1. **Lambda Architecture**

This Architecture is widely used in many big tech companies as it takes advantage of both real-time data processing as well as batch processing i.e. one can query both fresh data by real-time data processing technique and historical data using batch processing data technique.

What is Lambda Architecture?

Lambda architecture is an excellent architecture for handling massive real-time data and building fault-tolerant, scalable systems.

Lambda (λ) architecture is one of 3 big data architecture patterns. Apart from batch and stream processing, Lambda architecture also includes a data serving layer for responding to user queries.

Why use Lambda Architecture?

Lambda Architecture addresses several key challenges in big data processing:

1) Handling both historical and real time data: Lambda Architecture allows to process and analyze large volumes of historical data while simultaneously dealing with incoming real time data streams. It achieves this by separating the processing into distinct layers—batch processing for historical data and stream processing for real time data.

2) Balancing latency and accuracy: Lambda Architecture strikes a balance between the high accuracy of batch processing (which handles all data with perfect accuracy) and the low latency of stream processing (which provides near-instant insights with potentially less accuracy).

3) Fault tolerance: Lambda Architecture is designed to be resilient to hardware failures and human errors, guaranteeing data integrity and system reliability.

4) Scalability: Lambda Architecture can handle growing data volumes and increasing processing demands by leveraging distributed computing principles.

Core Key Principles of Lambda Architecture

Lambda Architecture is built on several key principles:

1) Immutable Data: All data entering the system is treated as immutable and append-only. This approach preserves the integrity of the historical record and simplifies data processing by avoiding the complexities associated with data mutations.

2) Recomputation: Lambda Architecture supports the ability to recompute results from raw data. This ensures accuracy and allows for corrections in the data processing pipeline. The architecture maintains consistent and reliable results by recomputing data from the master dataset.

3) Separation of Batch and Speed Layers: Lambda Architecture separates the batch layer (which processes historical data) from the speed layer (which processes real time data). This separation allows for optimized processing in each layer.

4) Serving Layer: The final results from both the batch and speed layers are merged in the serving layer. This layer provides low-latency access to processed data by indexing and storing precomputed views, which can be queried efficiently.

5) Polyglot Persistence: Lambda Architecture supports the use of multiple data storage technologies, optimizing storage solutions based on the specific characteristics of the data (e.g., type, access patterns).

Different Ways to Approach Lambda Architecture

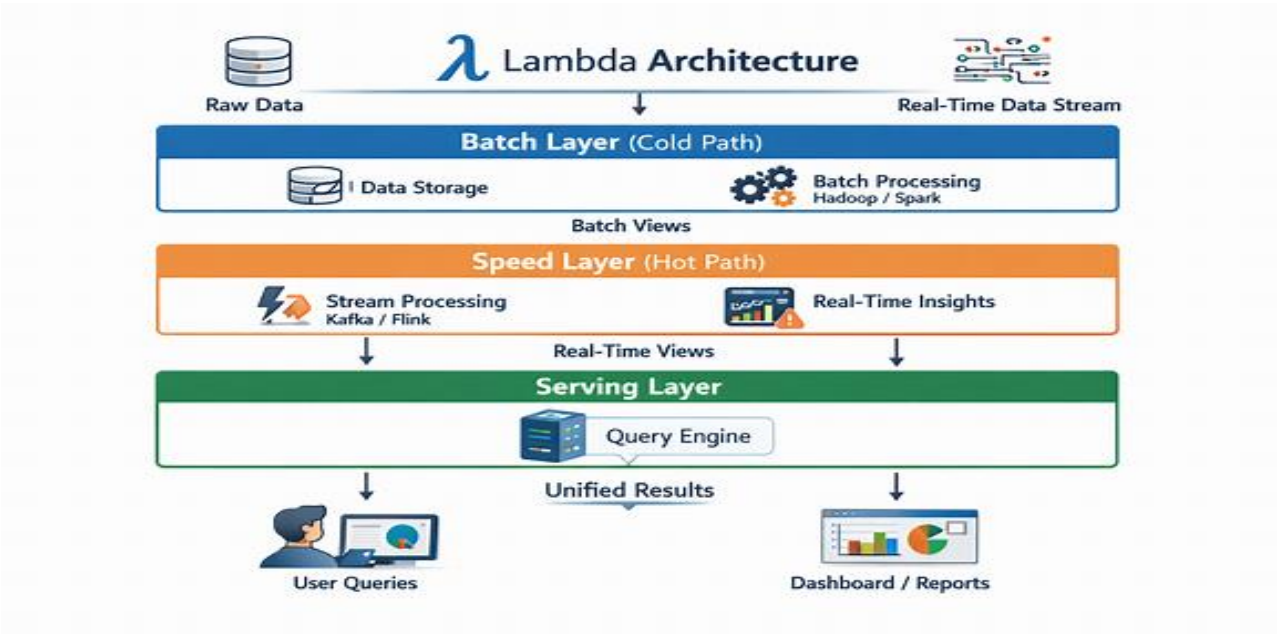
There are two approaches to Lambda Architecture:

