Kafka System Design 1: Event-Driven Microservices with Kafka

✨ Use Case:

In a system like Uber, after a payment is successfully processed, several services need to respond:

- Notify the driver for payout
- Send a ride receipt email
- Log the transaction for fraud detection
- Push the data into analytics/BI systems

✅ Architecture Overview:

Flow:

```
[Payment Service] (Producer)
    |
    |-- produce event to Kafka topic: payment.success
    V
[Kafka Topic: payment.success]
    |
    |-- Consumed independently by:
    |    [Payout Service (Consumer Group 1)]
    |    [Email Service (Consumer Group 2)]
    |    [Fraud Service (Consumer Group 3)]
    |    [Analytics Service (Consumer Group 4)]
```

🔧 Technical Highlights:

- Kafka Topic: `payment.success`
- Partitioned by: `order_id` or `user_id`
- Multiple consumer groups: each downstream system gets the event independently
- Offsets: committed after successful processing
- Idempotent consumers: each consumer deduplicates using `payment_id`
- Schema Management: Avro + Schema Registry

✨ Benefits:

- **Decoupled architecture**

- Independent scaling of services
- Fail-safe: one consumer fails, others continue **(i.e it is loosely coupled)**
- Replayable: messages can be replayed from Kafka

Kafka System Design 2: Outbox Pattern (DB + Kafka Sync without 2PC)

✨ Use Case:

When a service needs to:

1. Save business data to a database (e.g., ride order)
2. Publish an event to Kafka (e.g., `ride_created`)

And we need both to succeed reliably **without using 2PC (two-phase commit)**.

✅ Architecture Overview:

Flow:

```
[Order Service]
  |-- INSERT INTO orders (ride data)
  |-- INSERT INTO outbox_events (event payload, status = 'PENDING')
  |-- COMMIT  -->> (Atomic DB transaction)
      |
      |-- [Poller / Debezium CDC]
          |-- Reads rows with status = PENDING
          |-- Sends message to Kafka topic: ride.orders
          |-- Marks row as status = SENT

[Kafka Topic: ride.orders]
   |-- [Dispatch Service]
   |-- [Email Service]
   |-- [Pricing/Fraud/Analytics]
```

a. `rides` table (business table)

ride_id	user_id	pickup	drop	fare	status	created_at
R123	U001	MG Road	BTM Layout	180.0	confirmed	2024-05-10T18:30:00Z

---

b. Outbox Table Example:

event_id	event_type	aggregate_id	payload	status
E001	ride_created	R123	{"ride_id": "R123", "user_id": "U001"}	PENDING

🔍 Why Not Write Directly to Kafka in Transaction?

- Kafka doesn't support 2PC (cannot be part of a DB transaction)
- Risk of inconsistency if DB commit succeeds but Kafka fails
- Outbox solves this: all events originate from committed data

---

✅ Why It's Safe:

- DB write and event draft = 1PC
- Kafka publish is decoupled = retry-safe
- No 2PC overhead, no locks across systems
- Debezium makes it zero-maintenance (CDC auto-publishes)

🌟 Real-World Reliability:

- If Kafka is down: message stays in DB as PENDING
- If DB fails: transaction rolled back
- If Poller fails: retry later

Here it is 2 independent 1PC (1 Phase Commit) (i.e DB Commit and Kafka Commit)

`rides` table	`outbox_events` table
Core business data	Kafka-ready message queue
Complex updates	One row = one clean event
No event status	Has status = PENDING/SENT
Schema is normalized	Payload is structured JSON/Avro
Unsafe to stream from directly	Built exactly for streaming

System Design 3 – Uber Example: Ride Lifecycle with Debezium CDC

✅ 1. Use Case

> *"Let's say I'm designing Uber's backend where we want to stream all changes to the `rides` table — like fare updates, status changes, driver location — to Kafka, so that downstream services like fraud detection, real-time dashboards, ML models, and data lakes get this data in real time. But I don't want to modify the existing microservices or write custom Kafka producers in each one."*

→ **Solution: Debezium-based CDC pipeline.**

✅ 2. System Architecture (High-Level)

pgsql
CopyEdit

```
[Uber App / Backend Services]
        |
        | (INSERT, UPDATE, DELETE into rides table)
        v
[PostgreSQL Database]
        |
        | (WAL - write-ahead log) (For MySQL-BINLOG)
        v
[Debezium CDC Connector via Kafka Connect]
        |
        | (generates structured change events)
        v
[Kafka Topic: db.rides]
        |
        |----→ [ML Model: Driver ETA, Fraud Signals]
        |----→ [Data Lake / S3]
        |----→ [Analytics / Dashboards]
        |----→ [Real-time Alerts / SMS system]
```

✅ 3. Example Event Flow

User books ride → fare and status keep updating → ride completes.

