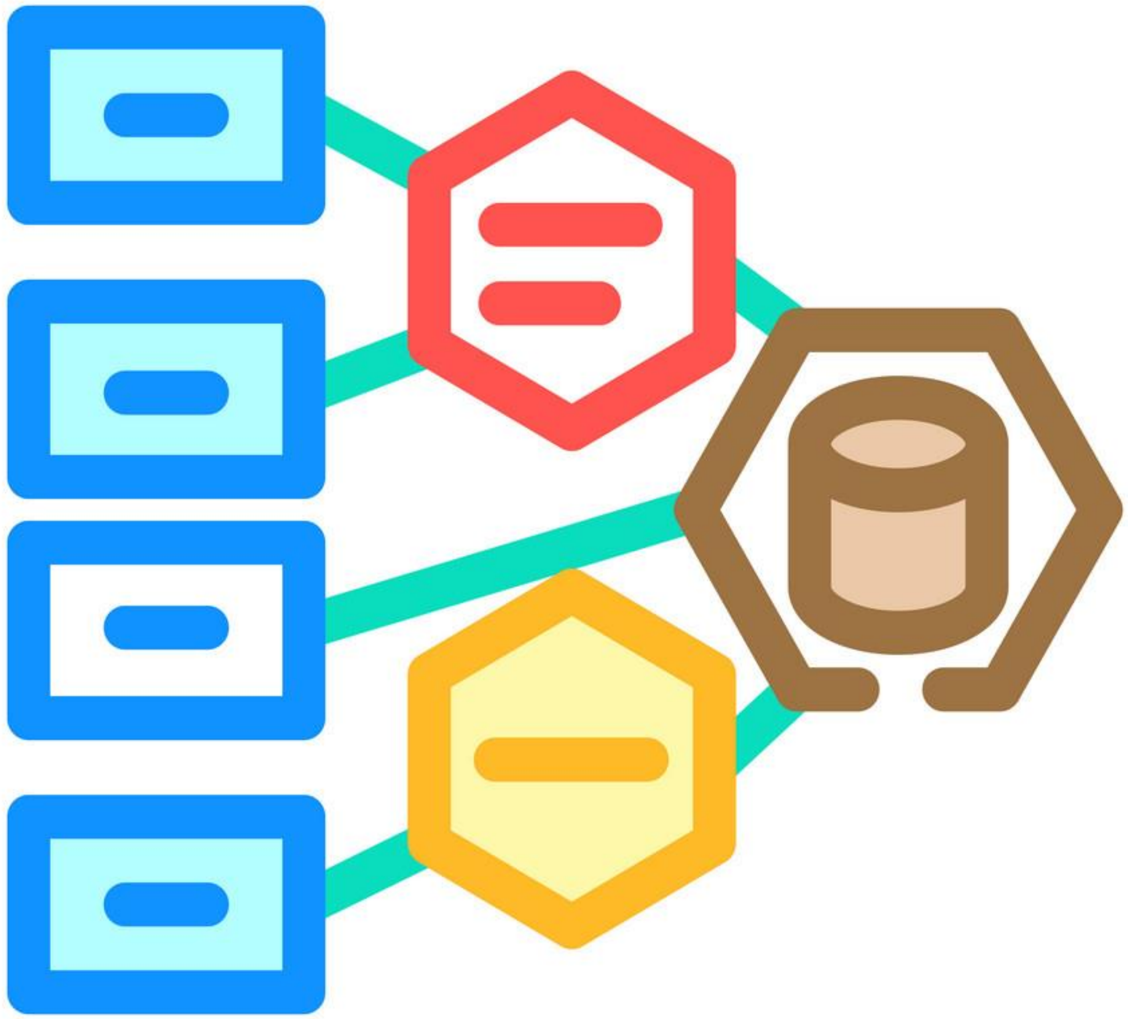
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Outline

- Architectural styles
  - ✓ Data flow architectures
  - ✓ Layered
  - ✓ Event-based
  - ✓ Data-centered
  - ✓ MVC
  - ✓ Multi-tier distributed
  - ✓ Service Oriented

Software architecture patterns

- A software architecture pattern defines the high-level structure and organization of a software system.
- It outlines the fundamental components, their interactions, and the overall layout of the system.
- Architectural patterns guide decisions about the system’s scalability, performance, and maintainability.

