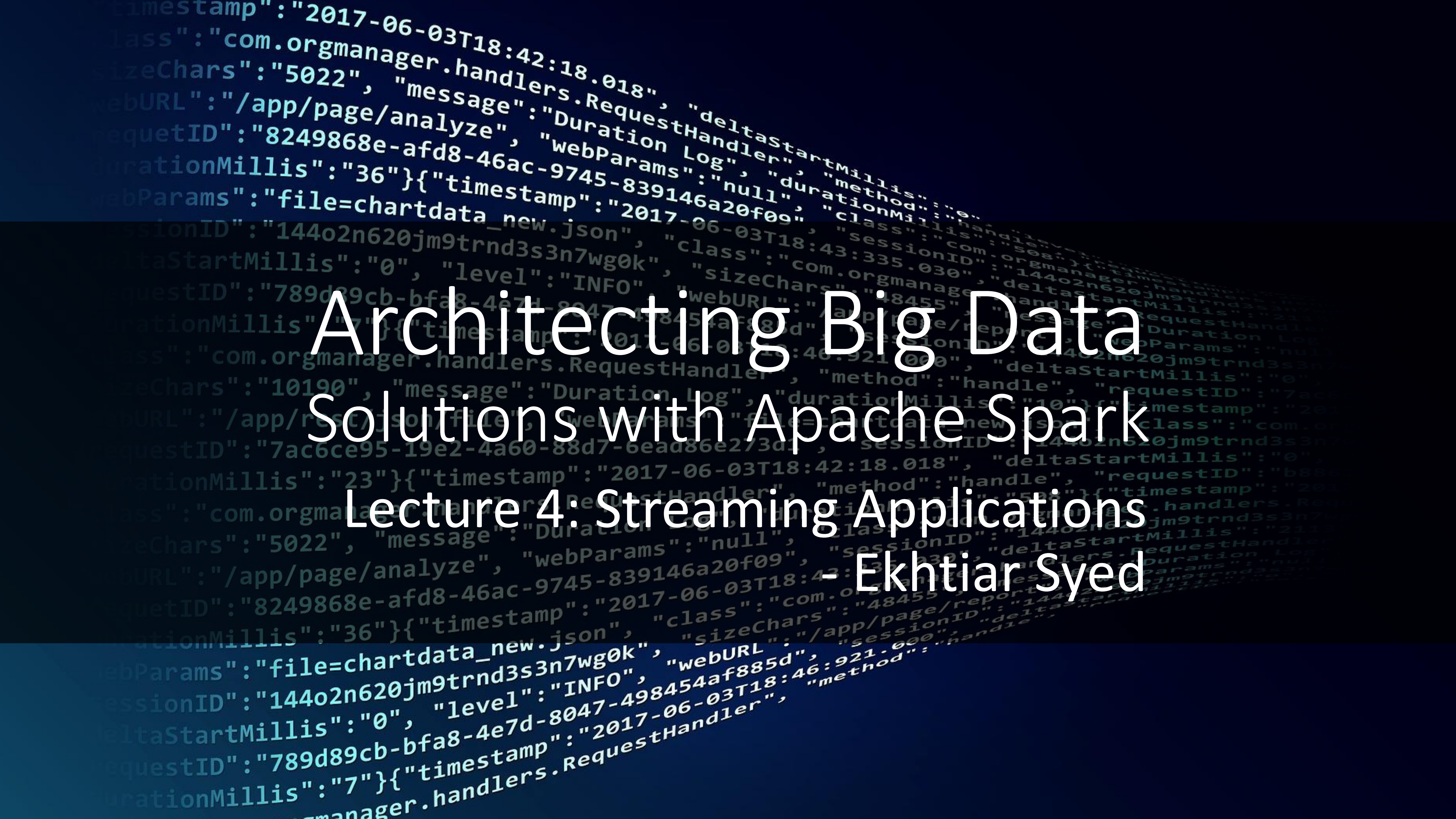


Architecting Big Data Solutions with Apache Spark

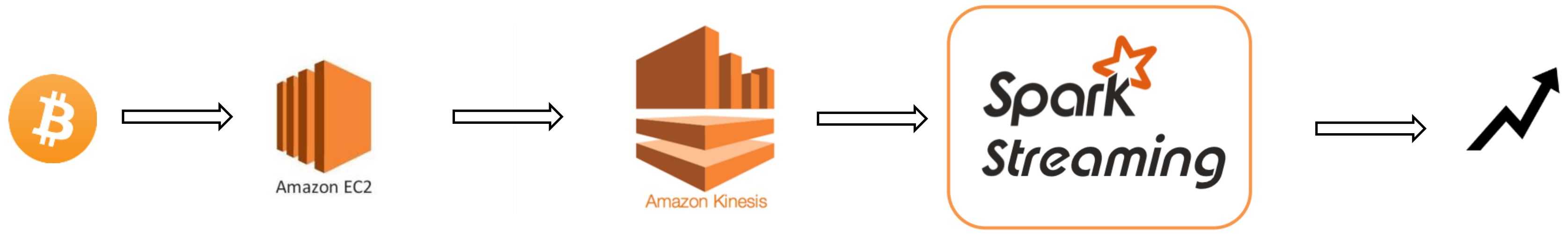
Lecture 4: Streaming Applications

- Ekhtiar Syed

Streaming Live Data

- So far, we covered data processing in batch.
- We will now focus on live streaming data processing with Spark.
- We are now looking at processing continuously flowing data and detecting facts and patterns on the fly.
- We are navigating from a lake to a river.
- Often we will want to deploy our models to make predictions on data "as it happens"
- Operating on a stream of data presents some unique challenges including:
    - Rebuilding your model to reflect the changing world
    - Deploying models that can run quickly
    - Scalability and Fault Tolerance

