

Kafka: From Zero to Semi-Hero



Building Event-Driven Systems with Apache Kafka

Connected Train Platform Workshop



About Us

Francesco Lacriola — Software Engineer, Senior Analyst

I'm Francesco Lacriola, a Senior Analyst & Software Engineer at Fincons Group, where I have worked for over six years. I've contributed to large-scale projects across Energy, Finance and Transportation (currently on the SBB project). My role blends technical expertise, curiosity for innovation and a strong focus on software quality. I enjoy exploring cutting-edge technologies and improving development practices. I believe in teamwork, continuous improvement and knowledge sharing as drivers for personal and professional growth. Outside of technology, my greatest passion is music — it helps me relax and recharge. 🎵

Michele Caccia — Senior Consultant

I'm Michele Caccia, a Senior Consultant at Fincons since 2019. After a short but formative experience at Corner Bank, I joined the SBB project in Bern.

I've been a technology enthusiast since childhood and I enjoy exploring new

What is Apache Kafka?

Distributed streaming platform for building real-time data pipelines

- Originally developed by LinkedIn, now open-source (Apache)
- Think of it as a "nervous system" for data in your organization
- Handles **trillions** of messages per day at companies like Netflix, Uber, LinkedIn

Key Features:

- High throughput & low latency
- Fault-tolerant & scalable
- Persistent storage
- Real-time processing

Topics: Your Data Channels

Topic = A category or feed name to which records are published

- Like a table in a database or a folder in a filesystem
- Topics are **multi-subscriber** (many consumers can read the same topic)

Examples in our workshop:

- `train-locations` - raw position data from trains
- `train-speed-averages` - processed speed analytics

Partitions: Scalability & Parallelism ⚡

- Each topic is divided into **partitions**
- Partitions allow parallel processing
- Messages in a partition are **ordered**
- Each message has an **offset** (sequential ID)
- Our `train-locations` topic has **3 partitions**

```
Topic: train-locations
└─ Partition 0: [msg0, msg3, msg6, ...]
└─ Partition 1: [msg1, msg4, msg7, ...]
└─ Partition 2: [msg2, msg5, msg8, ...]
```

Producers: Publishing Data



Applications that **publish** (write) records to topics

- Can specify which partition to send to (or let Kafka decide)
- Can use keys for consistent routing
- In our workshop: **TrainLocationSimulator** (Java)

Example:

Train T-123 → partition based on train ID → always same partition

This ensures all messages from the same train stay in order!

Consumers: Reading Data

Applications that **subscribe** to topics and process records

- Maintain an **offset** to track what they've read
- Can rewind and replay messages

In our workshop:

- **SpeedAnalysisStream** (Kafka Streams)
- **DashboardWebApp** (Java)
- **MaintenanceAlerter** (Python)

Consumer Groups: Team Work!

Multiple consumers working together as a **group**

- Each partition assigned to **only one** consumer in the group
- Enables **parallel processing** and **load balancing**
- If a consumer fails, its partitions are reassigned → **fault tolerance**

Key Point:

- ✓ Same group = share the workload
- ✓ Different groups = each gets all messages

Partition Rebalancing

What happens when consumers join/leave a group?

Scenario 1: 1 consumer, 3 partitions → consumer reads all 3

Scenario 2: 3 consumers join → each gets 1 partition

Scenario 3: 1 consumer leaves → partitions redistributed

Benefits:

- Automatic scaling
- Fault tolerance
- Zero message loss
- No duplicate processing

Our Workshop Architecture



Services:

1. **Producer:** TrainLocationSimulator (Java) - generates train positions
2. **Stream Processor:** SpeedAnalysisStream (Kafka Streams) - calculates averages
3. **WebSocket Dashboard:** DashboardWebApp (Java) - real-time visualization
4. **Maintenance Alerter:** MaintenanceAlerter (Python) - slow train alerts
5. **Dashboard Consumer:** Demonstrates partition rebalancing

All connected through Kafka topics! 

Kafka Streams

Library for building stream processing applications

- Processes data **as it arrives** (real-time)
- Supports transformations, aggregations, joins, windowing
- No separate cluster needed (unlike Spark/Flink)

Our Use Case:

1. Read from `train-locations`
2. Calculate average speed per train (10-second window)
3. Write to `train-speed-averages`

Kafka is Language Agnostic

Any language can produce/consume from Kafka:

-  Java (most common, best support)
-  Python
-  JavaScript/Node.js
-  Go
-  C#/.NET
- And many more...

Workshop Demonstration:

 Java producers and consumers

 Python consumer (maintenance alerter)

 Both read from the same topic seamlessly

Kafka Monitoring Tools



Command Line Tools:

```
kafka-topics --list  
kafka-topics --describe --topic train-locations  
kafka-consumer-groups --describe --group dashboard-group
```

Web Interface:

- **Kafka UI (localhost:8099)**
 - Monitor topics, partitions, consumer groups
 - View messages in real-time
 - Track consumer lag

Where is Kafka Used?