Data flow architectures

- ✓ Batch Sequential
- ✓ Pipes and Filters
- ✓ Event-Driven Architecture (EDA)
- ✓ Stream Processing

- **Data Flow Architecture** is a software architecture style where the system is organized around the movement and processing of data.
- It focuses on how data flows through the system, processing it in a series of transformations.
- This pattern emphasizes data transformations, where input data is transformed into output data through a series of operations.

Key Components

1. **Data Source:** Where the data originates from.
2. **Processing Unit (Transformation):** These are the modules that receive, process, and produce data.
3. **Data Sink:** The endpoint where processed data is consumed or stored.

Types of Data Flow Patterns:

Batch Sequential

- Data flows in chunks (batches) from one process to another.
- Suitable for scenarios where real-time processing is not critical.

Pipes and Filters

- The system is divided into filters (processing units) and pipes (connections that transmit data).
- Filters are independent and only transform the data passing through them.

Event-Driven Architecture (EDA)

- Data flows in the form of events that trigger processing.
- Suitable for systems with dynamic, real-time data needs.

Stream Processing

- Continuous flow of data through processing components.
- Ideal for scenarios where real-time or near-real-time processing is critical.

Batch Sequential Architecture

- **Batch Sequential Architecture** is a type of data flow architecture where data is processed in discrete, sequential batches rather than in real-time.
- In this pattern, the system processes a collection of data inputs as a single unit (batch) and moves the output of one processing stage to the next stage in sequence.
- This architecture style is common in systems where real-time processing is not necessary, and where it makes sense to accumulate data and process it in chunks or batches.

Key Characteristics

Sequential Processing: Data moves from one processing unit to the next in a linear, sequential order. Each processing unit or module completes its task before the next one begins.

Batch-Oriented: Data is accumulated into batches, meaning that a group of inputs is processed together as a single unit. Once the batch is processed, the output is sent to the next stage for further processing.

Non-Real-Time: Batch sequential systems are not designed for real-time processing. There is typically a delay between when data is collected and when it is processed.

Simplicity: The architecture is simple, as the system can work on predefined data in predefined stages. It is suitable for applications that do not require continuous updates or immediate feedback.

State Independence: Each stage of processing typically works independently of other stages, meaning each stage does not need to be aware of the inner workings of the previous or next stage.

Structure

Input Data Source: The system starts by collecting data, which can be user-generated, sensor data, database entries, etc.

Batch Process 1: The first processing unit or module takes a batch of input data and applies some form of transformation or processing to it.

Intermediate Data: The output of the first process serves as input for the next process.

Batch Process 2: The second processing unit takes the intermediate data and applies further transformations.

Output Data Sink: Once all processing stages are complete, the final output is stored or sent to its destination (e.g., stored in a database or displayed to the user).

Pros of Batch Sequential Architecture

Efficiency in Large-Scale Processing:

- Efficient for large datasets, especially when real-time results are not required.
- Systems can process large amounts of data at once, optimizing resource usage.

Predictability and Control:

- The batch sequential pattern offers predictable, controlled processing environments since each batch can be checked before processing starts.
- Batch jobs can be scheduled during off-peak hours to optimize resource utilization.

Simplicity:

- Simple to implement since processing units are arranged in a clear sequence.
- Each stage of processing is clearly defined, making the architecture easy to understand and maintain.

Fault Tolerance:

- If a batch fails during processing, only that batch needs to be reprocessed.
- There is no real-time dependency, so data integrity can be ensured by repeating the processing step.

Offline Processing:

- Useful for systems where the data does not need to be updated in real-time, like payroll systems or analytics systems that summarize data at the end of the day.

Cons of Batch Sequential Architecture

Latency:

- Batch sequential systems introduce delays since the data is not processed as it arrives.
- Data may wait in a queue for batch processing to start, causing a time lag between data collection and output.

Lack of Real-Time Feedback:

- Not suitable for applications that require immediate feedback or continuous, real-time updates, like online transactions or streaming platforms.

Resource Overhead:

- Requires more resources at specific times because data is processed in large batches, potentially leading to spikes in system resource utilization.

Inflexibility:

- Since data processing is linear, any changes to a batch in progress may disrupt the entire sequence.
- Introducing new stages or adjusting the process flow can be challenging, as each stage is tightly coupled in a sequence.

Error Handling:

- Errors can propagate through stages before being detected.
- If a problem occurs in the middle of a batch process, it can be more difficult to isolate and fix compared to real-time or modular systems.

