CSCI E-103

Data Engineering for Analytics to Solve Business Challenges

ETL
Batch & Streaming

Lecture 03

Anindita Mahapatra & Eric Gieseke

Harvard Extension, Fall 2025

Agenda

- Review previous class
- Data Pipeline
- Need for Streaming
- Streaming Concepts
- Unifying batch and streaming
  - Lambda Vs Kappa architectures
- Handling Failures
- Streaming Tradeoffs
- Streaming Best Practices

- Lab02

- Assignment 2
  - Will be published prior to Thursday's section
  - Review during section

Review Lecture-2 (part 1)

3NF (Third Normal Form)	Refers to Relational Data modeling used to reduce data duplication and achieve data integrity.
Denormalization	Usually followed for non-relational stores for faster data access at the cost of data redundancy
ETL	Extract, Transform, Load variation is ELT
What are the 2 popular data warehouse data mart schema designs	Star & Snowflake Facts in center flanked by Dimensions
Example of Key Value store	S3
Example of Document Store	MongoDB, CouchDB
Example of Columnar Data Store	DynamoDB, Cassandra

Review Lecture-2 (part 2)

Why are text formats like CSV, JSON less preferred over binary formats like avro, parquet

Compression, performance, flexibility

Parquet is what format? what is the main advantage ?

Columnar, binary
more performant

List 4 main properties of Delta format

Supports ACID transactions
Allows for schema evolution
Offers fine grained inserts, deletes
Maintains data version aka time travel

What is metadata?

Data about data, data context
Ex. Schema
Transaction log of delta

Tables

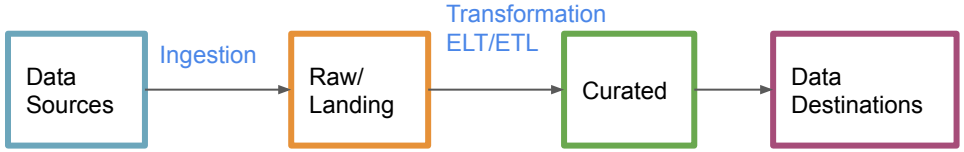
- Permanent Table
  - Persistent storage
- Temporary Table
  - local : Exists only for the duration of a session
  - global (shared): Shared across sessions in the same cluster
- Views (virtual table)
  - Select query off 1 or more tables
  - Stores only the **query definition**, not the actual data.
  - Every time you query the view → the underlying query runs on the base tables.
- Materialized Views (MV)
  - A precomputed table of query results, so results may be stale
  - Refreshable, faster than recomputing queries repeatedly.
  - Very fast for repeated queries, especially aggregates/joins. Used in BI reports
- Streaming Table(ST)
  - Continuously updated as new events arrive (used in streaming pipelines)

Tables: Managed Vs External

Feature	Managed Table	External Table
Data location	Default warehouse path	User-specified external path
Ownership	Database manages data + metadata	Database manages only metadata
DROP behavior	Deletes both metadata and data	Deletes only metadata (files remain)
Data sharing	Not easily shared	Ideal for sharing across systems
Risk of accidental loss	Higher (drop = permanent data loss)	Lower (data persists even if table is dropped)

Code	CREATE TABLE sales (
	  id INT,
	  amount DOUBLE
	);
		CREATE TABLE sales_external (
		  id INT,
		  amount DOUBLE
		)
		USING PARQUET
		LOCATION 's3://company-data/sales/';

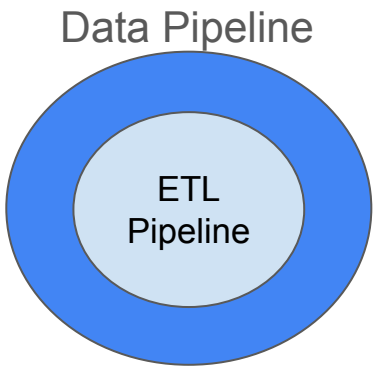
Data Pipeline



A data pipeline might be as simple as moving data from point A to point B, and as complex as gathering data from multiple sources, transforming it, and storing it in multiple destinations.

A **data pipeline** is an artifact of data integration and data engineering processes.

ETL pipelines transform raw data into data ready for analytics, applications, machine learning and AI systems to solve problems, inform decisions, and, make our lives more convenient.



Data pipeline = general plumbing for data movement.

ETL pipeline = specialized pipeline that cleans & reshapes data for analytics.