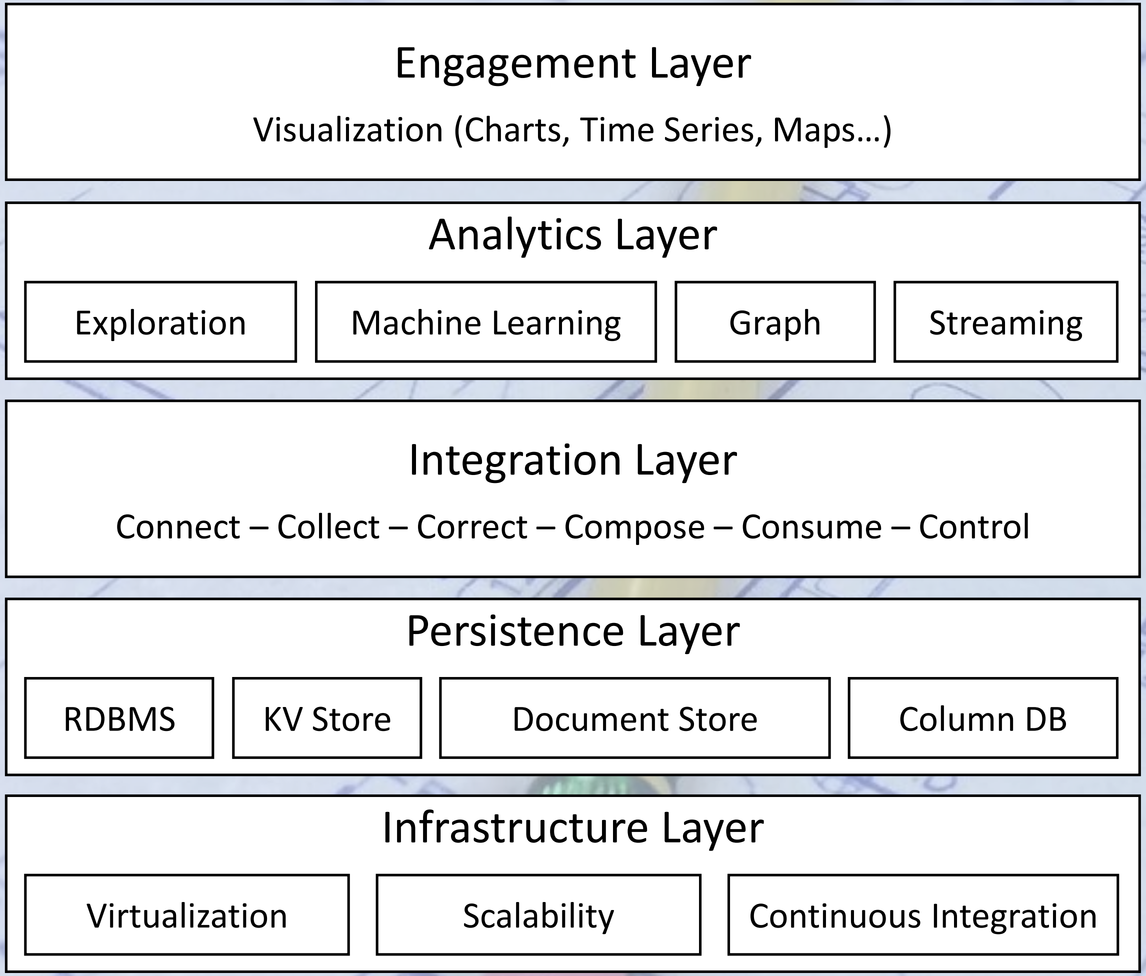
Live Example – Bitcoin Price Pipeline



Now I will demo a data real time data pipeline, which we will work on next week!

This data pipeline has streaming data of Bitcoin Price Index (BPI) in Euro, USD, and GBP. First we have a producer running in an EC2 instance (in AWS). This pushes BPI data to a message queuing system In AWS called Kinesis. Then we have a Streaming Query in Databricks to Visualize the rise and fall of Bitcoin price in Euro in real Time.

