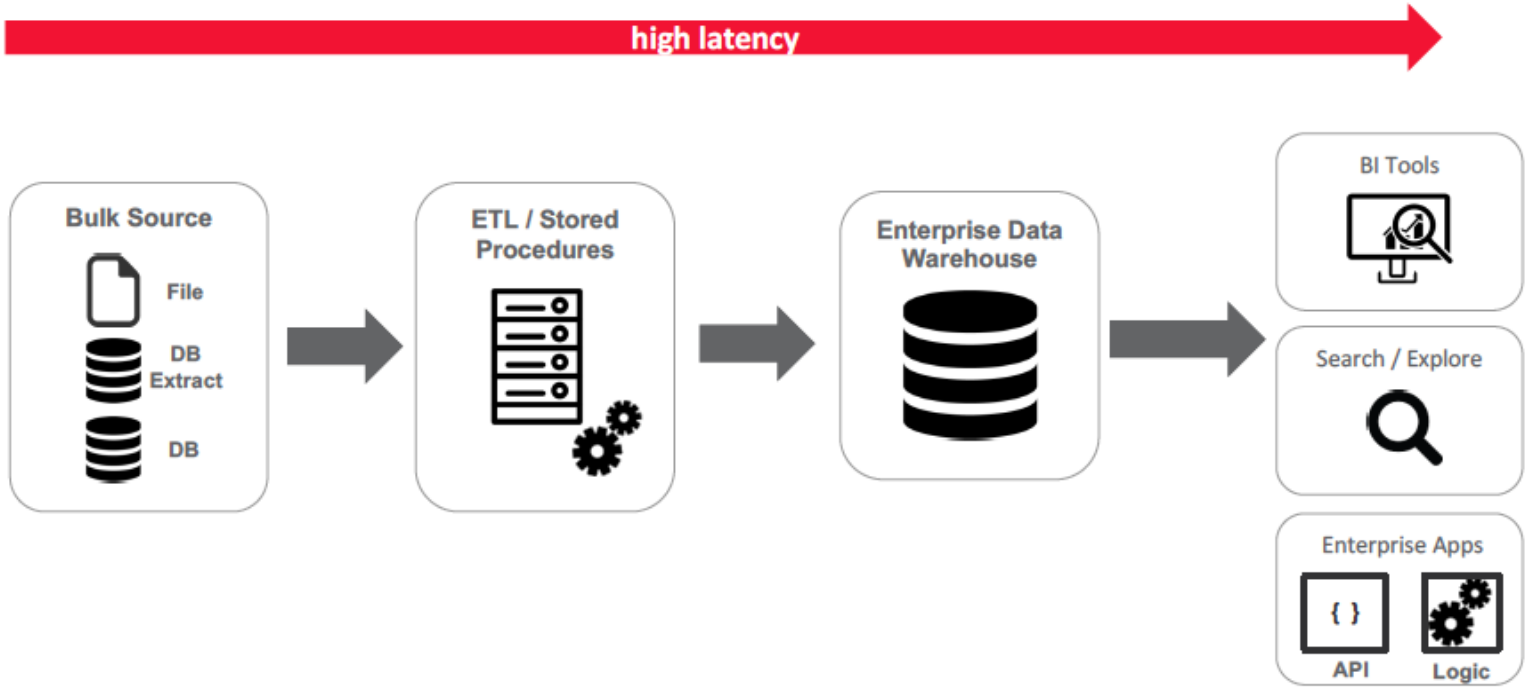
Chương 8
Kiến trúc dữ liệu lớn

Traditional BI infrastructures



Batch and Stream Processing

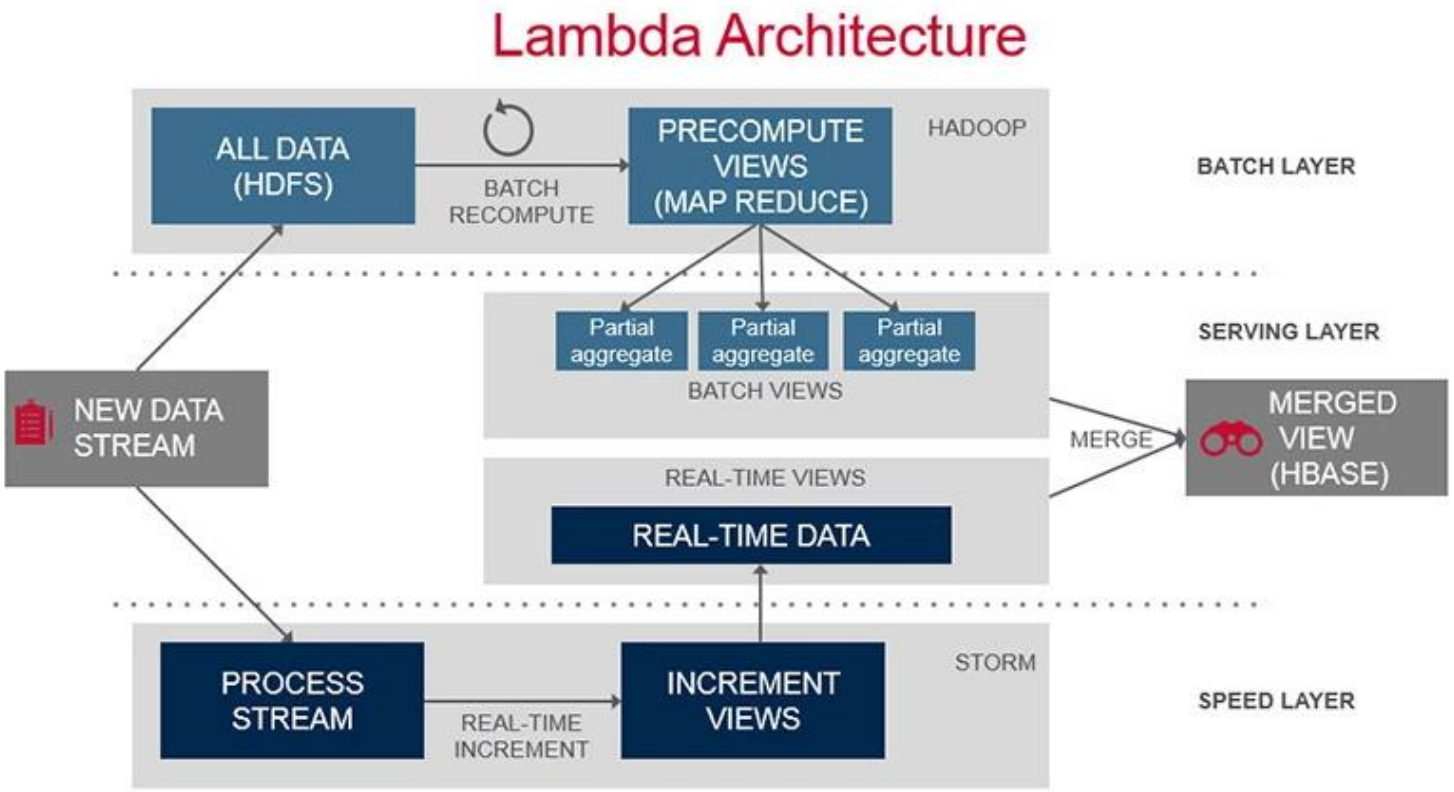
- **Batch Processing**
  - Involves collecting data over a period and processing it in large, fixed-sized batches.
  - Key Characteristics
    - Processes finite, static datasets
    - High throughput
    - Higher latency
    - Suitable for complex analytics and large-scale data transformations
- **Stream Processing**
  - Involves processing data in real-time as it arrives, typically one record at a time or in micro-batches.
  - Key Characteristics
    - Processes unbounded, continuous data streams
    - Low latency
    - Real-time or near-real-time results
    - Suitable for immediate insights and quick reactions

Lambda Architecture

- Lambda Architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods.

I need fast access to historical data on the fly for predictive modeling with real time data from the stream

Lambda Architecture Layers



Batch Layer

- Manages the master dataset and pre-computes batch views
- Key Components
  - Distributed file system (e.g., HDFS)
  - Batch processing system (e.g., Apache Hadoop, Apache Spark)
- Functions
  - Data storage
  - Batch processing
  - Creation of batch views