Use Cases for Batch Sequential Architecture

Payroll Systems:

- Payroll calculations are often run in batch mode, typically at the end of a pay period.
- Data (e.g., hours worked, taxes, deductions) is accumulated, processed as a batch, and then the result is output (pay slips, transfers, reports).

End-of-Day Reporting:

- In banking and financial systems, batch sequential architecture is used for processing transactions and generating end-of-day reports, summarizing account activity, updating balances, and producing statements.

Data Warehousing:

- Batch processing is used for large-scale data integration and transformation tasks, such as loading data into a data warehouse, where data from various sources is collected and processed in a batch at scheduled intervals.

- **Image and Video Processing:**
- Media companies often use batch sequential architecture to process video or image files in batches, applying filters, rendering, or compression to a large number of files at once.
- **Log Analysis:**
- Batch processing can be used for analyzing large volumes of log files collected over a period of time, generating reports for system performance, errors, or security monitoring.
- **ETL (Extract, Transform, Load):**
- In ETL processes, data from different systems is collected, transformed, and loaded into a target system (e.g., a data warehouse). These processes are often done in batches, especially during non-peak hours.

Pipes and Filters

- The Pipes and Filters architectural pattern is a design model where components, known as filters, process streams of data in stages.
- Each filter performs a specific operation, transforms the data, and passes the result to the next filter via a connection called a pipe.
- The pattern is widely used when data transformations or streams are the core functionality of a system.

Components:

Filter:

- A processing unit that transforms the input data and produces output.
- Filters are generally stateless, meaning they do not hold any persistent state between executions.

Pipe:

- A connector that passes the output of one filter to the input of the next.
- It acts as a data conduit and facilitates the flow of data.

Details:

Data Flow:

Data flows through a sequence of filters, each transforming or filtering the data before passing it to the next.

Reusability:

Each filter is an independent unit, which increases modularity and reusability in different contexts or applications.

Flexibility:

The system can be easily extended by adding new filters or changing the order of filters without affecting the overall system design.

Pros:

Modularity:

The separation of concerns makes it easier to develop, understand, and maintain filters independently.

Reusability:

Since filters are standalone components, they can be reused in other systems or pipelines without modification.

Scalability:

The pattern can handle large volumes of data efficiently by allowing filters to run in parallel.

Testability:
Each filter can be tested independently, simplifying debugging and validation of the system.
Extensibility:
Filters can be added, removed, or rearranged to extend or modify the system’s behavior without significant redesign.

Cons:

Performance Overhead:

The transfer of data between filters introduces overhead, especially when filters are connected via external resources (e.g., file systems, networks).

Data Format Dependency:

Filters often require specific data formats. Transformations between formats can introduce complexity and errors.

Error Handling:

Managing errors across multiple filters can be challenging, especially if filters do not communicate error states effectively.

Latency:

In some cases, processing through multiple filters may introduce latency, particularly when large datasets or real-time processing is involved.

Use Cases:

Data Processing Pipelines:

Data streams (e.g., sensor data, logs) are processed in stages (e.g., filtering, transformation, aggregation) in systems like IoT or log analysis.

Compilers:

The different stages of a compiler (lexical analysis, parsing, semantic analysis, optimization, and code generation) are modeled using the pipes and filters pattern.

Stream Processing Systems:

Real-time data streaming systems (e.g., Kafka Streams, Apache Flink) implement this pattern for event processing and transformations.

Audio and Video Processing:
Multimedia processing pipelines (e.g., FFmpeg) handle different stages of media encoding, decoding, and filtering through a pipes and filters architecture.
Text Processing Tools:
Tools like Unix pipelines (grep, sed, awk) use this pattern where the output of one tool is piped to the next for additional processing.

Event-Driven Architecture (EDA)

- **Event-Driven Architecture (EDA)** is a software design pattern where components of a system communicate and interact based on events.
- In EDA, an event is a significant change in the state of a system or an action taken by a user or another system component.
- When an event occurs, one or more parts of the system (event producers) emit events, and other parts of the system (event consumers) react to them.

Key Components of EDA:

1. **Event Producers**: These generate events when something of significance happens, like a user action, system state change, or external service interaction.
2. **Event Consumers**: These are triggered by events, performing actions like processing data, invoking services, or updating systems.
3. **Event Channels**: The medium through which events are transferred from producers to consumers, often using a message broker or event streaming platform.
4. **Event Brokers/Message Brokers**: Tools like Kafka, RabbitMQ, or AWS SNS/SQS that decouple event producers and consumers, ensuring reliable delivery and enabling scalability.