Layering The Foundation - DIA Architecture



Source: Spark for Python Developers – Amit Nandi

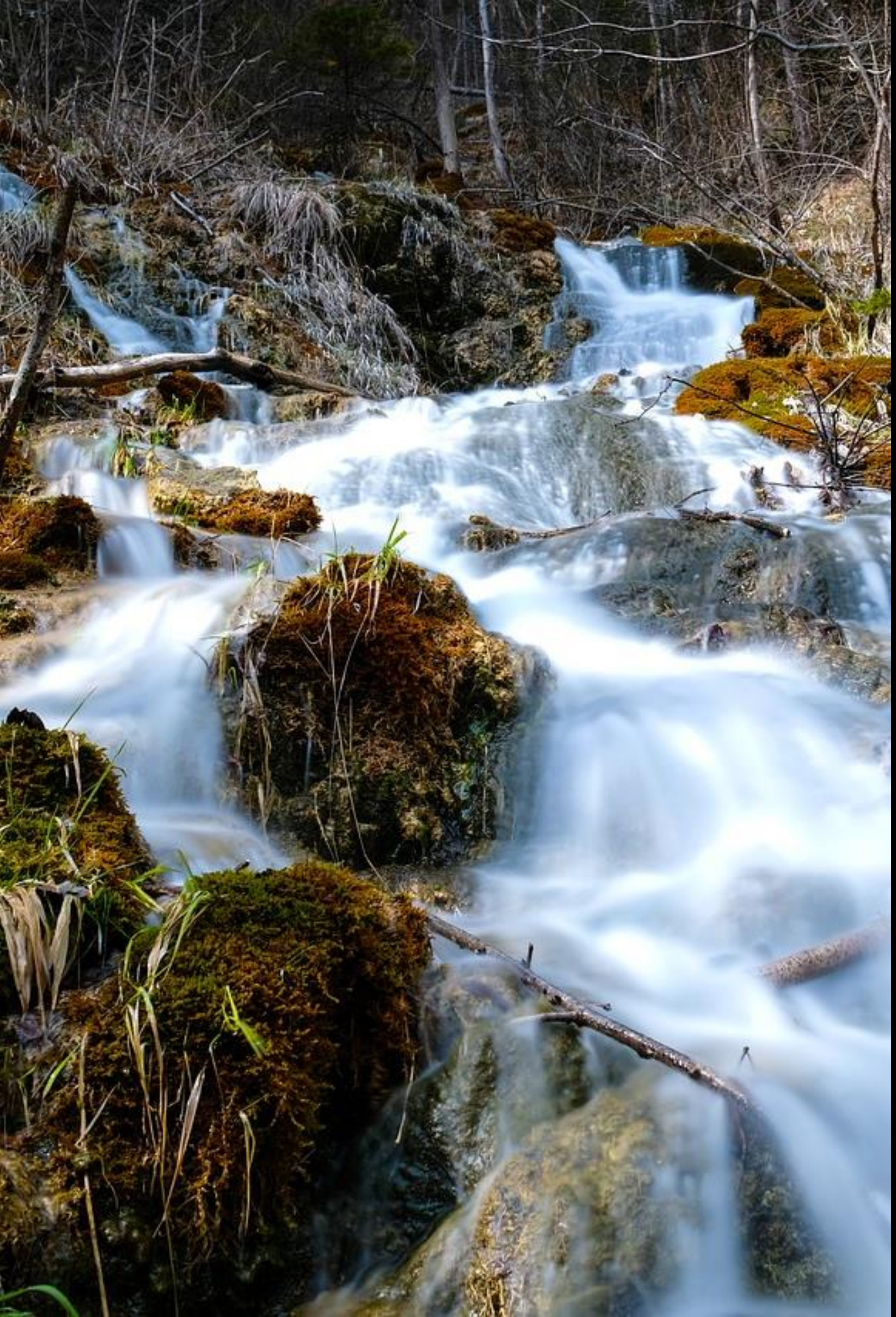
Let's first go back to our original drawing of the data-intensive apps architecture blueprint and discuss challenges of going real time!

Process data in real time with acceptable latency and throughput to ensure the right value of data.

Collect data at speed in real time and facilitate consumption of data to all downstream sources in a fault-tolerant manner.

Persistence Layer: High Read / Write throughput is required for good performance of Real-Time Application.

Infrastructure Layer: Needs to be robust, and capable to handle incoming data at variable speed.



Real Time Data Processing (Analytics Layer)

Two models emerge for processing streams of data:

- Processing one record at a time as they come in. We do not buffer the incoming records in a container before processing them. This is the case of Twitter's Storm, Yahoo's S4, and Google's MillWheel.
- Micro-batching or batch computations on small intervals as performed by Spark Streaming and Storm Trident. In this case, we buffer the incoming records in a container according to the time window prescribed in the micro-batching settings.

Which paradigm to choose depends on the use case. For example, in the case for fraud detection and prevention, mobile cross-sell and upsell, or traffic alerts; every millisecond counts!!

Real Time Data Processing (Analytics Layer)

Spark Streaming is based on micro-batching. Storm is based on processing records as they come in.

The driving factor in a streaming application is latency. Latency varies from the milliseconds range in the case of RPC (short for Remote Procedure Call) to several seconds or minutes for micro batching solution such as Spark Streaming.

RPC allows synchronous operations between the requesting programs waiting for the results from the remote server's procedure. Threads allow concurrency of multiple RPC calls to the server.

An example of software implementing a distributed RPC model is Apache Storm.

Real Time Data Processing (Analytics Layer)

Storm implements stateless sub millisecond latency processing of unbounded tuples using topologies or directed acyclic and supports operations such as filter, join, aggregation, and transformation.

Storm also implements a higher level abstraction called **Trident** which, similarly to Spark, processes data streams in micro batches.

So, looking at the latency continuum, from sub millisecond to second, Storm is a good candidate.

For seconds to minutes scale, Spark Streaming and Storm Trident are excellent its.