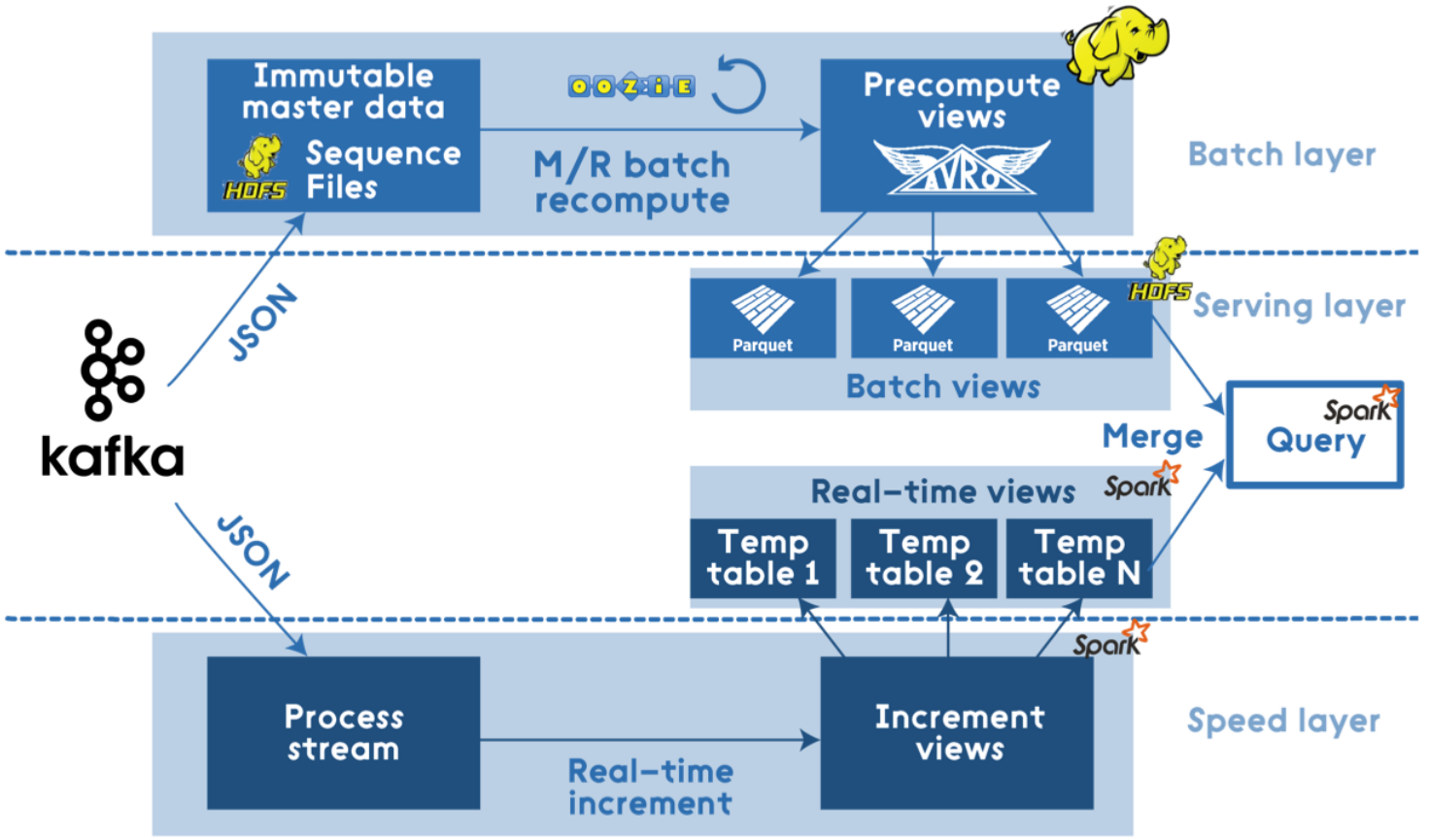
Speed Layer

- Compensates for the processing lag of the batch layer
- Key Components
  - Stream processing systems (e.g., Apache Storm, Apache Flink)
- Functions
  - Real-time data processing
  - Incremental updates of real-time views