- **Hybrid approach:**
    - It is designed to harness enormous volumes of rapidly created data, enabling businesses to make use of data more quickly.
- **Specific approach:**
    - It attempts to balance latency, throughput, and fault tolerance by using batch processing to provide accurate views by batch data, while simultaneously using real-time stream processing to provide views of online data. The outputs from both batch and speed layers can be merged before the presentation.

Layers in Lambda Architecture

Lambda Architecture has mainly three layers to process big data:

- Batch Layer (Cold process)
- Stream Layer (Hot process or Speed Layer)
- Serving layer



1. Batch layer

Batch Layer operates on the complete data and thus allows the system to produce the most accurate results. However, the results come at the cost of high latency due to high computation time.
The batch layer stores the raw data as it arrives and computes the batch views for consumption. Naturally, batch processes will occur at some interval and will be long-lived. The scope of data is anywhere from minutes to years.

2. Stream layer

Stream Layer operates on the real-time data to complement the batch views. It receives the arriving data from various clients and performs incremental updates to the batch layer results and store them in processed data Database.

This layer generates results in a low-latency, near real-time fashion. By implementing incremental algorithms(like insertion sort) at the Stream layer, the computation cost can be significantly reduced. The batch views may be processed with more complex or expensive rules and takes more time but has better data quality and less skew, while the real-time views processed simply by incoming traffic give you access to the latest possible data.

3. Serving Layer

Serving Layer is a server or a set of servers which processes output of various queries from different modules(like analytics module, Notification module) using the results sent from the batch and speed layers.

The outputs from the batch layer in the form of batch views and the speed layer in the form of near-real-time views are stored in the Processed Data DB as well as sent to serving layer, and this output is used by the serving layer to compute the queries on an ad-hoc basis and the database is used by the serving layer to compute the queries on premeditated basis.

List of tools used in the Lambda Architecture

- Apache Hadoop is used to store data and create distributed clusters.
- Hadoop Distributed File System (HDFS) is used for managing immutable data in the batch layer.
- Apache Spark is used for data streaming, graph processing, and data batch process.
- Apache Cassandra is used to store real-time views.

- Apache Kafka is used for data streaming in the speed layer.
- Apache Storm is used for the speed layer tasks.
- Apache HBASE is used for the serving layer tasks.

Advantages of Lambda Architecture

- It is a good balance of speed, reliability, and scalability.
- The batch layer of Lambda architecture manages historical data with the fault-tolerant, distributed storage, ensuring a low possibility of errors even if the system crashes.
- The Stream layer of Lambda architecture manages the real time data with immediate response with somewhat less precision.
- Access to both real-time and offline data results in covering many data analysis scenarios very well.

Disadvantages of Lambda Architecture

- Lambda architecture is complex infrastructure as it has many layers involved.
- Although the offline layer and the real-time stream face different scenarios, their internal processing logic is the same, so there are many duplicate modules and require different codebase.
- Maintaining the different code base and keeping them in sync so that processed data produces same results from both paths.
- Computes every batch cycle more then once, which decreases the system performance and requires more resources.
- A data set modeled with Lambda architecture is difficult to migrate or reorganize.

Real-world applications

- **Social media analytics:** Lambda architecture in social media analytics allows you to perform sentiment analysis. Sentiment analysis allows businesses to understand how people

talk about them on social media to gauge public opinions and overall feelings toward the company.

- **Fraud detection:** Lambda architecture can help with real-time fraud detection. It can directly compare historical data with real-time data to identify concerning activity.
- **Marketing:** Personalized marketing approaches allow you to make specific recommendations for different customers. By analyzing historical and real-time data with lambda architecture, you can take a data-driven approach to building better relationships with consumers and offering products they're more likely to be interested in.