Data Pipeline

Process of building a pipeline:

- Design, Implement, Test, Deploy, Monitor
- Handle Changes, Redeploy
- Automation
- Trigger (file, schedule, manual)

Advantages:

- Reduce dependency on IT folks by making data available via self-serve channels
- Help accelerate cloud adoption with dynamic resources
- Support real time analytics and applications that drive real time decisions

Challenges:

- Continuously chasing data issues including schema and quality changes (data drift)
- Fixing these issues can cause outages & delays to existing jobs
- Tied tightly to infrastructure, process, technology and can be vulnerable to changes there

Comparison of Pipeline Types

Pipeline Type	Processing Style	Key Tools	Typical Use Case
Batch	Bulk, scheduled (hourly, daily, weekly)	Apache Spark, AWS Glue, Talend	Daily reports, billing cycles, historical analysis
Streaming (Real-Time)	Continuous, event-driven	Apache Kafka, Flink, AWS Kinesis, Databricks Streaming	Fraud detection, real-time dashboards, recommendations, IoT
ETL (Extract → Transform → Load)	Batch (traditionally)	Informatica, Talend, Matillion, dbt	Clean & load CRM/ERP data into a data warehouse
ELT (Extract → Load → Transform)	Batch or near real-time	dbt, Snowflake, Databricks	Load raw logs into warehouse and transform later
Data Replication	Near real-time or scheduled	Fivetran, Airbyte, AWS DMS	Syncing transactional DB to reporting DB
ML Pipeline	Batch or real-time	MLflow, Kubeflow, Vertex AI	Training/deploying models (e.g., churn prediction)
Workflow / Orchestration	Meta-pipeline (manages dependencies)	Apache Airflow, Prefect, Dagster	Triggering ETL, monitoring jobs, updating dashboards

Streaming Data Pipeline

- *Why?* To provide insights faster by turning event streams into analytics-ready data
- Stream processing focuses on the real-time processing of continuous streams of large amounts of data in motion to
  - collect
  - analyze
  - store
- Stream processing naturally fits with time series data and supports detecting patterns over time.
- For some scenarios, streaming is a must, ex.
  - Sensor data, ad data, server security logs, click stream data, payment transactions, etc,
  - In some others, it is not but every batch job can be considered as a streaming job with a longer trigger interval

The modern data platform is built on business-centric value chains rather than IT-centric coding processes

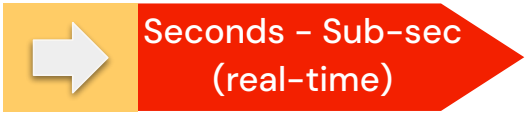
Streaming Concepts

Sources & Sinks/Targets	Every streaming pipeline has 3 elements: source, processing, and sink
File based Vs Event based	Landing to disk first and then ingesting Vs doing it on the fly in-stream
Micro-batch Vs continuous streaming	Batch for a few sec or millisec Vs continuous meaning as soon as data hits
Processing trigger	The microbatch interval in the readStream
Output modes	Option specified in writeStream to append/overwrite/update data
Checkpoint	Recover from failure by remembering the last known offset processed
Window	Aggregation interval for counting, averaging, etc for ready to consume metrics
Watermark	How long to tolerate arrival of late data before closing the gates and dropping data
Stream Operations	All transformations to wrangle streaming data
In-Stream Analytics	On the fly transformations on the moving data without going to disk first

Use Cases

- ETL
- Metering and billing
- Ad hoc analytics
- Business Intelligence

- Mobile and IoT data capture
- Service monitoring and alert
- Log ingestion
- Digital advertising
- Clickstream analytics

