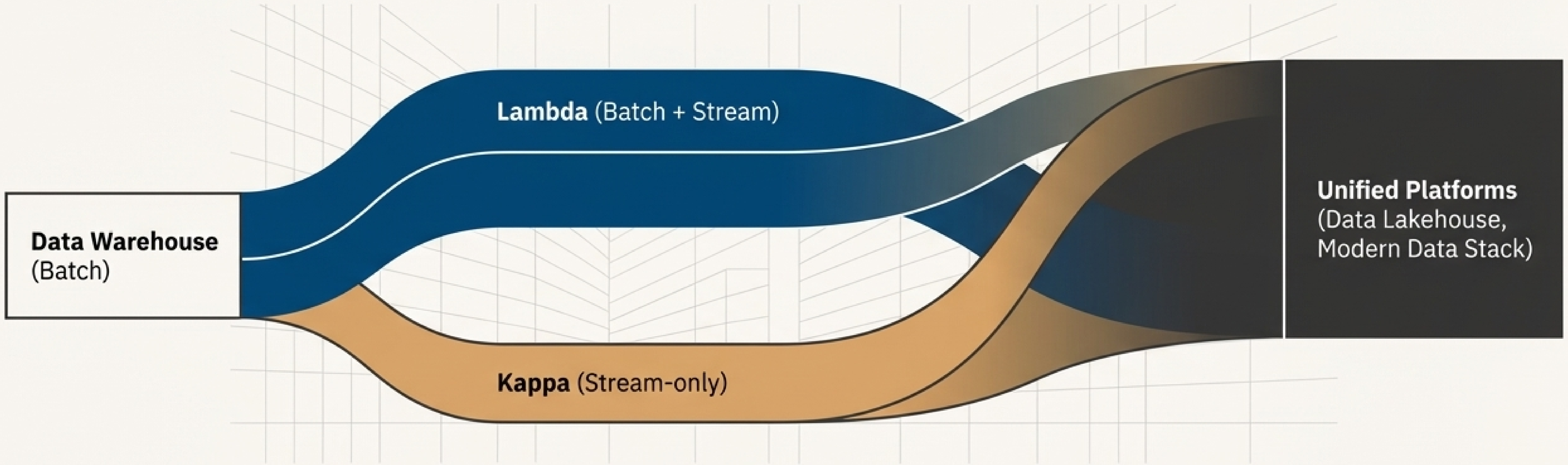
The Event Log is the Source of Truth

The Kappa architecture's core idea of an immutable, replayable log is foundational to modern, resilient systems. It enables reprocessing, debugging, and system evolution.

Be Prepared for Evolution and Reprocessing

Systems and requirements change. The ability to reprocess historical data to derive new views or fix mistakes is not a "nice-to-have," it is a core mechanism for system maintenance and evolution.

The Enduring Legacy of Lambda and Kappa


Data Warehouse (Batch)
Lambda (Batch + Stream)
Kappa (Stream-only)
Unified Platforms (Data Lakehouse, Modern Data Stack)

The Takeaway

Lambda and Kappa were not final destinations, but critical steps in the evolution of data architecture.

They forced the industry to grapple with the challenges of combining batch and real-time processing, the importance of immutable data, and the power of reprocessing.

The core principles they pioneered—deriving views from an immutable event stream and reconciling speed with accuracy—are now fundamental concepts embedded in modern data platforms like the Data Lakehouse, which aims to provide the best of both worlds on a single, unified foundation.

Understanding their history is key to understanding the 'why' behind the architecture of today.