Benefits of lambda architectures

Here are the main benefits of lambda architectures:

No Server Management – you do not have to install, maintain, or administer any software.

Flexible Scaling – your application can be either automatically scaled or scaled by the adjustment of its capacity

Automated High Availability – refers to the fact that serverless applications have already built-in availability and faults tolerance. It represents a guarantee that all requests will get a response about whether they were successful or not.

Business Agility – React in real-time to changing business/market scenarios

Challenges with lambda architectures

Complexity – lambda architectures can be highly complex. Administrators must typically maintain two separate code bases for batch and streaming layers, which can make debugging difficult.

2. Kappa Architecture

What is Kappa Architecture?

Kappa Architecture is a data processing architecture designed for handling real-time data streams. It simplifies the data pipeline by processing all data as streams, eliminating the need for a separate batch processing layer, which is central to Lambda Architecture.

In Kappa Architecture:

- Data is ingested as streams from various sources.
- Stream processing engines (like Apache Kafka, Apache Flink, or Apache Samza) handle continuous real-time data transformations, aggregations, and computations.
- The architecture treats both new and historical data uniformly, replaying events when needed to reprocess data or handle errors.

This model provides scalability, simplicity, and low-latency data processing, making it ideal for applications where real-time insights are crucial.

Core Components of Kappa Architecture

While implementations vary, most Kappa-style systems share three core building blocks:

Immutable Log (Event Store)

The source of truth: a distributed, append-only log such as Apache Kafka, Apache Pulsar, or Redpanda. All events are written here and retained long-term, sometimes indefinitely. Events are immutable; changes are represented by new events, not in-place updates.