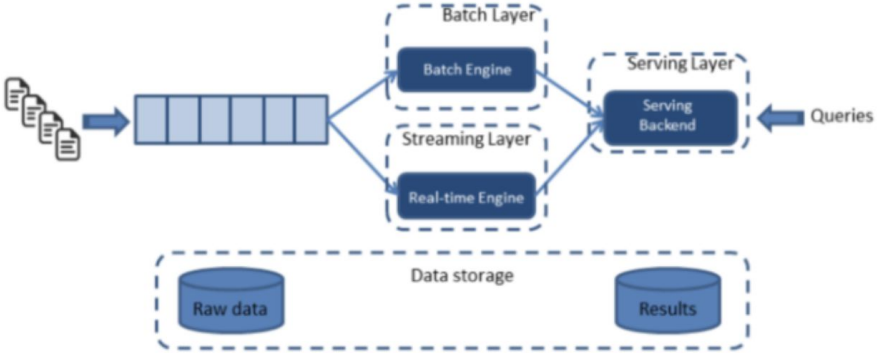


- Fraud detection
- Financial asset trading
- Multiplayer gaming
- ML inference

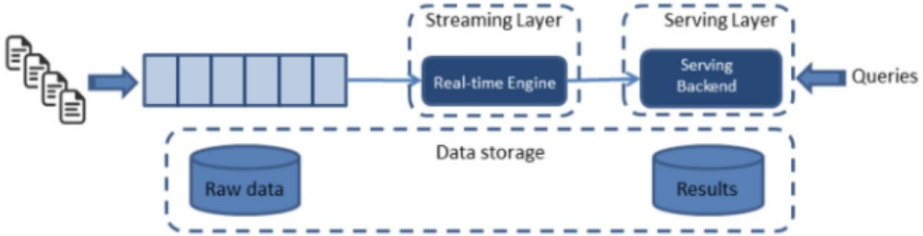
Unifying batch and streaming

λ vs 𝜅

Reference

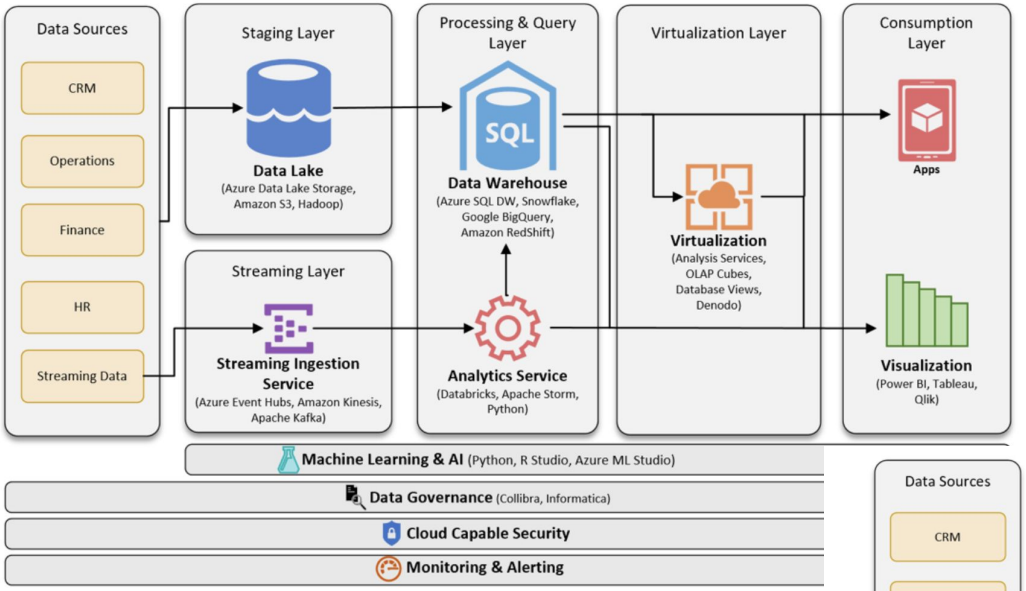


Lambda Architecture



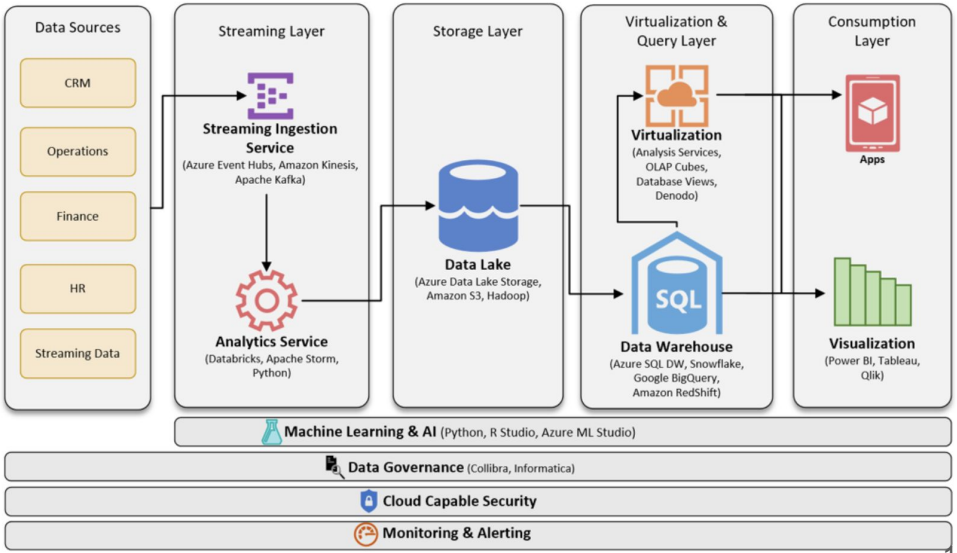
Kappa Architecture

Lambda	Kappa
• Batch + Streaming • Good balance of speed and reliability • Data processing duplication • Batch is reliable & Streaming is approximate • Works better for historical load re-processing in Batch + Speed + Serving layers	• Batch treated as streaming • Doesn't need a separate batch layer • Horizontally scalable, fewer resources, less reconciliation • Streaming with Consistency (exactly once) • Works well with high-speed stream processing engine to enable low latency in the processing