Types of Event-Driven Architecture:

1. **Simple Event Processing**: Each event triggers a simple action in response.
2. **Complex Event Processing (CEP)**: Multiple events are analyzed to identify patterns or trends that lead to more complex actions or insights.

Pros of Event-Driven Architecture:

1. **Loose Coupling**: Producers and consumers do not need to be aware of each other, making the system more modular and scalable.
2. **Real-Time Processing**: EDA allows for real-time or near-real-time event handling, which is crucial in time-sensitive applications.
3. **Scalability**: Since components are decoupled and can be independently scaled, it is easier to distribute workloads across multiple instances.
4. **Resilience**: Failure in one part of the system doesn’t necessarily cause failures in other parts. Events can be queued, retried, or handled independently, improving fault tolerance.
5. **Asynchronous Communication**: Events are processed asynchronously, reducing bottlenecks and improving system performance.
6. **Flexibility**: New event consumers can be added without affecting existing producers, making the system highly adaptable to change.

Cons of Event-Driven Architecture:

1. **Complex Debugging and Monitoring**: Since events flow asynchronously, tracking the root cause of issues across multiple services can be challenging.
2. **Event Ordering**: Maintaining the order of events can be complex, especially in distributed systems.
3. **Latency**: While EDA can be real-time, the use of message brokers and network latency can introduce delays in event processing.
4. **Data Consistency**: Since components are loosely coupled and work asynchronously, maintaining consistency across systems can be challenging without proper coordination.
5. **Increased Complexity**: Managing event routing, queues, retries, and failures adds architectural complexity.
6. **Overhead**: The infrastructure required to manage events, especially at scale, can add resource and cost overhead, such as the use of message brokers and monitoring tools.

Use Cases for Event-Driven Architecture:

E-commerce Systems: When a user places an order, multiple events are triggered for processing payments, updating inventory, sending confirmations, etc.

Real-Time Analytics: Processing streams of events from IoT devices, social media platforms, or other data sources for real-time monitoring or trend detection.

Microservices Architecture: EDA is often used to coordinate microservices that need to communicate asynchronously without direct coupling.

Financial Transactions: Bank systems use EDA to react to account activity, fraud detection, or payment processing events in real-time.

IoT Systems: Devices in IoT networks often emit streams of events (e.g., sensor readings), which are processed and analyzed in real-time.
Logistics and Supply Chain: Events like shipment updates or inventory changes trigger automated workflows, such as updating dashboards, notifying customers, or placing restocking orders.
Healthcare Systems: Patient monitoring systems that trigger alerts when certain health metrics are outside safe limits, sending data to doctors or emergency systems.
Streaming Platforms: Video streaming services use EDA to track user actions like play, pause, or recommendations to personalize content.

Stream Processing

- **Stream Processing** is a computational paradigm that processes data continuously as it arrives, allowing real-time or near-real-time analysis and decision-making.
- Unlike traditional batch processing, which processes data in large blocks, stream processing deals with data as individual records (events) that flow through the system.

Key Components of Stream Processing:

1. **Data Streams**: Continuous sequences of data (events) generated by various sources like IoT devices, user interactions, logs, etc.
2. **Stream Processor**: The component that processes the data streams in real-time, often applying operations like filtering, aggregation, transformation, or windowing.
3. **Windowing**: Stream processing often requires dividing streams into logical windows (e.g., 5-second intervals) to perform operations over specific timeframes.
4. **Event Time vs. Processing Time**: Event time refers to when the event occurred, while processing time refers to when the event is processed. Stream processors often have to handle delays between these two times.
5. **Stream Processing Frameworks**: Tools such as Apache Kafka Streams, Apache Flink, Apache Samza, and Apache Spark Streaming help manage and process large streams of data in real-time.

Stream Processing Models:

1. **Stateless Processing**: Each event is processed independently without considering the state of previous events.
2. **Stateful Processing**: The processing of an event depends on the history of events or aggregated state (e.g., tracking averages over time or maintaining counters).