For several minutes onward, Spark and a NoSQL database such as Cassandra or HBase are adequate solutions.

For ranges beyond the hour and with high volume of data, Batch processing is the ideal contender.

Real Time Data Processing (Analytics Layer)

Although throughput is correlated to latency, it is not a simple inversely linear relationship.

If processing a message takes 2 ms, which determines the latency, then one would assume the throughput is limited to 500 messages per sec.

Batching messages allows for higher throughput if we allow our messages to be buffered for 8 ms more. With a latency of 10 ms, the system can buffer up to 10,000 messages.

For a bearable increase in latency, we have substantially increased throughput. This is the magic of micro-batching that Spark Streaming exploits.

Ensuring Reliability and Scalability at Integration Layer

- Ingesting data is the process of acquiring data from various sources and storing it for processing immediately or at a later stage.
- Data consuming systems are dispersed and can be physically and architecturally far from the sources.
- Data ingestion is often implemented manually with scripts and rudimentary automation. It actually calls for higher level frameworks like Flume and Kafka.
- The challenges of data ingestion arise from the fact that the sources are physically spread out and are transient which makes the integration brittle.

Ensuring Reliability and Scalability at Integration Layer

- Data production is continuous for weather, traffic, social media, network activity, shop floor sensors, security, and surveillance.
- Ever increasing data volumes and rates coupled with ever changing data structure and semantics makes data ingestion ad hoc and error prone.
- The aim is to become more agile, reliable, and scalable. Agility, reliability, and scalability of the data ingestion determine the overall health of the pipeline.
- Agility means integrating new sources as they arise and incorporating changes to existing sources as needed.

Ensuring Reliability and Scalability at Integration Layer

- In order to ensure safety and reliability, we need to protect the infrastructure against data loss and downstream applications from silent data corruption at ingress.
- Scalability avoids ingest bottlenecks while keeping cost tractable.
- In order to enable an event-driven business that is able to ingest multiple streams of data, process it in light-speed, and make sense of it all to get to rapid decisions, the key driver is the Unified Log.

Ensuring Reliability and Scalability at Integration Layer

A Unified Log is a centralized enterprise structured log available for real-time subscription. All the organization's data is put in a central log for subscription. Records are numbered beginning with zero in the order that they are written. It is also known as a commit log or journal. The concept of the Unified Log is the central tenet of the Kappa architecture.

The properties of the Unified Log are as follows:

- Unified: There is a single deployment for the entire organization
- Append only: Events are immutable and are appended
- Ordered: Each event has a unique offset within a shard
- Distributed: For fault tolerance purpose, the Unified Log is distributed redundantly on a cluster of computers
- Fast: The systems ingests thousands of messages per second

Apache Kafka

- In order to isolate downstream particular consumption of data from the complexities of upstream emission of data, we need to decouple the providers of data from the receivers or consumers of data.
- As they are living in two different worlds with different cycles and constraints, Kafka decouples the data pipelines.
- Apache Kafka is a distributed publish subscribe messaging system rethought as a distributed commit log. The messages are stored by **topic**.

Kafka Features

Apache Kafka has the following properties. It supports:

- High throughput for high volume of events feeds
- Real-time processing of new and derived feeds
- Large data backlogs and persistence for offline consumption
- Low latency as enterprise wide messaging system
- Fault tolerance thanks to its distributed nature

Terminologies

Messages are stored in partition with a unique sequential ID called offset. All Kafka messages are organized into **topics**. Topics are split in replicated **partitions**.

Consumers track their pointers via tuple of (**offset**, **partition**, **topic**).

Kafka has essentially three components: *producers*, *consumers* and *brokers*.

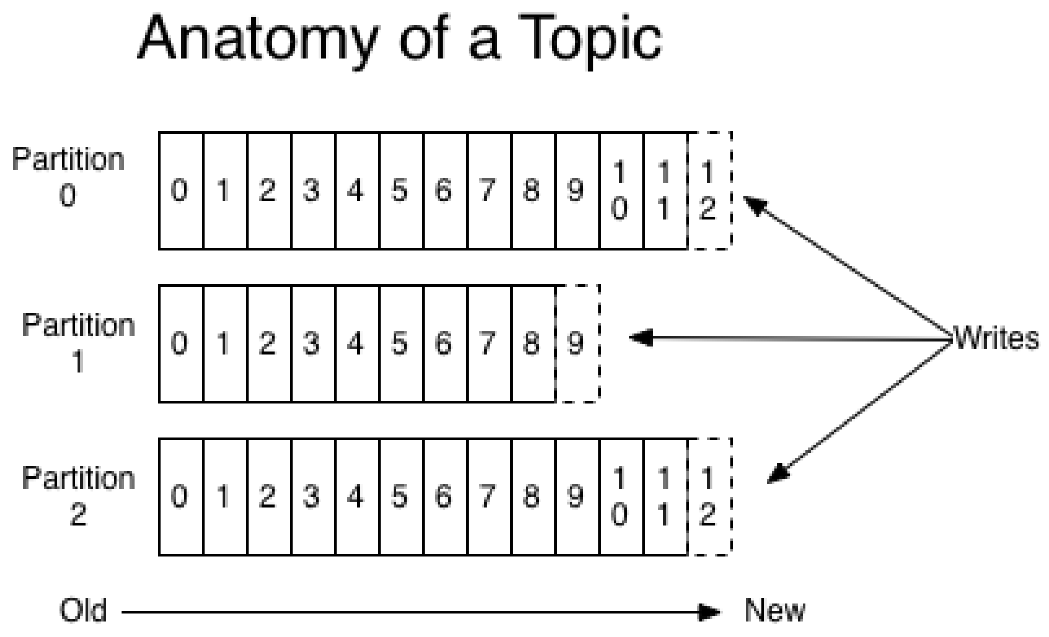
Producers push and write data to brokers.