Lambda architecture example

Examples of Lambda & Kappa in Azure



Kappa architecture example

Streaming Processing Frameworks

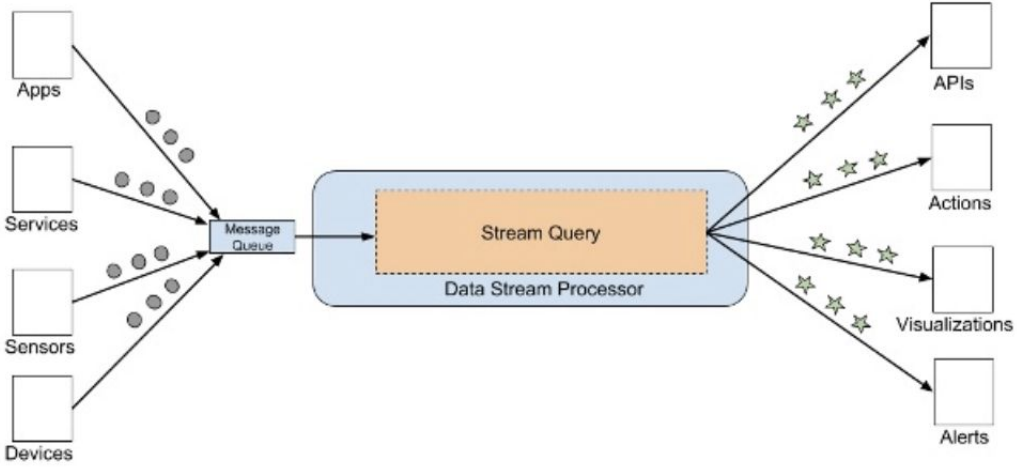
Framework	Processing Model	Latency	State Management	Fault Tolerance	Ease of Use	Ecosystem / Best For
Kafka Streams	Event-at-a-time	Low (<10ms)	RocksDB-backed	Yes (Kafka log)	Easy (Java lib)	Apps already on Kafka needing lightweight stream processing
Apache Flink	True streaming (event-time)	Very Low (ms)	Advanced (large state, checkpoints)	Strong (exactly-once)	Medium (Java/Scala API)	Complex stateful processing, CEP, large-scale analytics
Spark Structured Streaming	Micro-batch (default), continuous mode	Higher (100ms – sec)	State store via checkpointing	Strong (exactly-once)	High (SQL/DataFrame API)	Teams already using Spark for batch + streaming (unified API)
Apache Storm	Event-at-a-time	Very Low (ms)	Limited	Basic (acks)	Hard (manual topologies)	Legacy systems, very low latency, not widely adopted now
Apache Samza	Event-at-a-time	Low	RocksDB-backed	Yes (YARN/Kafka)	Medium	Kafka + YARN shops, but niche today
KSQLDB	Continuous SQL over Kafka	Low	Internal	Inherits from Kafka	Very Easy (SQL)	Real-time analytics, Kafka-first teams without coding
Amazon Kinesis	Micro-batch (100ms–sec)	Medium	Managed	Managed	Easy (AWS native)	AWS-native apps, IoT, clickstreams
Google Dataflow (Beam)	Unified batch + streaming	Low–Medium	Advanced (stateful operators)	Strong	Medium (Java/Python SDK)	GCP-native, multi-cloud portability via Apache Beam
Azure Stream Analytics	SQL-like streaming	Medium	Limited	Managed	Very Easy (SQL)	Azure-native, low-code teams

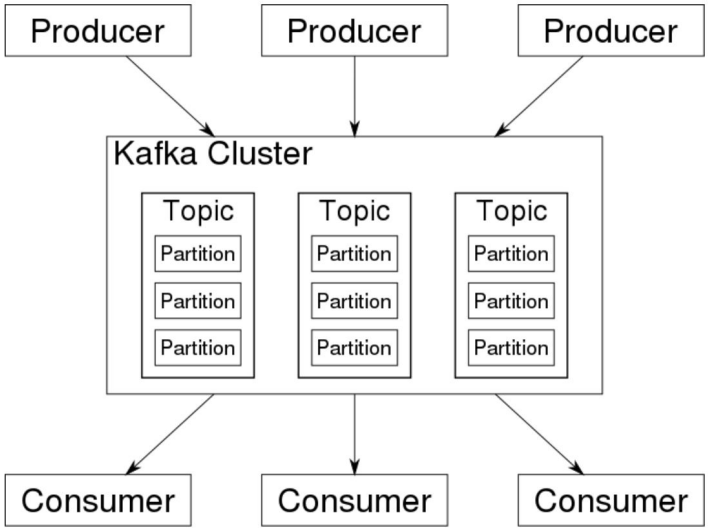
Streaming Processing Frameworks

- **Apache Kafka Streams** – lightweight library for stream processing directly on Kafka topics
- **Apache Flink** – advanced stream (and batch) processing engine, strong for stateful and event-time processing.
- **Apache Spark Structured Streaming** – micro-batch or continuous mode, strong integration with the Spark ecosystem.
- **Apache Storm** (older, less used now) – early distributed stream processor, low latency but harder to manage.
- **Apache Samza** – built by LinkedIn, runs on Kafka + YARN, less popular now.
- **KSQLDB** – SQL-based streaming built on top of Kafka.
- **Amazon Kinesis** – AWS-managed stream processing service.
- **Google Cloud Dataflow** – GCP-managed service based on Apache Beam.
- **Azure Stream Analytics** – fully managed Azure SQL-like streaming engine.