Pros of Stream Processing:

1. **Real-Time Data Processing**: Stream processing allows for immediate reaction to data, making it ideal for time-sensitive applications.
2. **Low Latency**: Since events are processed as they arrive, stream processing systems typically exhibit low-latency responses, enabling faster decision-making.
3. **Continuous Insights**: It provides continuous insights into data trends or anomalies, which is beneficial in scenarios where ongoing monitoring is required (e.g., fraud detection).
4. **Scalability**: Stream processing frameworks are built to handle large-scale, distributed data streams, making them scalable across multiple nodes.
5. **Event-Driven**: Stream processing is often naturally integrated with event-driven architectures, which are common in modern, scalable systems.
6. **Flexibility**: Stream processing frameworks allow for complex transformations and computations on the fly, enabling sophisticated real-time analytics.

Cons of Stream Processing:

Complexity: Building and managing a stream processing system is more complex than traditional batch processing. It involves handling windowing, event time discrepancies, and maintaining fault tolerance.

Data Ordering: Managing the correct order of events can be challenging, especially in distributed environments where events might arrive out of sequence.

Error Handling: Errors in stream processing can be more difficult to recover from compared to batch processing, where data can be reprocessed in bulk.

Consistency Challenges: Stream processing systems often sacrifice consistency for availability and speed, which can complicate use cases that require strict data integrity (e.g., financial transactions).

Resource-Intensive: Stream processing requires constant resource allocation, as the system must always be "on," processing streams in real-time. This can result in higher operational costs compared to batch systems.

Maintenance Overhead: Continuous processing introduces higher maintenance demands, as failures or bugs need to be addressed immediately to avoid system downtime.

Use Cases for Stream Processing:

Real-Time Analytics: Stream processing is widely used for real-time monitoring and reporting, such as analyzing stock market data, user engagement metrics, or website traffic.

Fraud Detection: In financial services, stream processing systems can analyze transaction data in real-time to identify suspicious behavior or detect fraud as it happens.

IoT Data Processing: IoT devices generate massive amounts of continuous data (e.g., sensor readings), which can be processed in real-time to trigger actions or inform decision-making.

Social Media and Marketing: Companies use stream processing to track user behavior on social media platforms or websites in real-time, allowing for personalized recommendations and marketing efforts.

Financial Markets: In algorithmic trading and stock market monitoring, stream processing helps execute trades or react to market conditions with minimal delay.
Telecommunication Networks: Telecom providers use stream processing to monitor network performance, detect anomalies, and manage traffic in real-time.
Recommendation Engines: Streaming data, such as user activity on streaming services (Netflix, Spotify), is processed in real-time to provide personalized recommendations based on current behavior.
Supply Chain Management: Stream processing helps monitor logistics, inventory levels, and shipments, providing real-time updates and alerts if issues arise.
Healthcare Monitoring: Wearable devices and medical equipment generate real-time data streams that can be processed to monitor patient health metrics and provide alerts for any abnormalities.
Security and Intrusion Detection: Security systems use stream processing to analyze logs and detect potential security breaches or intrusions in real-time.

Case Studies

Data flow architecture

Google’s MapReduce:

- Google’s MapReduce framework is a classic example of a **batch-processing data flow system** where large datasets are processed across distributed clusters.
- **Pros:** Scales well for large datasets, easy to distribute the processing.
- **Cons:** High latency due to batch processing.

Apache Kafka & Apache Storm:

< Stream Processing and Event-Driven >

- Kafka is widely used for building real-time data pipelines, and Storm is often used to process the streams of data in real-time.
- **Pros:** Low latency, high throughput, real-time processing.
- **Cons:** Complex to set up, requires extensive infrastructure and fault tolerance mechanisms.

Facebook’s News Feed:

< Event-Driven and Stream Processing >

- **Context:** Facebook uses a data flow model in its news feed, where data flows from various sources (posts, likes, comments) and is filtered, prioritized, and presented to users in real-time.
- **Pros:** High engagement, personalized, real-time.
- **Cons:** Handling errors or inconsistencies in real-time can be challenging.

Netflix:
< Stream Processing and Event-Driven >

Netflix uses a real-time streaming architecture for video delivery and user interactions. Data flows from user actions (e.g., play, pause, stop) through a pipeline that processes and logs this information for monitoring and recommendation systems.

- **Pros:** Real-time video quality adjustments and personalized recommendations.
- **Cons:** Scaling such a system across millions of users requires highly optimized data handling and error recovery mechanisms.

That’s it