Consumers pull and read data from brokers.

The **brokers** manage and store the data in topics. Brokers do not push messages to consumers. Consumers pull message from brokers.

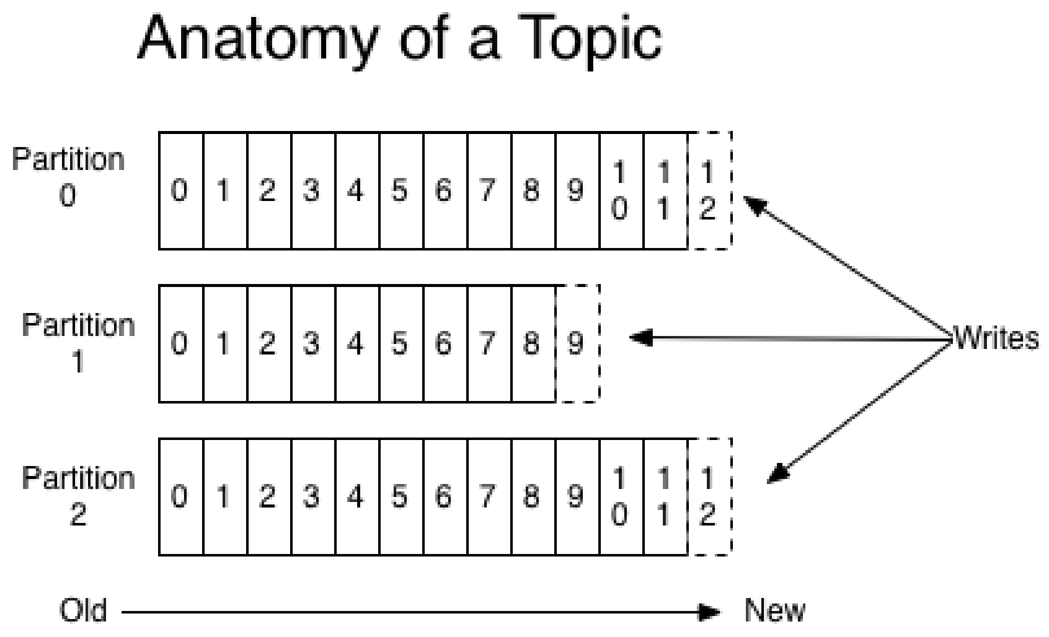
The setup is distributed and coordinated by **Apache Zookeeper**.

The data is persisted in the broker, but not removed upon consumption, but until **retention period**. If a consumer fails, it can always go back to the broker to fetch the data.



Kafka Topic

- Kafka topics are divided into a number of *partitions*.
- Partitions allow you to parallelize a topic by splitting the data in a particular topic across multiple brokers — each partition can be placed on a separate machine to allow for multiple consumers to read from a topic in parallel.
- Consumers can also be parallelized so that multiple consumers can read from multiple partitions in a topic allowing for very high message processing throughput.
- Each message within a partition has an identifier called its *offset*.

