Event-Driven Architecture & Streaming

Apache Kafka

WSMT Course

The Limitation of Synchronous REST

- We've seen REST and gRPC. These are essentially synchronous (Request/Response)
- **The Chain of Pain:**
  - If `User` -> calls `Order` -> calls `Payment` -> calls `Inventory` -> calls `Email`...
  - If the `Email` service is slow (10 seconds), the *entire* chain blocks for 10 seconds
- **Temporal Coupling:** All 5 services must be explicitly alive and available at the exact same millisecond

What is Event-Driven Architecture (EDA)?

- **Event-Driven Architecture (EDA)** is a software architecture paradigm promoting the production, detection, consumption of, and reaction to *events*
- An **Event** is a significant change in state. (e.g., `User Created`, `Order Shipped`, `Payment Failed`)
- **Producers** publish events without any knowledge of who will listen to them
- **Consumers** listen to events and act upon them without any knowledge of who produced them

Event vs Command

A critical distinction in Distributed Systems:

- **Command:** An imperative request to a single specific service to perform an action. (e.g., `ChargeCreditCard(user_id=5, amount=100)`). It has an *intent*. It can be rejected.
- **Event:** A historical fact that has *already* happened. (e.g., `CreditCardCharged(user_id=5, amount=100)`). It is immutable. It cannot be rejected or undone because it is in the past.

The Three Patterns of EDA

Martin Fowler identifies three distinct patterns for Event-Driven systems:

1. **Event Notification**
2. **Event-Carried State Transfer**
3. **Event Sourcing**

We will explore all three, as they represent increasing levels of architectural complexity.

Pattern 1: Event Notification

- The simplest pattern
- The system publishes a lightweight message explicitly stating something happened
- *Payload:* `{ "event": "UserUpdated", "userId": 1234 }`
- **The Trap:** When the `EmailService` receives this ping, it only knows User 1234 was updated. It must immediately make a synchronous REST `GET /users/1234` call back to the User Service to figure out *what* the new email address is

Pattern 2: Event-Carried State Transfer

- *Solution to the Notification trap*
- The Producer publishes the event *and* includes all the relevant data necessary for the consumer to do its job
- *Payload:* `{ "event": "UserUpdated", "userId": 1234, "oldEmail": "bob@abc.com", "newEmail": "bob@xyz.com" }`
- **The Benefit:** The `EmailService` never has to call the `UserService` REST API. It has everything it needs to send the confirmation email

What is Event Sourcing?

- How do you store data in a database? Usually, you store the *current* state
- If User A balance is $100, and they deposit $50, you `UPDATE Users SET balance=150`
- You lose the history of the $100
- **Event Sourcing:** You do not store the current state. You store the immutable *sequence of events* that led to the state
- `[AccountCreated, Deposited(100), Deposited(50)]`

Advantages of Event Sourcing

1. **Auditing:** You have a mathematically perfect, immutable audit log of exactly what happened to the system since Day 1
2. **Time Travel:** You can rebuild the exact state of the `User` object as it existed on March 4th at 2:00 PM simply by replaying the event stream up to that exact millisecond
3. **Bug Recovery:** If the `CalculateLoyaltyPoints` microservice had a bug for 3 weeks, you deploy the fix, delete its database, and simply *replay the last 3 weeks of events* through the fixed code

Event Sourcing Drawbacks

1. **Complexity:** It is incredibly difficult to mentally parse for developers used to simple CRUD apps
2. **Event Schema Evolution:** What if the JSON structure of `OrderPlaced` changes radically in 2025, but your system needs to replay events from 2020? You must write complex "Upcasters" to translate old events into new formats on the fly
3. **Storage Costs:** You are storing every action a user has ever taken forever. The database grows infinitely

CQRS and Event Sourcing Together

- If you don’t store the `balance=150` in the database, how do you query a user’s balance?
- You can’t replay 10,000 events every time they open the mobile app. It’s too slow
- **CQRS (Command/Query Responsibility Segregation)**
- The "Write Side" appends the transaction to the Event Sourced log
- The "Read Side" listens to the log in real-time, calculates the `150` balance, and saves it into a fast Redis cache or MongoDB collection
- The Mobile app only queries the Redis cache

CQRS Architecture Diagram

- `Mobile App -> (Command) -> API -> Event Store (Append Only)`
- `Event Store -> (Publishes Event) -> Read Projector`
- `Read Projector -> (Calculates Data) -> Read DB (MongoDB)`
- `Mobile App -> (Query) -> API -> Read DB`
- The Write Database and Read Database can be entirely different physical technologies!