Ultra-low latency & stateful CEP → **Apache Flink**.
Best for Kafka-only teams → **Kafka Streams** or **KSQLDB**.
Unified batch + streaming (lakehouse) → **Spark Structured Streaming**.
Fully managed cloud-native → **Kinesis (AWS)**, **Dataflow (GCP)**, **Azure Stream Analytics (Azure)**.
Legacy but low-latency → **Apache Storm** (rarely chosen today).

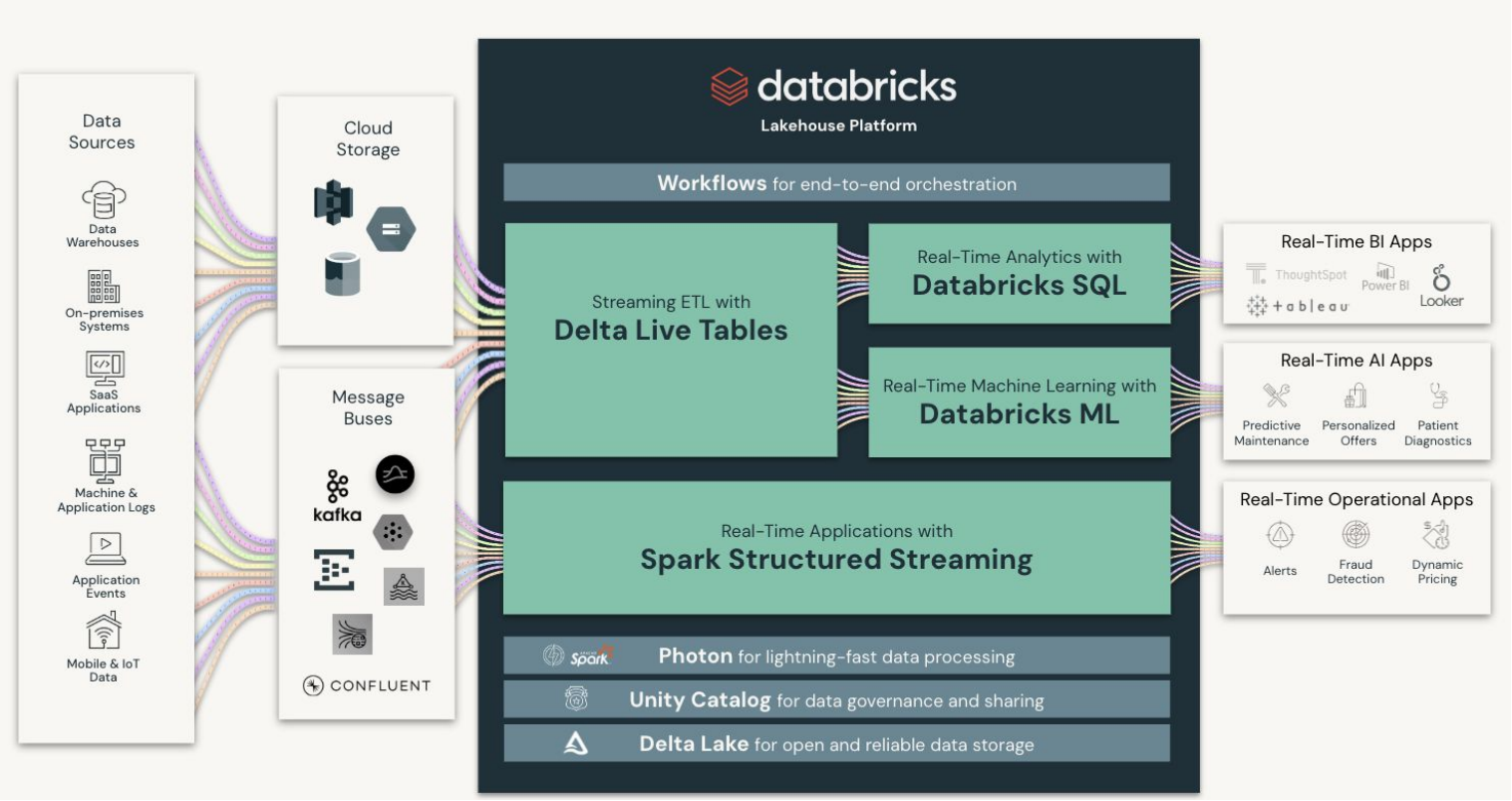
Stream Ingestion (Event & File)





- First Gen Msg Brokers:
  - ActiveMQ, RabbitMQ, etc based on MOM (Message Oriented Middleware)
- Later Gen: Stream processors
  - ex. Apache Kafka, Amazon Kinesis Data Streams
    - support very high performance with persistence,
    - have massive capacity of a Gigabyte per second or more of message traffic,
    - and are tightly focused on streaming with little support for data transformations or task scheduling