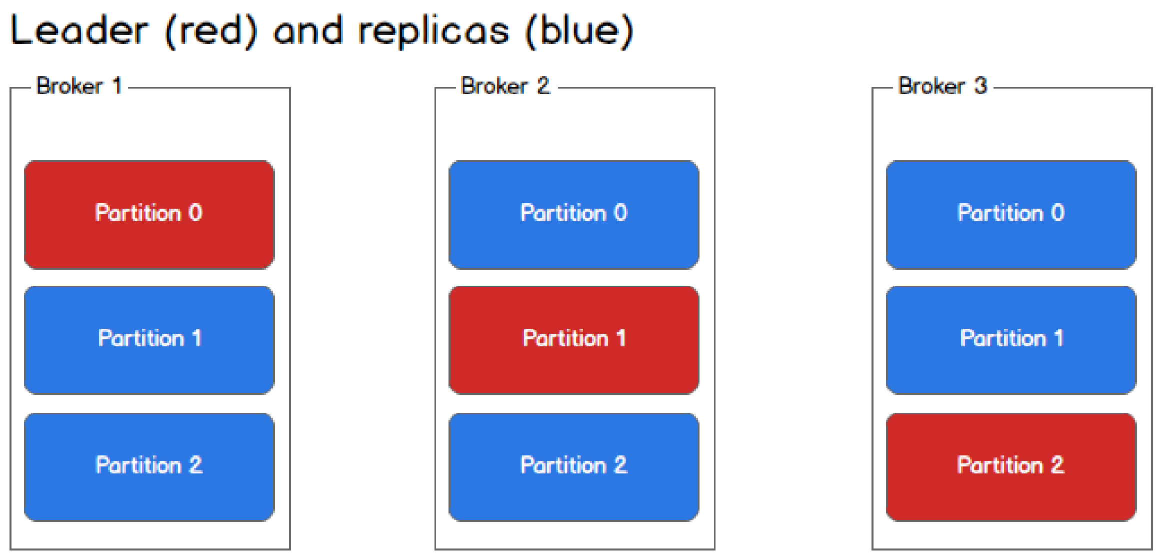


Kafka Topic

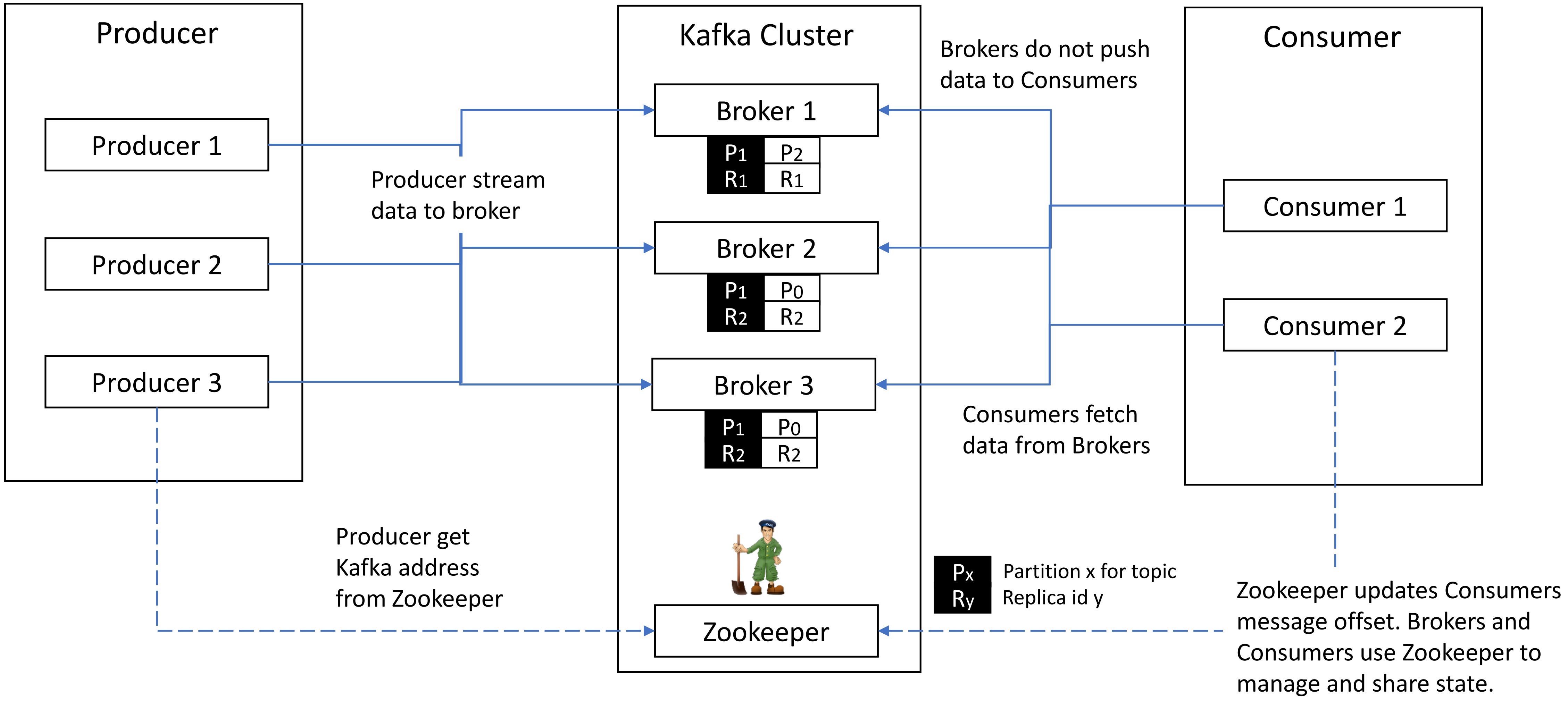
- The offset the ordering of messages as an immutable sequence. Kafka maintains this message ordering for you.
- Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose, allowing consumers to join the cluster at any point in time they see fit.
- Given these constraints, each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the message’s topic, partition, and offset within the partition.

Kafka Topic

Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic. All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data. If a leader fails, a replica takes over as the new leader.



Apache Kafka Architecture


Producer
Producer 1
Producer 2
Producer 3
Producer stream data to broker
Producer get Kafka address from Zookeeper
Kafka Cluster
Broker 1
P1 R1 P2 R1
Broker 2
P1 R2 P0 R2
Broker 3
P1 R2 P0 R2
Zookeeper
Brokers do not push data to Consumers
Consumers fetch data from Brokers
Px Ry Partition x for topic Replica id y
Consumer
Consumer 1
Consumer 2
Zookeeper updates Consumers message offset. Brokers and Consumers use Zookeeper to manage and share state.

Apache Kafka

- Kafka requires Apache ZooKeeper.
- ZooKeeper is a high-performance coordination service for distributed applications.
- It centrally manages configuration, registry or naming service, group membership, lock, and synchronization for coordination between servers.
- It provides a hierarchical namespace with metadata, monitoring statistics, and state of the cluster.
- ZooKeeper can introduce brokers and consumers on the fly and then rebalances the cluster.