Choosing the Right Broker

Traditional vs Modern Brokers

- To build an Event-Driven Architecture, you need a high-speed network pipe to carry the events
- **Traditional Message Queues:** RabbitMQ, ActiveMQ, Amazon SQS
- **Modern Streaming Platforms:** Apache Kafka, Amazon Kinesis, Google Pub/Sub
- What is the difference? Why is Kafka suddenly everywhere?

Queue-Based Messaging (RabbitMQ)

- **Destructive Read:** In RabbitMQ, a message sits in a queue. When Consumer A reads the message and acknowledges it (`ACK`), RabbitMQ *deletes* the message permanently from RAM/Disk
- If Consumer B comes online an hour later and wants to read that message, it is impossible. It is gone
- *Transient Storage:* Queues are designed to be empty. If the queue is full, the system is backed up

Log-Based Messaging (Apache Kafka)

- **Immutable Append-Only Log:** In Kafka, messages are written to a massive sequential log file on disk
- When Consumer A reads event #4, Kafka does NOT delete it
- Consumer A simply updates a pointer (called an **Offset**), moving from 4 to 5
- Consumer B can connect, start its offset at #0, and read all events from the beginning of time

Introduction to Apache Kafka

- Originally developed at LinkedIn in 2011 to handle trillions of server logs per day
- It is not a traditional Message Broker. It is a **Distributed Streaming Platform**
- *Capabilities:*
  1. Publish & Subscribe to streams of records
  2. Store streams of records securely and durably on disk
  3. Process streams of records as they occur in real-time (Kafka Streams)

Kafka Core Concepts: Topics

- A **Topic** is a category or feed name to which records are published
- Topics in Kafka are always multi-subscriber
- A topic can have zero, one, or many consumers that subscribe to the data written to it
- *Analogy:* A Topic is simply a massive database table, but you can only `INSERT` at the bottom, and you cannot `UPDATE` or `DELETE`

Kafka Core Concepts: Partitions

- If a Topic receives 1,000,000 events per second, it cannot fit on one physical server's hard drive
- **Partitions:** A Topic is sliced logically into multiple Partitions (Partition 0, Partition 1, Partition 2)
- Each Partition can be stored on a completely different physical server (Broker)
- This is the secret to Kafka's infinite horizontal scalability

Message Ordering in Kafka

- Does Kafka guarantee ordering? (e.g., Do I process `Cart Item Added` before `Checkout Complete`?)
- **Yes, but ONLY within a single Partition**
- Kafka does not guarantee global ordering across an entire Topic
- How do you ensure Bob's checkout events stay in order? By using a **Message Key** (e.g., `user_id=Bob`)
- Kafka guarantees that all messages with the exact same Key always go to the exact same Partition

Kafka Brokers and Clusters

- A single Kafka server is called a **Broker**
- A group of Brokers working together is a **Kafka Cluster**
- Historically, clusters used **Zookeeper** to manage consensus
- **SOTA: KRaft (Kafka Raft)** natively manages consensus across brokers without Zookeeper
- **Replication:** Every partition is duplicated across multiple brokers (usually 3). If a broker bursts into flames, the data is perfectly safe

Kafka Consumers and Consumer Groups

- If 1 million events hit a topic per second, one single Node.js consumer process will crash trying to process them
- **Consumer Groups:** You boot up 50 identical Node.js containers and put them in the same `Consumer Group`
- Kafka automatically evenly distributes the Partitions across the 50 containers
- If Container 12 crashes, Kafka detects it and instantly re-assigns its partitions to Container 13

Kafka Topics under the Hood (Offsets)

- `[ 0 ] [ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ]` -> The Ordered Partition Log
- The **Offset** is just a simple integer indicating the position in the log
- The Consumer reads offset `4`. It then sends a hidden message to Kafka saying: "I successfully processed offset 4. Please commit this."
- If the consumer crashes and reboots, it asks Kafka: "Where was I?", Kafka replies "You were at offset 4. Start at 5."