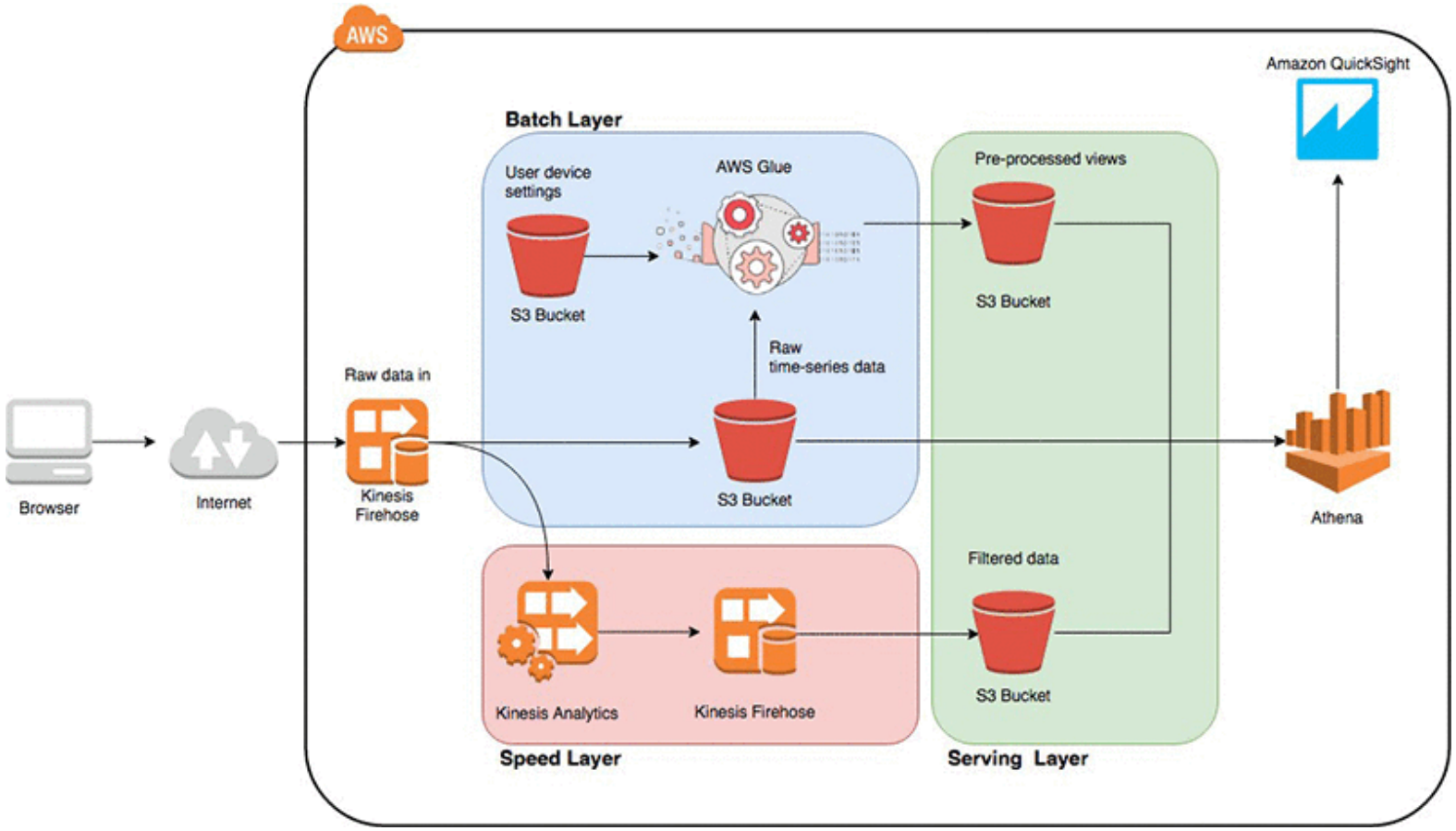
Serving Layer

- Responds to queries by merging batch and real-time views
- Key Components
  - Low-latency, read-optimized databases (e.g., Apache HBase, Cassandra)
- Functions
  - Indexing batch views
  - Facilitating fast reads
  - Merging of batch and real-time views

Lambda architecture implementation



Lambda architecture in AWS



Addressing Fault-tolerance

- Lambda Architecture enhances fault-tolerance through
  - Immutable Data Storage
    - Raw data is stored unchanged in the Batch Layer
    - Ensures data integrity even if processing fails
  - Distributed Processing
    - Both Batch and Speed Layers use distributed systems (e.g., Hadoop, Spark)
    - If one node fails, work can be redistributed
  - Redundancy
    - Data is processed in both Batch and Speed Layers
    - Provides a backup if one layer fails

Mitigating Human Errors in Coding

- Lambda Architecture reduces the impact of coding errors by
  - Reprocessing Capability
    - If a bug is found, fix the code and rerun the entire dataset
    - No data loss due to immutable data storage
  - Separation of Concerns
    - Batch and Speed Layers are separate
    - Errors in one layer don't necessarily affect the other
  - Easier Testing
    - Batch Layer allows thorough testing of complex computations
    - Helps catch errors before they impact real-time processing