Streaming on Databricks



Structured Streaming Example

```
spark.readStream.format("kafka|kinesis|socket|...")
.option(<>,<>)...
.load()
```

source

```
.select(cast("string").alias("jsonData"))
.select(from_json($"jsonData",jsonSchema).alias("payload"))
```

transformation

```
.writeStream
.format("kafka")
.option("kafka.bootstrap.servers",...)
```

sink

```
.trigger("30 seconds")
.option("checkpointLocation",...)
.start()
```

config

LakeFlow

The evolution of data engineering on Databricks

Ingest Transform **Orchestrate**

LakeFlow Jobs

Orchestrate any production workload

High reliability

Full observability and monitoring

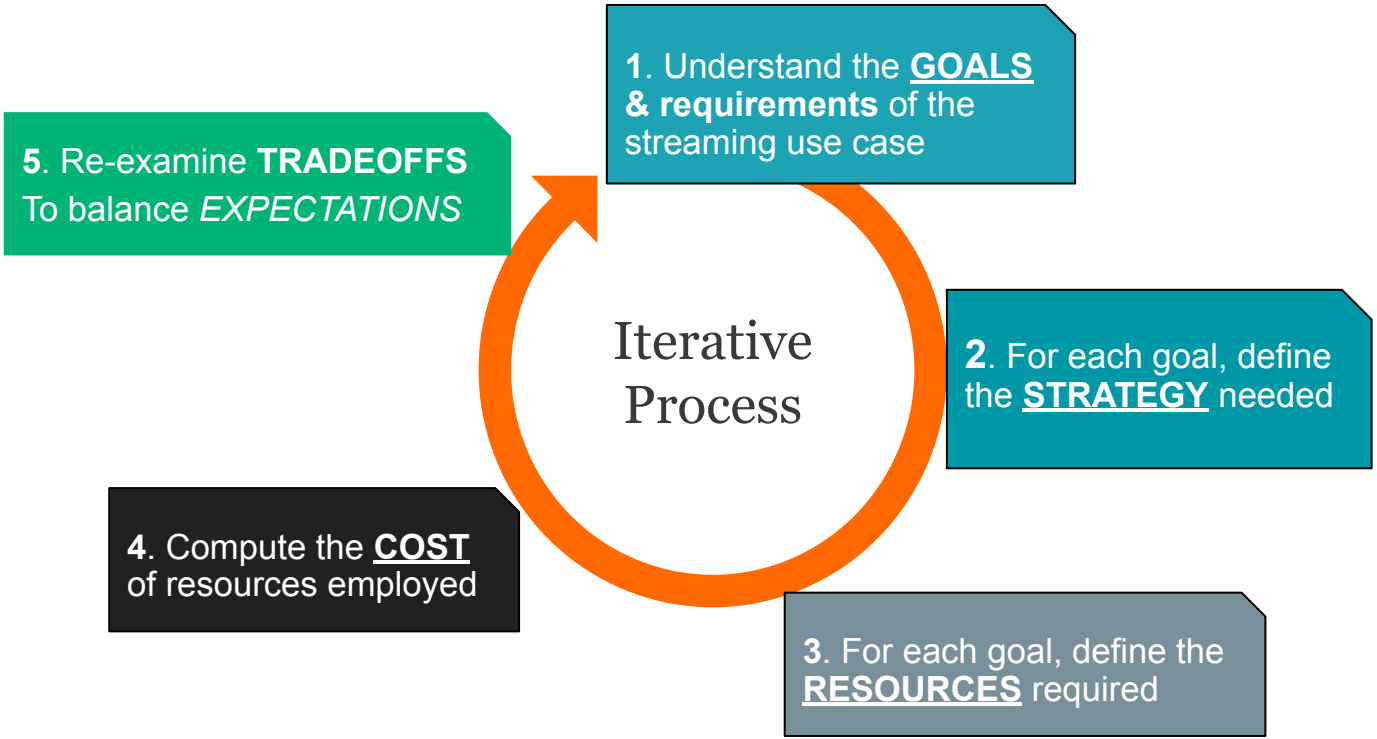
Built with the latest innovations of **Databricks Workflows**

File Arrival Triggers	Conditional Tasks (if/else)	Serverless Compute	Job Level Parameters
Webhooks	Job Queues	Continuous Execution	Task Looping (forEach)
Table Triggers	Visual Monitoring	Jobs data in system tables	Modular Workflows
Duration Alerts	Gantt Visualization	SQL Tasks	Databricks Asset Bundles

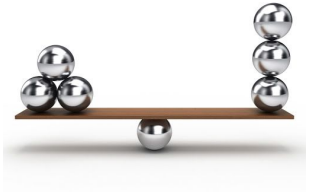
What kind of tradeoffs?

Isn't it just supposed to work?