The Delivery Semantics (At-most vs At-least)

Because networks fail, how does Kafka handle crashes?

1. **At-Most-Once:** Consumer reads message, Commits Offset instantly, then processes. If processing crashes, message is skipped forever. (Fast, dangerous)
2. **At-Least-Once (Default):** Consumer reads message, processes to Database, *then* Commits Offset. If it crashes after Database write but before Commit, upon reboot it will read the message again and write duplicate data

Delivery Semantics: Exactly-Once

- Is true **Exactly-Once** processing possible in Distributed Systems?
- Yes. Kafka introduced the **Transactional API** (Idempotent Producers + Transactional Consumers)
- It mathematically guarantees that even if a Producer crashes while pushing data, or a Consumer crashes while reading, the event is processed and stored in downstream topics *exactly one single time*
- *Tradeoff:* Significantly higher latency and CPU overhead

Kafka Code Example (Node.js Producer)

Using `kafkajs` library:

```
const { Kafka } = require('kafkajs')

const kafka = new Kafka({ clientId: 'my-app', brokers: ['kafka1:9092'] })
const producer = kafka.producer()

const run = async () => {
  await producer.connect()
  await producer.send({
    topic: 'ecommerce-orders',
    messages: [
      // Bob's events always hash to the same partition
      { key: 'user_id_bob', value: JSON.stringify({ action: "PAYMENT_SUCCESS" }) },
    ],
  })
}
run().catch(console.error)
```

Kafka Code Example (Node.js Consumer)

```
const consumer = kafka.consumer({ groupId: 'email-service-group' })

const run = async () => {
  await consumer.connect()
  await consumer.subscribe({ topic: 'ecommerce-orders', fromBeginning: true })

  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      console.log(`Received msg on partition ${partition}: ${message.value.toString()}`)
      // Assume sendEmail() is Idempotent!
      await sendEmail(message.value)
      // KafkaJS automatically commits the offset after this function completes!
    },
  })
}
run().catch(console.error)
```

Apache Kafka vs Service Mesh

- Why not just use gRPC and Istio to connect the internal services?
- gRPC provides synchronous data
- **Kafka provides highly decoupled asynchronous state**
- If the `FraudDetection` microservice takes 5 minutes to analyze a transaction via an obscure Machine Learning model, it absolutely must be done asynchronously via Kafka off the critical path of the user's checkout flow

Confluent and the Kafka Ecosystem

- Kafka is bare-code open source
- **Confluent** is the major enterprise company driving Kafka (founded by Kafka's original creators)
- They provide managed Cloud Kafka (Confluent Cloud) so you don't have to manage Zookeeper/KRaft nodes
- They also provide the **Confluent Schema Registry**

Schema Registry (Avro)

- You do not send raw JSON into Kafka in the Enterprise
- You use **Apache Avro** (a binary serialization system similar to Protobuf)
- Producer: "I want to send an OrderEvent."
- Producer contacts Schema Registry: "Is this valid OrderEvent shape?"
- Registry verifies, returns a schema ID
- Producer serializes to binary Avro and sends it to Kafka

Stream Processing (Kafka Streams)

- Kafka was initially just a pipe mechanism
- **Kafka Streams:** A Java library that sits on top of Kafka
- It allows you to read a stream, perform complex SQL-like aggregations (Windowing, Joins, Filtering), and write the results to a new Kafka topic in real-time
- *Example:* "Give me the rolling average transaction cost of the last 60 seconds of events coming from Partition 4, updated every millisecond."