Industries & Applications:

-  **Netflix**: Real-time recommendations & monitoring
-  **Uber**: Real-time pricing, trip tracking
-  **LinkedIn**: Activity streams, operational metrics
-  **PayPal**: Risk detection, fraud prevention
-  **Banking**: Transaction processing, fraud detection
-  **E-commerce**: Inventory management, order processing

Common Patterns:

Event sourcing • CQRS • Data pipelines • Microservices communication

Kafka vs RabbitMQ/ActiveMQ



Kafka Advantages:

- ✓ Higher throughput (millions of msgs/sec)
- ✓ Persistent storage (replay capability)
- ✓ Horizontal scalability
- ✓ Built for streaming & big data

Traditional Messaging:

- ✓ Complex routing (exchanges, bindings)
- ✓ Lower latency for small messages
- ✓ Message priority queues
- ✓ Simpler request/reply patterns

When to use Kafka:

High volume event streaming • Log aggregation • Real-time analytics • Event sourcing

Delivery Semantics & Guarantees



Three Delivery Modes:

1. At-most-once (0 or 1)

- Messages may be lost but never redelivered
- Fastest, lowest guarantee

2. At-least-once (1 or more) ★ Default

- Messages never lost but may be redelivered
- Most common

3. Exactly-once (exactly 1)

- Most complex, requires idempotent producers
- Highest guarantee, newer feature

Production Best Practices

Design Patterns:

-  Use shared libraries for data models (avoid duplication)
-  Schema Registry for data contracts (Avro, JSON Schema)
-  Version your message formats
-  Use message keys for partitioning logic
-  Configure appropriate retention policies

Operational:

-  Monitor consumer lag
-  Right-size partition count
-  Enable authentication & encryption (production)

Architectural Simplifications !

Workshop Approach:

- Shared package structure
- `TrainPosition` class copied
- Simple for learning

Production Approach:

- Separate shared library/JAR
- Schema Registry
- Proper versioning
- Microservice data ownership
- API contracts

Key Takeaway: Workshops simplify for clarity; production requires more rigor

Hands-On Time!

What You'll Do:

1.  Set up Kafka infrastructure (Docker)
2.  Start the train location producer
3.  Run the stream processor
4.  Launch the WebSocket dashboard
5.  Add Python maintenance alerter
6.  Test partition rebalancing with scaling

Prerequisites Check:

 Docker & Docker Compose •  Java JDK 17+ •  Maven •  Python 3.8+

Live Demonstration



Demo Flow:

1. Show Kafka UI - topics & partitions
2. Start producer - see messages in UI
3. Start stream processor - show transformation
4. Launch dashboard - see real-time updates
5. Start Python alerter - show interoperability
6. Scale consumers - demonstrate rebalancing

Interactive: Browser tabs, terminal windows, Kafka UI monitoring

Partition Rebalancing in Action

Live Scaling Demo:

```
# Start with 1 consumer  
docker-compose up -d  
  
# Scale to 3 consumers  
docker-compose up -d --scale dashboard-consumer=3  
  
# Scale to 2 consumers  
docker-compose up -d --scale dashboard-consumer=2  
  
# Back to 1  
docker-compose up -d --scale dashboard-consumer=1
```

Watch: Partition reassignment • Rebalancing events • Zero message loss

What You Learned Today

1.  **Kafka Fundamentals:** Topics, partitions, producers, consumers
2.  **Consumer Groups:** Workload sharing & fault tolerance
3.  **Partition Rebalancing:** Automatic load balancing
4.  **Language Interoperability:** Java + Python working together
5.  **Stream Processing:** Real-time transformations with Kafka Streams
6.  **Monitoring:** CLI tools & web interfaces
7.  **Scalability:** Horizontal scaling patterns

Avoiding Common Mistakes !

Top Issues:

1. **Connection refused:** Kafka not running → `docker ps`
2. **Deserialization errors:** Mismatched serializers
3. **Consumer lag:** Too few consumers for partition count
4. **Wrong consumer group:** Competing consumers
5. **Offset reset issues:** Check `auto.offset.reset config`

Debug Commands:

```
docker-compose logs kafka  
kafka-consumer-groups --describe --group <group-id>
```

Continue Your Kafka Journey



Deep Dive Topics:

- Advanced Kafka Streams (joins, state stores)
- Security (SSL, SASL, ACLs)
- Multi-datacenter replication
- Performance tuning
- Schema evolution strategies

Learning Resources:

- [Confluent Documentation](#)
- "Kafka: The Definitive Guide" (O'Reilly)

Questions? 🤔

Common Questions to Anticipate:

- How does Kafka compare to Kinesis/Pulsar?
- When should I use Kafka vs a traditional database?
- How do I handle schema changes?
- What's the maximum throughput?
- How do I monitor Kafka in production?

Open forum for your questions!

Thank You! 🙏

- Workshop repository: [GitHub](#)
- Join the community: Kafka Slack/Discord

Call to Action:

- ★ Star the repository
- ⟳ Try building your own use case
- 📣 Share your experience

Happy Streaming! 🚂💨