Spark has multiple ways of achieving the end goal with **tunable** performance, cost and quality



Balancing Goals with Resources



Goals/Requirements/Needs

- Scalability
  - Data Volume
  - TPS
- Performance *(measure of scalability)*
  - E2E Latency
- Processing
  - Transformations (hops, sinks, joins)
  - Stateful Processing (late data)
  - Reprocessing Logic
- Quality
  - Data Correctness - no drops nor dups
  - Towards Idempotency
- Reliability/Availability
  - Failure Handling/Recovery
  - Disaster Recovery

Resources

- Compute
  - Number & type of VMs, Pools
  - DBUs
- Storage
  - Type
  - Configuration
- Services
  - Kafka, Kinesis, Event Hub, IoT Hub
  - API Calls (rate limits, batch)
- Effort
  - PeopleHours & Skills
  - To develop, manage & maintain

Debug a scenario

- Use Case
  - Network Threat Detection
- Scenario (Goal/Requirement)
  - Data ingested from Event Hub at low frequency, transformed, aggregated
  - Models were applied to predict threat detection patterns and suggest recommendation
- Customer Pain
  - "*After spending way too much on storage yesterday we shut down the 24/7 processing this morning. Storage was 70%, VMs were 30%*"

- High egress cost
- Low streaming volume caused lots of very small files resulting in high compute cost.
- High storage access cost.

Cost Tradeoffs

Access Tier

- Cool Vs Hot

Replication for disaster recovery

- Locally Redundant Storage (LRS) Vs Geo Redundant Storage (GRS)

- **Storage**: When data needs to be archived, 'cool' storage is used since it is less expensive than warm
- **Access**: However, if data needs to be frequently accessed, access from cool storage is more expensive
- Active data should be retained in 'warm' storage

Processing Trigger Interval can be used to control cost

Data Reprocessing Options

1) Data Catch Up
(After a system outage)

- Scale up to large cluster
- Adjust throttles if required
- Clear the backlog asap
- This is possible if the additional time required to clear the backlog can be tolerated
- Also the partitioning should foster increased compute

2) Skip to Live Stream
(Backed up data is lower priority)

- In general, this is not a good idea as it affects checkpoint
- Spin up 2nd cluster
  1) to handle from offset T+
  2) older cluster continues as before trying to catch up

  Till they eventually catch up

  One cluster can be shut

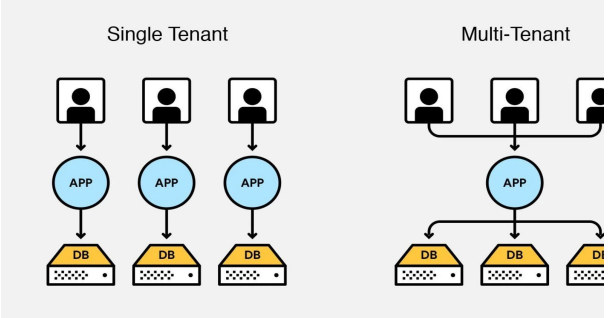
3) Logic Change
(Complete Reprocess)

- This is not time sensitive
  Let Structured Streaming do the heavy lifting
- Separate job on separate cluster to separate sink.
- Allow 2nd job to catch up
- Stop first job
- Swap sinks or adjust consumer configuration to correct sink

Reading Link: Recovering from failures with **checkpointing**

There are limitations on what changes in a streaming query are allowed between restarts from the same checkpoint location

Scenario: Multi-Tenancy



- Goal:
  - Satisfy client's need for data isolation
- Strategy
  - On-prem, they had a separate node per client
  - So in the cloud, they started with creating a separate job and cluster for each of their clients
- Consequence
  - High cost and poor resource utilization
  - High maintenance on large number of jobs from both code and devops perspective
- Recommendation:
  - Single job/cluster for getting the data ingested into common Delta table partitioned by client
  - This will eventually cause driver load and it might be necessary to assign multiple such clusters
  - Separate database for each client in serving layer where aggregated data is stored to feed reporting dashboards
  - Views on the data available for individual clients who wish to see data in raw zones

Scenario: Deduplication

- Goal:
  - Report accurately on the number of unique callers in the reporting interval
- Strategy
  - Use **watermarking** (to deal with permissible lateness of incoming data)
  - Use dropDuplicates
- Consequence
  - Incorrect use of watermarking
    - statestore size was eventually blowing up leading to OOM errors
    - It takes a while to reach this state, so was detected while doing stress testing
  - By default, spark remembers all the windows forever and waits for all late events
    - For small data volume it is ok, over time resource utilization spikes
- Recommendation:
  - Include a watermarked timestamp column, to prevent the state to grow unbounded
  - Repartitioning after dedup is a good idea
    - The number of partitions in the state store should never change for any stateful processing.
    - foreachBatch + MERGE INTO is slower but preferable
  - *compromise the deduplication results (where some late events were dropped)*
  - Further Reading (*on conditions for watermarking to clean aggregated state)*

```
uniqueVisitors =
     .withWatermark("ET", "10
minutes")
     .dropDuplicates("ET", "uid")
```

Streaming Best Practices

- Understand data processing requirements & SLAs
  - Ask the right questions around speed of data arrival (TPS - transactions per second)
  - Understand end to end latency requirements
    - If ASAP -> ask how the data will be consumed and by whom
  - Get a ballpark cost estimate
  - List existing technologies to understand integration points
- Always use Checkpoint
  - Choose Sql.shuffle.partitions
  - checkpointing happens for each aggregation for each sql shuffle partition.
- Introduce processing trigger interval
  - To avoid frequent checking and increase in storage API costs
- Optimized writes
  - delta.autoOptimize.optimizeWrite = true to reduce the number of files written
  - To avoid recomputations, we should cache the output DataFrame/Dataset, write it to multiple locations, and then uncache it
- Planning Capacity
  - How many job definitions/runs/streams per cluster
  - Mux - demux architecture - where everything lands in bronze and is then split