KSQL (SQL for the Stream)

- Writing Java Kafka Streams code is hard
- **ksqlDB:** Allows you to query moving data using standard SQL syntax!

```
CREATE STREAM VIP_USERS AS SELECT * FROM users WHERE status = 'VIP';
```

This effectively creates a continuous real-time pipeline that siphons VIP events out of the main topic and into a new topic indefinitely without writing a single line of Java.

Log Compaction

- We said Kafka stores things forever, meaning infinite disk space. Is that true?
- **Log Compaction:** A feature where Kafka looks at message Keys
- If your key is `user_1` and you have events: `[UpdateName="Alice", UpdateEmail="x@x.com", UpdateName="Alice Smith"]`
- Kafka can periodically compact the log by deleting the older versions, keeping only the absolute latest message for a particular key

SOTA: Tiered Storage

- Keeping petabytes of data on fast NVMe Kafka Broker SSDs is too expensive
- **SOTA: Tiered Storage**
- Kafka automatically moves "hot" new real-time events to expensive NVMe SSDs
- Kafka seamlessly and transparently archives "cold" 6-month-old historical events down to ultra-cheap Amazon S3 Object Storage!
- Consumers can pull from S3 as if it were still on the hard drive via the same API!

The Event-Carried State Transfer Problem

- We mentioned this pattern earlier. Microservice B keeps a local copy of Microservice A's data by listening to Kafka
- *The Danger:* Data Duplication
- If we have 50 microservices, and 30 of them keep a cached copy of the `User` object, we are paying massive database storage costs globally
- Ensure that consumers only cache the absolute minimum variables from the event that they explicitly need to function

Outbox Pattern (Transactional Safety)

- **The Scenario:** A REST endpoint creates a User in Postgres, and sends a "UserCreated" event to Kafka
- What if Postgres succeeds, but the network crashes before the Kafka message is sent? The system is in an inconsistent state forever
- **Solution: The Transactional Outbox Pattern**

Outbox Pattern Architecture

1. Begin DB Transaction
2. `INSERT INTO users (name, email) ...`
3. `INSERT INTO event_outbox (event_type, payload) VALUES ('UserCreated', '{...}')`
4. Commit DB Transaction. (Either both succeed, or both fail)
5. A separate background worker constantly polls the `event_outbox` table and safely pushes those messages into Kafka

CDC (Change Data Capture) with Debezium

- How does the outbox polling work? Writing a polling script is inefficient
- **Debezium:** An open-source distributed platform for Change Data Capture
- It taps directly into the lowest level of the PostgreSQL transaction log (`WAL`), detects the `INSERT INTO event_outbox` at the physical disk level, and instantly streams it directly into Kafka

Handling Event Ordering Issues

- What if a consumer receives `UserUpdated` before it receives `UserCreated`?
- This happens if networks are partitioned or if consumer containers scale down incorrectly
- *Defense 1:* Ensure Kafka Partitions use the `user_id` as the Key to enforce strict ordering
- *Defense 2:* Attach strict `timestamp` or `version_id` metadata to every event payload. If a consumer sees an event with `version=3`, but its local database is on `version=1`, it can reject the message back to the queue to process later

WebHooks vs Streaming

- If an external company (like Stripe) wants to send you an Event ("Payment Succeeded"), do they connect directly to your internal Kafka cluster?
- No. Kafka operates on binary TCP and requires specific client drivers
- **WebHooks:** The industry standard for passing events over the public internet
- Stripe makes a standard `POST /webhook` REST call to your server. Your API Gateway translates that JSON REST call into an internal binary event on your Kafka topic

Real-World Use Case: The Recommendation Engine

1. User browses Amazon. Every product page load issues a tiny asynchronous event into the Kafka `clickstream_topic`
2. A Kafka Stream processor in the background aggregates: "User 123 looked at 5 dog food pages in 60 seconds."
3. The Stream processor dynamically updates the Redis Cache: "User 123 is interested in Dogs."
4. User navigates to the Homepage. The Web Server queries the Redis Cache in 2ms and serves an advertisement for Dog collars