Apache Kafka

- Kafka producers do not need ZooKeeper. Kafka brokers use ZooKeeper to provide general state information as well elect leader in case of failure.
- Kafka consumers use ZooKeeper to track message offset. Newer versions of Kafka will save the consumers to go through ZooKeeper and can retrieve the Kafka special topics information.
- Kafka provides automatic load balancing for producers.

Lambda and Kappa Architecture

Two architecture paradigms are currently in vogue: the Lambda and Kappa architectures

- Lambda is the brainchild of the Storm creator and main committer, Nathan Marz. It essentially advocates building a functional architecture on all data. The architecture has two branches.
- Kappa is the brainchild of one the main committer of Kafka, Jay Kreps, and his colleagues at Confluent (previously at LinkedIn). It is advocating a full streaming pipeline, effectively implementing, at the enterprise level, the unified log enounced in the previous pages.

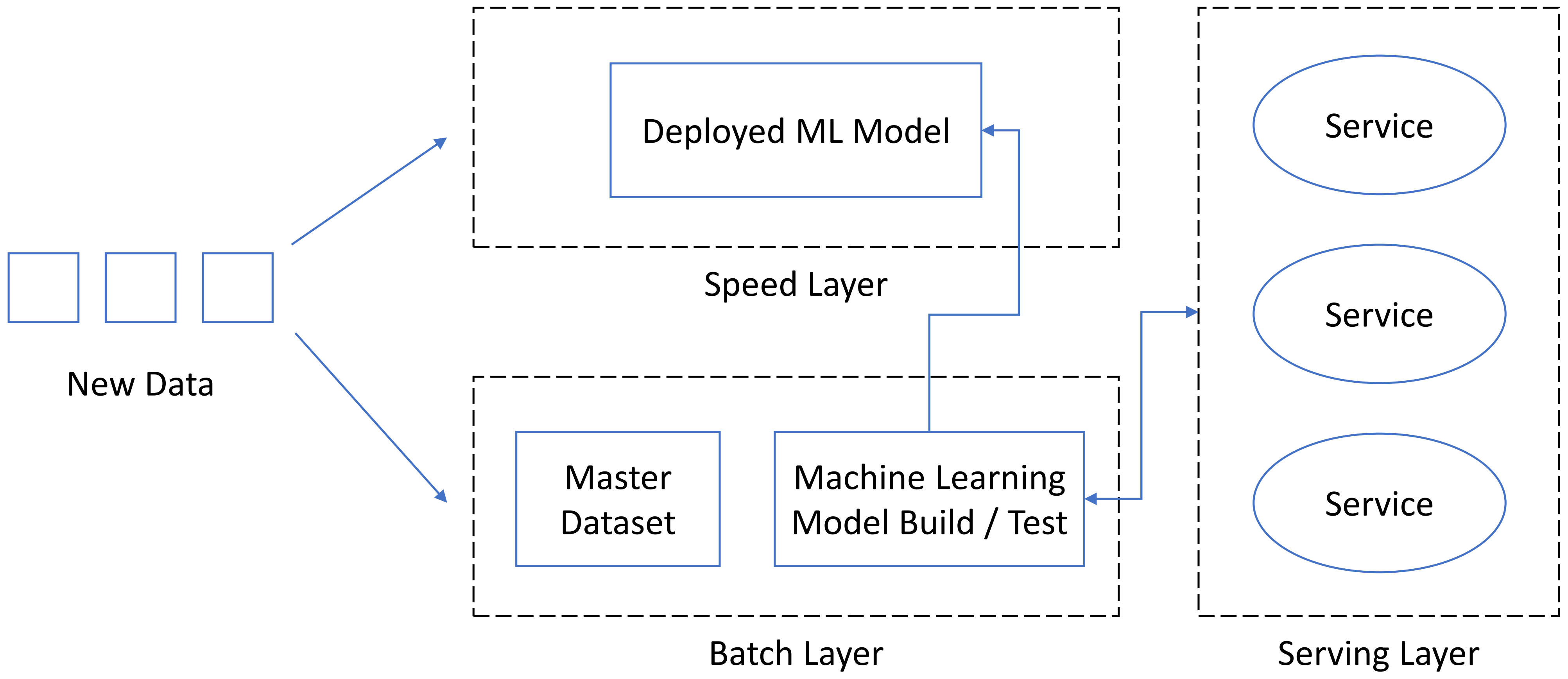
Lambda

Lambda architecture combines batch and streaming data to provide a unified query mechanism on all available data.

Lambda architecture envisions three layers:

- Batch layer where precomputed information are stored
- Speed layer where real-time incremental information is processed as data streams
- Serving layer that merges batch and real-time views for ad hoc queries.

Lambda Architecture



Lambda

- Batch layer datasets can be in a distributed filesystem, while Spark can be used to create batch views that can be fed to the serving layer.
- The serving layer can be implemented using NoSQL technologies such as HBase, Druid, etc.
- Querying can be implemented by technologies such as Apache Drill or Spark SQL
- Speed layer can be realized with data streaming technologies such as Apache Storm or Spark Streaming

Pros of Lambda Architecture

- Retain the input data unchanged. Think about modeling data transformations, series of data states from the original input.
- Lambda architecture take in account the problem of reprocessing data. this happens all the time, the code will change, and you will need to reprocess all the information. Lots of reasons and you will need to live with this.