Facilitating Data Reprocessing

- Lambda Architecture supports efficient data reprocessing through
  - Immutable Data
    - Original data is always available for reprocessing
    - No risk of data corruption during updates
  - Batch Layer Design
    - Optimized for processing entire datasets efficiently
    - Allows for easy recomputation when needed
  - Decoupled Processing
    - While Batch Layer reprocesses historical data...
    - Speed Layer continues to handle new data
    - System remains operational during reprocessing

Eventual Consistency Model in Lambda Architecture

- System will become consistent over time, given that the system stops receiving new updates
  - Batch Layer: Provides a strongly consistent but outdated view
  - Speed Layer: Provides a possibly inconsistent but up-to-date view
  - Serving Layer: Merges both views, achieving eventual consistency

Challenges and Limitations

- Complexity of Maintaining Dual Systems
  - Managing two separate processing systems (batch and speed)
  - Coordinating updates and ensuring synchronization
  - Increased monitoring and troubleshooting complexity
- Code Redundancy Between Batch and Speed Layers
  - Same computations often implemented twice
  - Increases risk of inconsistencies and bugs
- Potential Data Inconsistencies
  - Lag between batch and real-time processing results
  - Challenges in correctly merging batch and real-time views
- Resource Intensive
  - Need for substantial computational resources to run both layers

Kappa architecture

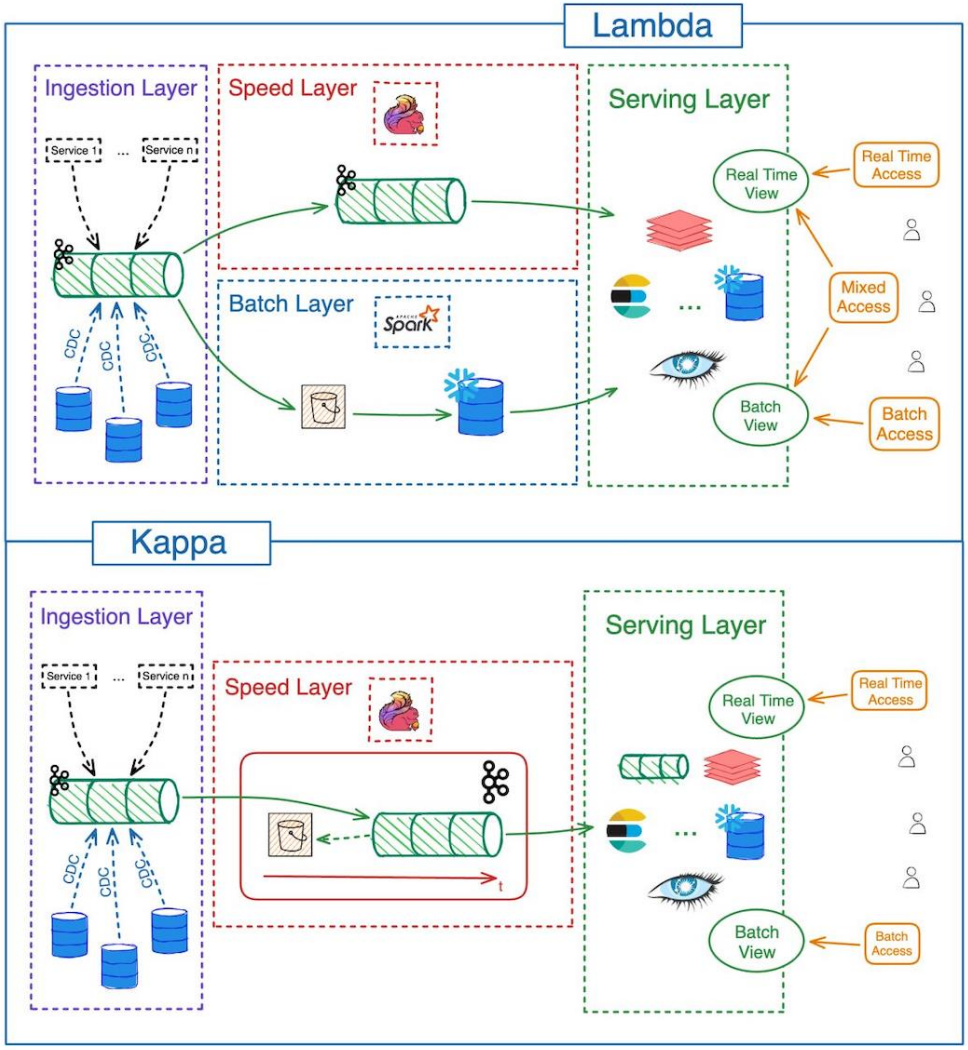
Introduction

- Kappa Architecture is a software architecture pattern for real-time data processing that uses a single, unified system for both stream and batch processing.
- Motivation
  - Need for real-time data processing in modern applications
  - Challenges with maintaining separate batch and stream processing systems
  - Desire for simpler, more maintainable big data architectures