Managed Cloud Alternatives (AWS Kinesis / GCP PubSub)

- Running KRaft and Kafka Brokers requires a dedicated DevOps engineering team
- What if you want streaming data, but you are a small team on Serverless?
- **AWS Kinesis:** Amazon's managed, serverless alternative to Kafka. It works identically (append-only log, partitions, offsets)
- **GCP Pub/Sub:** Google's managed alternative. Slightly different architecture (no explicit partitions/offsets, massive global routing map), but fulfills the same EDA requirement

Architecting the Payload

When sending an Event across the company, establish a strict standard:

```
{
  "eventId": "123e4567-e89b-12d3",
  "eventType": "eu.company.domain.order.created.v1",
  "timestamp": "2024-10-21T10:00:00Z",
  "source": "/services/checkout/v1",
  "data": {
     "userId": 55,
     "orderTotal": 25.50
  }
}
```

The CloudEvents Specification

- Because parsing that JSON metadata was chaotic across different tools, the CNCF created an open standard
- **CloudEvents:** A specification for describing event data in common formats to provide interoperability across services, platforms and systems
- It mandates standard headers (e.g., `ce-id`, `ce-source`, `ce-type`) via HTTP context

The Danger of "The Distributed Monolith"

- Event-Driven Architecture is powerful, but overusing it leads to the "Distributed Monolith Error"
- If Service A publishes an event, and Service B, C, and D *all immediately fail* if the Kafka message doesn't arrive within 500ms…
- You have essentially built a tightly coupled Monolith, but you've added the unreliability of a network and a message broker in the middle of it

Monitoring EDA (Distributed Tracing)

- In REST, you look at the user request ID in the application log
- In EDA, tracking an error is nearly impossible. A user clicked a button, and 8 hours later an obscure asynchronous Kafka consumer threw a Java exception
- **Distributed Tracing (OpenTelemetry/Jaeger):** You must extract a unique `TraceID` from the initial user HTTP request, attach it to the Kafka Event metadata, extract it in the Consumer, and log it

Testing Event Driven Systems

- You cannot use Jest or JUnit to easily mock an infinite stream of real-time server events
- **Testcontainers:** An incredibly popular library (Java/Node/Go)
- During your CI/CD test phase, Testcontainers actually spins up a real, temporary Docker container running Apache Kafka
- Your backend integration tests publish real messages to the isolated real Kafka container, wait 2 seconds, and assert the consumer processed them correctly

Dead Letter Queues in Kafka

- RabbitMQ has built-in DLQs. Does Kafka?
- Kafka does *not* automatically route broken messages. The Consumer just stops processing and hangs dynamically
- You must *manually* write a `try/catch` block
- In the `catch`, you programmatically use a Producer to push the broken message into a separate `topic.DLQ`, and then artificially commit the original offset to keep moving forward

Advanced Sagas with Kafka

- In Lecture 9, we discussed the Saga Pattern for distributed transactions
- Kafka is the perfect backend to drive a Choreographed Saga
- *Order Service* -> `OrderCreatedEvent`
- *Payment Service* -> Listens -> `PaymentAcceptedEvent`
- *Inventory Service* -> Listens -> `ItemsReservedEvent`
- If Inventory fails -> `InventoryFailedEvent` -> Payment Service listens to this and triggers a `Refund`

The Problem of Large Payloads

- What if a user uploads a 4GB Video? Do you send a 4GB message through Kafka?
- No. Kafka default message size limit is 1 Megabyte! (Massive files destroy stream latency)
- **The Claim Check Pattern:**
- You save the 4GB video to an Amazon S3 Bucket
- You publish a tiny Kafka event: `{ "event": "VideoUploaded", "video_s3_url": "s3://bucket/video.mp4" }`

When should I NOT use Kafka?