Cons of Lambda Architecture

- Maintain the code that need to produce the same result from two complex distributed system is painful. Very different code for MapReduce and Storm/ Apache Spark.
- Not only is about different code, is also about debugging and interaction with other products like (hive, Oozie, Cascading, etc)
- At the end is a problem about different and diverging programming paradigms.

Kappa

- The Kappa architecture proposes to drive the full enterprise in streaming mode.
- The Kappa architecture arose from a critique from Jay Kreps and his colleagues at LinkedIn at the time. Since then, they moved and created Confluent with Apache Kafka as the main enabler of the Kappa architecture vision.
- The basic tenet is to move in all streaming mode with a Unified Log as the main backbone of the enterprise information architecture.

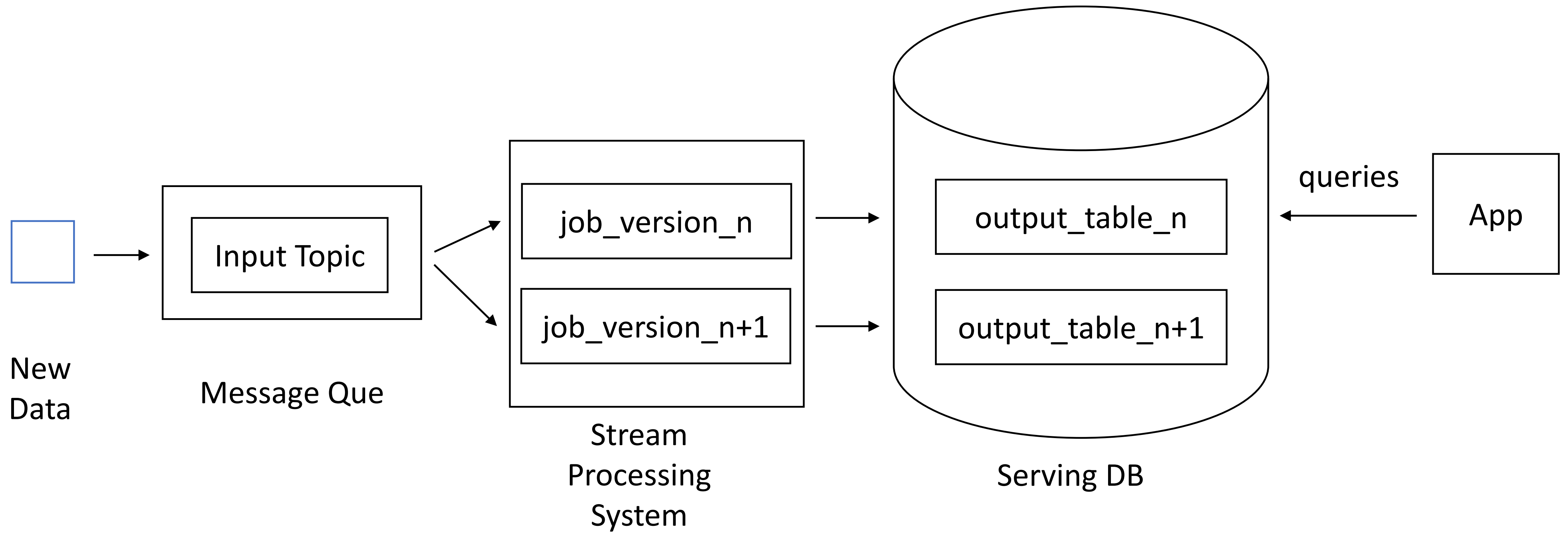
Kappa

- The proposal: Use Kafka (or other system) that will let you retain the full log of the data you need to reprocess.
- When you want to do the reprocessing, start a second instance of your stream processing job that starts processing from the beginning of the retained data, but direct this output data to a new output table.
- When the second job has caught up, switch the application to read from the new table. Stop the old version of the job, and delete the old output table

Kappa

- A Unified Log is a centralized enterprise structured log available for real-time subscription.
- All the organization's data is put in a central log for subscription.
- The concept of the Unified Log is the central tenet of the Kappa architecture.
- Ref: Properties of Unified Log discussed previously in the section “Ensuring Reliability and Scalability at Integration Layer”.

Kappa Architecture



Pros of Kappa Architecture

- The first benefit is that only you need to reprocessing only when you change the code.
- You can check if the new version is working ok and if not reverse to the old output table.
- You can mirror a Kafka topic to HDFS so you are not limited to the Kafka retention configuration.
- You have only a code to maintain with an unique framework.

Cons of Kappa Architecture

Creating an unified centralized log takes a huge amount of effort!

Implementing data governance on Centralized log is more difficult.

Batch layer is often very useful for EDA (exploratory data analysis). EDA on top of unified log adds additional codes and complexity.

Next Week

- DStream and Structured Streaming Lab
- Project 2: Real Time DIA: Bitcoin Trading Algorithm
- Introduction of our Bitcoin Trading Platform (Simulation)
- Project Submission Details: 1 Week For Homework, 1 Week to run algorithm to see who made the most amount of fake money!
- Disclaimers so I am not hold accountable in future for your investment plans.

Thank you and see you next week!