Kappa as a simplification

- Limitations of Lambda Architecture
  - Complexity of maintaining two separate systems
  - Potential for data inconsistencies between batch and speed layers
  - Higher development and operational costs
- Kappa as a simplification
  - Single processing path for both real-time and historical data
  - Eliminates need for separate batch processing system
  - Reduces complexity and potential for errors

Kappa architecture



Core Principles of Kappa Architecture

- Data as an Event Stream
  - All data is treated as a series of immutable events
  - Events are ordered by time of occurrence
  - Append-only log of events (e.g., Kafka topics)
- Reprocessing Capability
  - Ability to reprocess entire event stream from the beginning
  - Enables bug fixes, algorithm updates, and historical analysis
- Single Pipeline
  - One processing path for both real-time and batch-like workloads
  - Unified code base for all data processing tasks

Key Components

- Event Streaming Platform
  - Purpose: Durable storage of all input events
  - Example: Apache Kafka
  - Key features: High throughput, fault-tolerance, scalability
- Stream Processing System
  - Purpose: Real-time processing of event streams
  - Examples: Apache Flink, Kafka Streams, Apache Spark Streaming
  - Key features: Low-latency, stateful processing, exactly-once semantics
- Serving Layer
  - Purpose: Store and serve processed results
  - Examples: Apache Cassandra, Elasticsearch, Apache Druid
  - Key features: Fast reads, support for real-time updates

Data Flow in Kappa Architecture

- Data Ingestion
  - Events captured from various sources (e.g., user interactions, IoT devices)
  - Events published to the event streaming platform (e.g., Kafka topics)
- Stream Processing
  - Continuous consumption of events from the streaming platform
  - Application of business logic, transformations, and aggregations
  - Stateful processing using local state stores
- State Updates
  - Processed results written to the serving layer
  - Incremental updates to maintain real-time views
- Query Serving
  - Applications read processed data from the serving layer
  - Support for both real-time and historical queries

Data Reprocessing in Kappa Architecture

- Scenarios requiring reprocessing
  - Bug fixes in processing logic
  - Algorithm improvements
  - Addition of new features or data fields
  - Historical analysis with updated logic
- Reprocessing mechanism
  - Deploy updated stream processing application
  - Reset consumer offsets to beginning of event log
  - Reprocess entire event stream with new logic
  - Overwrite or merge results in serving layer
- Comparison with Lambda Architecture
  - Kappa: Single codebase to maintain, simpler reprocessing
  - Lambda: Separate batch system for reprocessing, potential inconsistencies

Advantages of Kappa Architecture

- Simplified System Architecture
  - Single processing pipeline reduces complexity
  - Unified codebase for all data processing tasks
  - Easier to reason about and maintain
- Reduced Latency
  - Real-time processing of all data
  - No delay waiting for batch processing jobs
- Improved Consistency
  - Single source of truth (event log)
  - No discrepancies between batch and real-time views
- Scalability and Flexibility
  - Event streaming platforms designed for high scalability
  - Easy to add new consumers or modify existing processing logic
- Cost-Effective
  - Reduced infrastructure needs compared to Lambda Architecture
  - Simplified operations and maintenance

Challenges and Limitations

- Long-term Storage Requirements
  - Need to retain complete event history for reprocessing
  - Can lead to significant storage costs for high-volume systems
- Complexity in Stream Processing Design
  - Handling out-of-order events and late data
  - Ensuring exactly-once processing semantics
  - Managing state in distributed stream processors
- Limitations in Batch-like Processing
  - Some operations more challenging in streaming paradigm (e.g., joins across large datasets)
  - Potential performance issues for large-scale reprocessing
- Learning Curve
  - Shift in thinking from traditional batch processing
  - Requires familiarity with stream processing concepts and tools
- Use Case Applicability
  - May not be optimal for all scenarios, especially those with clear separation between historical and real-time data needs

HUST

hust.edu.vn fb.com/dhbkhn

Thank you for your attention!

Q&A