- Do not use Kafka if you just want to run a background job (e.g., "resize this image"). Use simple queuing like RabbitMQ or AWS SQS
- Do not use Kafka if you require strictly synchronous data
- Do not use Kafka if you have a massive amount of incredibly slow consumers (e.g., each message takes 60 seconds to process). Kafka excels at moving millions of tiny, fast messages per second

Implementing WebSocket Realtime Updates

- If the architecture is asynchronous, how does the User Interface know the `Payment` finished? They just submitted an HTTP `POST` and got an immediate `202 Accepted`
- **WebSockets (SignalR/Socket.io)**
- The Frontend maintains a constant WebSocket to an Edge Gateway
- When the final `OrderCompletedEvent` reaches the end of the Kafka pipeline, an Edge Microservice listens for it, matches the User ID, and pushes a real-time WebSocket packet to the browser
- The browser flashes green: "Order Complete!"

The Transition Strategy (Strangler Fig revisited)

- Moving a legacy monolithic SQL database to Kafka Event Sourcing is terrifying
- Use the **Strangler Fig Pattern**
- Start by attaching **Debezium** to the legacy SQL database
- Let the legacy Monolith keep writing to SQL normally
- Debezium streams the changes into Kafka topic
- Build new Microservices that read from the Kafka topic

Kafka Security

- Kafka clusters hold the entire lifeblood of the corporate data
- **Authentication:** SASL/SCRAM or mTLS (Mutual TLS). Ensure no rogue developer laptop can connect to the production cluster
- **Authorization:** Kafka uses an ACL system. E.g., The `BillingService` ConsumerGroup is allowed to read from `orders`, but not allowed to write to it
- **Encryption:** Configure TLS encryption "in transit" and encrypt the physical broker hard drives "at rest"

Event Sourcing Complexities: Upcasting

- If you deploy v2.0 of your software, and it expects `{"first_name": "John", "last_name": "Doe"}`, but your historical event stream from v1.0 contains `{"name": "John Doe"}`...
- When the Event Sourcing engine replays v1.0 events, your v2.0 code will crash
- **Upcasting:** A layer of functional middleware that intercepts older-schema events during a replay and mutates them into the v2.0 expected shape on the fly before passing it to the business logic

Event Sourcing Complexities: Snapshotting

- If a bank account has 500,000 transaction events since 2010... rebuilding the balance `(0 + 10 - 5 + 20 ...)` takes 5 minutes of server CPU every time the service reboots
- **Snapshotting:** Every night, an engine calculates the current balance ($20,000) and writes a `Snapshot` event to a completely different database
- When the service reboots, it reads the Snapshot ($20,000) from last night, and only replays the 5 transaction events that occurred *today*

Summary of Event Driven Architecture (EDA)

- Decouples microservices. They no longer care if downstream partners are offline
- **Event Sourcing** provides an immutable, mathematically perfect audit log by never overwriting data
- **CQRS** separates heavy data writes from highly optimized caching reads
- Massive corporate agility: If a new team wants access to Order Data, they simply subscribe to the Kafka topic. The Order Team does not have to change any code

Summary of Apache Kafka

- A Distributed, highly available, log-based streaming platform
- Not a traditional queue. Messages are retained on disk for massive historical replay
- **Topics:** Virtual locations for categorizing logs
- **Partitions:** Physical slices of Topics enabling infinite horizontal cluster scaling
- **Offsets:** Simple numeric pointers tracking consumer progress

The Real-World Engineering Tradeoffs

- Event-Driven systems are extremely powerful
- However, they are difficult to trace, difficult to test, and introduce "Eventual Consistency" to your front-end web applications
- Do not default to Event-Driven Architecture simply because it is a modern buzzword
- Begin with a Monolith, transition to synchronous REST microservices, and only introduce Kafka when you physically reach processing bottlenecks

Recap of Communication Patterns

So far in this course, we have covered:

1. **SOAP:** XML, WSDLs, strict enterprise SOAP envelopes
2. **REST:** JSON, standardized HTTP methods, cacheable web-native logic
3. **gRPC:** Protobufs, binary, HTTP/2 multiplexing for datacenter speed
4. **GraphQL:** Resolvers, customized client queries for React Apps
5. **MOM/Kafka:** Asynchronous, immutable append-only logs for massive decoupling

Q&A

Questions?