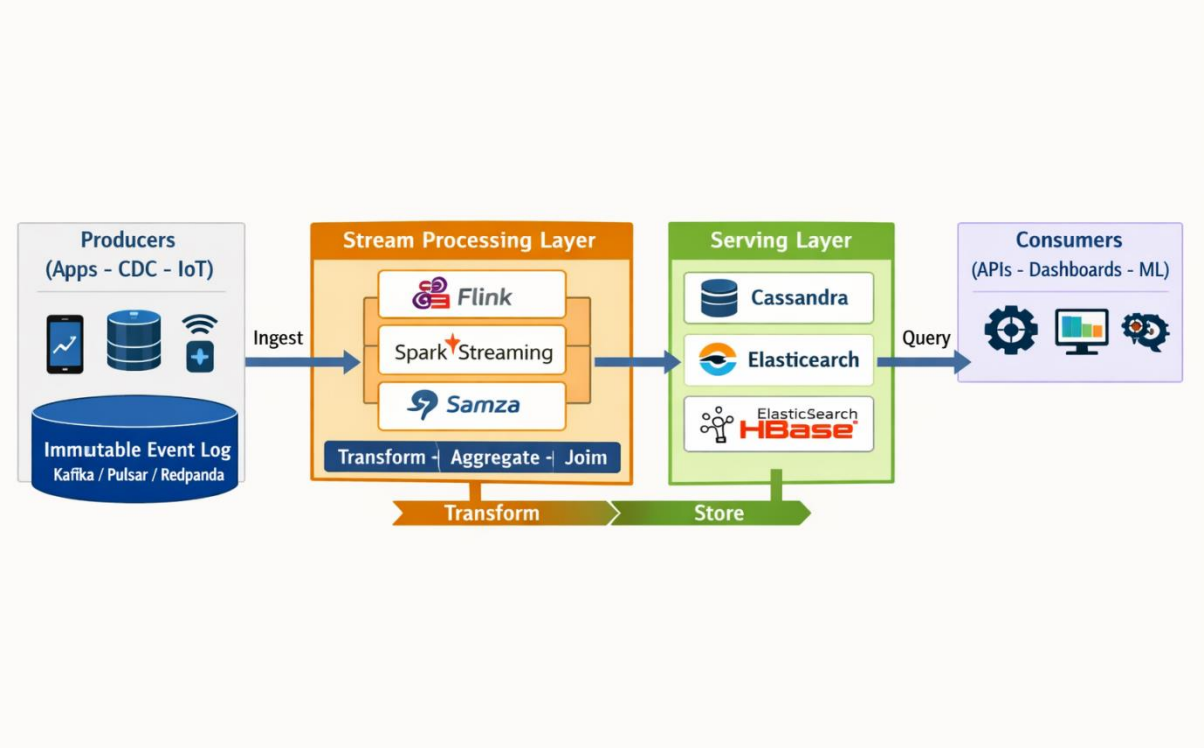
Stream Processing Layer

A stream processing engine (e.g., Apache Flink, Kafka Streams, Apache Samza, Spark Structured Streaming) consumes events from the log, applies transformations, aggregations, joins, and enrichments, and emits derived streams. This layer encodes your business logic and is responsible for both real-time and historical computations.

Serving Layer (Materialized Views)

Outputs from stream processing land in one or more serving stores optimized for queries: relational databases, OLAP systems, search indices, key-value stores, or NoSQL databases. These stores are not canonical; they can be dropped and rebuilt by replaying from the log whenever needed.

Around these, there are usually producers (applications, services, sensors, CDC tools) writing events into the log, and consumers (dashboards, APIs, ML models) reading from the serving layer.



Use Cases for Kappa Architecture

Kappa Architecture is well-suited for real-time data processing applications. Here are some common use cases:

- **Real-time Analytics:** Businesses that need to analyze live data streams, such as website clickstreams, sensor data, or financial transactions, can use Kappa Architecture for instant insights and decision-making.

- **Fraud Detection:** Financial institutions and online platforms can implement Kappa Architecture to detect fraudulent activities in real-time by continuously monitoring transactions or user behaviors.

- **IoT Data Processing:** In IoT ecosystems, devices generate massive streams of data that require real-time processing and monitoring, such as in smart cities, connected vehicles, and industrial sensors.

- **Log and Event Processing:** Systems that need to monitor and analyze application logs, user activity events, or server metrics can benefit from Kappa Architecture's ability to provide immediate insights and trigger actions.

- **Personalization and Recommendation Engines:** Online platforms like e-commerce sites and streaming services can use real-time data streams to deliver personalized recommendations to users based on their current activity.

- **Monitoring and Alerting:** Organizations can monitor system performance, security threats, or network activity in real-time, enabling faster detection and response to issues.

These use cases leverage Kappa Architecture's ability to handle high-volume, low-latency, continuous data streams without the complexity of managing separate batch and real-time processing layers.

Kappa Architecture Design Principles

Kappa Architecture is built around a set of design principles that focus on simplicity, scalability, and real-time data processing. Here are the key principles:

- **Stream Processing as the Core:** All data is treated as a continuous stream, whether it's real-time or historical. This eliminates the need for separate batch and real-time layers, as in Lambda Architecture, simplifying the overall system.

- **Event Replayability:** Kappa Architecture relies on the ability to replay past events for reprocessing. This allows systems to handle changes, bug fixes, or updates by simply reprocessing the event stream without the need for a separate batch process.

- **Immutable Logs:** Data streams are treated as immutable logs, where events are appended rather than modified. This ensures data consistency and enables easy reprocessing.

- **Stateless and Stateful Processing:** The architecture supports both stateless and stateful stream processing. Stateless operations can be applied independently to each event, while stateful processing involves maintaining data (e.g., aggregations or windows) across multiple events.

- **Scalability and Fault Tolerance:** The architecture is designed to scale horizontally by distributing the workload across multiple nodes. Fault tolerance is achieved through replication and durable storage, ensuring data is never lost even if processing fails.

- **Low Latency:** With all data processed as a stream, Kappa Architecture ensures minimal delay from data ingestion to output, providing near real-time insights.

- **Unified Codebase:** By processing data through a single stream-processing engine, the system has a unified codebase for real-time and historical data, reducing complexity in development and maintenance.

These principles make Kappa Architecture highly suitable for environments where real-time data processing, scalability, and simplicity are critical.

Benefits of Kappa Architecture

Kappa Architecture offers several benefits that make it a compelling choice for real-time data processing systems:

- **Simplicity:** By removing the need for a batch processing layer, Kappa Architecture simplifies the overall design. All data is processed as a stream, making the system easier to maintain and understand.

- **Low Latency:** Since all data is processed in real-time, the architecture delivers low-latency results, making it ideal for applications requiring immediate insights or actions.