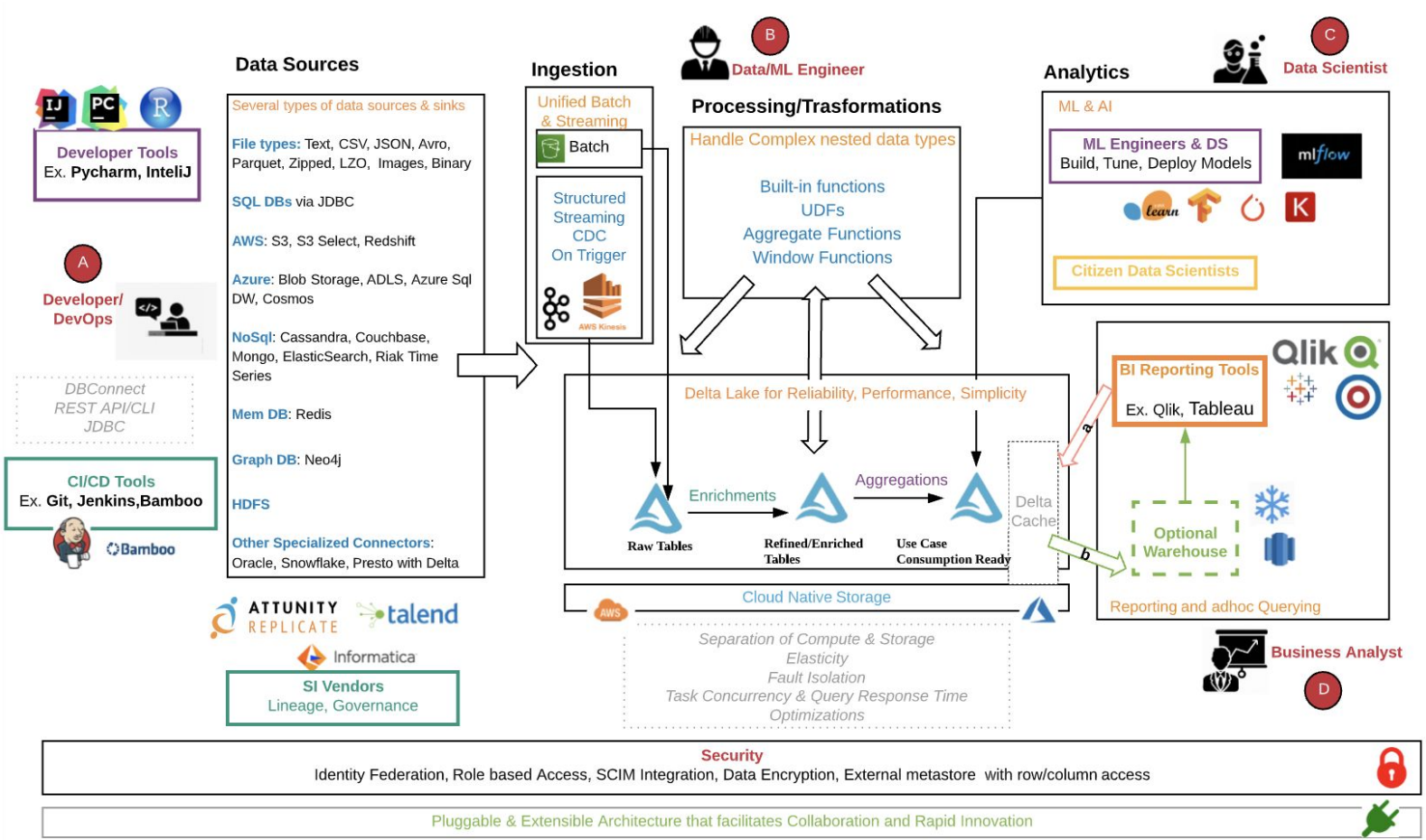
Reminders

Reading: through ch-5

Lab02 under https://canvas.harvard.edu/courses/164104/files/folder/Labs

Appendix

ETL Architecture Blueprints



References

Documentation and best practice reference

- Structured Streaming Programming Guide
- Structured Streaming in Production (important!)

Talks:

- Deep Dive into Stateful Stream Processing in Structured Streaming
- Designing Structured Streaming Pipelines (many common patterns explained!)
- Productizing Spark Structured Streaming

Scientific Papers:

- The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale Unbounded, Out-of-Order Data Processing
- Streams and Tables: Two Sides of the Same Coin

Blogs:

- Streaming 101: The world beyond batch
- Streaming 102: The world beyond batch