- **Unified Codebase:** The same stream-processing logic is applied to both real-time and historical data, which reduces the complexity of maintaining separate codebases for batch and real-time processing.

- **Scalability:** Kappa Architecture scales easily to handle large data volumes, as stream processing engines like Kafka and Flink can distribute data across multiple nodes and handle horizontal scaling efficiently.

- **Event Replayability:** The ability to replay events allows for easy reprocessing of historical data, fixing errors, or reapplying new logic without the need for complex batch jobs.

- **Fault Tolerance:** With built-in replication and fault tolerance mechanisms in stream processing engines, Kappa Architecture ensures high availability and data durability, even in case of failures.

- **Flexibility:** It supports both stateless and stateful processing, making it suitable for a wide range of applications, from simple filtering to complex aggregations over time windows.

These benefits make Kappa Architecture a strong candidate for systems needing real-time data processing, continuous updates, and scalable performance.

Challenges of Kappa Architecture

While Kappa Architecture offers several advantages, it also comes with some challenges:

- **Complex Stream Processing:** Designing and maintaining stream-processing logic can be complex, especially when dealing with stateful operations or large-scale distributed systems. Developers need to carefully manage windowing, aggregations, and state consistency.

- **Data Ordering and Deduplication:** Ensuring the correct ordering of events in distributed stream systems can be challenging. Similarly, handling duplicate events (which can occur due to retries or network issues) requires additional logic to prevent incorrect data processing.

- **Reprocessing Overhead:** While event replayability is a strength, reprocessing large amounts of historical data can become resource-intensive, especially for high-throughput systems.

- **Limited Support for Batch Processing:** Kappa Architecture is optimized for stream processing, which may not be suitable for applications that require extensive batch processing or for systems where batch jobs are more efficient.

- **Fault Tolerance Complexity:** While Kappa systems are fault-tolerant, ensuring seamless recovery from failures in a distributed streaming environment requires careful management of checkpoints, state storage, and data replication.

- **Consistency and Latency Trade-offs:** Achieving consistency across distributed nodes can introduce additional complexity. Balancing low-latency data processing with consistency guarantees (like exactly-once processing) may require trade-offs depending on the use case.

- **Resource Management:** Real-time stream processing demands continuous computing resources, potentially leading to higher infrastructure costs if the system is not well-optimized.

These challenges must be addressed to successfully implement Kappa Architecture in large-scale, real-time data systems.

Real-World Examples of Kappa Architecture

Several companies and platforms use Kappa Architecture to handle real-time data processing at scale. Here are some notable real-world examples:

- **LinkedIn (Apache Kafka):** LinkedIn developed Apache Kafka, which is central to many Kappa Architecture implementations. Kafka processes billions of events per day, powering real-time features such as activity feeds, messaging, and monitoring systems.

- **Netflix:** Netflix uses Kappa Architecture to process and analyze large volumes of streaming data from user interactions, device metrics, and content delivery in real-time. This helps with personalization, real-time recommendations, and operational monitoring.

- **Uber:** Uber relies on Kappa Architecture to manage real-time data streams from their ride-hailing app. This architecture processes data from drivers, riders, GPS signals, and more to provide real-time insights for pricing, routing, and demand prediction.

- **Spotify:** Spotify uses Kappa Architecture to deliver real-time analytics, including user behavior tracking, music recommendation systems, and real-time ad serving. This architecture enables quick insights into user preferences and content trends.

- **Pinterest:** Pinterest uses Kappa Architecture to track and process real-time user interactions such as pins, repins, and clicks. The system continuously updates analytics dashboards and recommendation engines to reflect the latest user activity.

- **Yelp:** Yelp processes user-generated reviews, photos, and check-ins using real-time data streams. This enables real-time content moderation, location-based services, and personalized recommendations based on live data from its users.

These companies leverage the simplicity, scalability, and low-latency capabilities of Kappa Architecture to power their real-time analytics, personalized recommendations, and event-driven systems.

Kappa Architecture Vs Lambda Architecture

Feature	**Kappa**	**Lambda**
Data Processing	Stream-only	Batch + Stream
Complexity	Lower (single pipeline)	Higher (dual pipelines)
Latency	Very low	Balanced (batch accuracy + stream speed)
Reprocessing	Event replay	Batch recomputation
Use Cases	IoT, fraud detection, real-time analytics	Hybrid systems needing both historical accuracy & real-time