App code (microservice):

sql
CopyEdit
```sql
UPDATE rides SET status = 'en_route', driver_location =
'12.93,77.61', fare = 185.0;
```

Debezium emits:

json
CopyEdit
```json
{
  "before": { "status": "confirmed", "fare": 180 },
  "after":  { "status": "en_route", "fare": 185 },
  "op": "u"
}
```

→ Published to Kafka
→ Consumed by: fraud systems, heatmaps, Redis cache, S3 writer

---

✅ 4. Why Debezium Is Ideal Here

- **Zero application changes**: we don't modify any service
- **Complete row-level history**: before & after state
- **Great for analytics + ML**: live streaming + historical data
- **Replayable**: Kafka topics can be reprocessed downstream

✅ 5. When Not to Use Debezium (Complement with Outbox)

> *"For one-time semantic events like `ride_booked` or `payment_successful`, I'd still use Outbox Pattern. But for live sync of all `rides` table changes — CDC with Debezium is ideal."*

✅ Final Statement You Say in Interview:

> *"So to stream all low-level data changes from Uber's `rides` table — including fare, location, and status updates — I'd use Debezium CDC. It gives full auditability, no app change, and works well for analytics and ML. I'd complement this with Outbox Pattern for business events like booking confirmation or payment success."*

EXAMPLE:

1. User books a ride

- A record is inserted into the `rides` table:

sql
CopyEdit
```
INSERT INTO rides (ride_id, user_id, status, fare)
VALUES ('R101', 'U501', 'confirmed', 180);
```

- This booking confirmation **is a business event** → handled by **Outbox Pattern**
  - Used to trigger: Email, SMS, invoice, dispatching

---

2. Debezium takes over for the rest of the ride

Now the actual ride progresses:

---

3. Driver is en route → updates start

Every 10–15 seconds, the `rides` table updates:

sql
CopyEdit
```
UPDATE rides SET driver_location = '12.935,77.615', status =
'en_route', updated_at = now();
```

Debezium (plugged into **Postgres WAL**) sees this:

json
CopyEdit
```json
{
  "before": { "status": "confirmed" },
  "after": { "status": "en_route", "driver_location":
"12.935,77.615" },
  "op": "u"
}
```

Published to Kafka topic: `db.rides`

Consumer systems:

- Driver ETA prediction (ML)
- Live maps / support dashboards
- Heatmaps for surge pricing

---

4. Driver reaches the pickup point

sql
CopyEdit
```sql
UPDATE rides SET status = 'arrived', driver_location =
'12.934,77.611';
```

Debezium publishes another update event to Kafka:

- Dashboard now shows "Driver arrived"
- SMS system gets triggered
- ETA countdown stops

---

5. User boards + starts ride

sql
CopyEdit
```sql
UPDATE rides SET status = 'started', boarding_time = now();
```

Debezium emits:

- ML system logs time to pickup
- Real-time fare calculation begins
- Audit system logs pickup event

---

6. Live fare & location updates (every few seconds)

sql
CopyEdit

```
UPDATE rides SET fare = 192.5, driver_location = '12.932,77.609';
```

Debezium streams all changes **without app code knowing**.

Kafka topic `db.rides` is consumed by:

- Real-time pricing
- Heatmap engines
- Billing estimator
- Support escalation system (delayed ride detection)

---

7. User completes the ride

sql
CopyEdit

```
UPDATE rides SET status = 'completed', fare = 204.5;
```

Debezium emits:

- Final record for analytics
- Triggers downstream ML models (e.g., cancellation prediction, fare fairness)
- Kafka → Data Lake → Delta/Parquet

✅ Key Point:

App only wrote to the DB.
Debezium captured every change passively and streamed it.
No code change, no outbox logic, no Kafka producers inside the app.

Use Outbox when:

- You emit **intentional business events**
- You **control the app**
- You need **precise semantic control** over what's published

Use Debezium CDC when:

- You need to stream **all DB changes**
- You **cannot modify app code**
- You want to feed **analytics, audit logs, or ML pipelines**

Term	Means
App / Application	**Your backend microservice that performs writes to DB**
Modify App Code	**Change its logic to emit events, write to outbox, or produce to Kafka**
Debezium avoids this	**Just monitors DB changes behind the scenes**

Outbox requires this	**App inserts to `outbox_events` + manages `status` like `PENDING`, `SENT`**

Kafka-Based System Design: Quick Checklist (Upstream → Kafka → Downstream)

🔺 1. Upstream: Data Producers

Source Type	Options
Application writes to Kafka directly	Use Kafka producer APIs (Java, Python, etc.)
Application uses Outbox Pattern	App writes to `outbox_events` table + poller or Debezium Outbox Connector publishes to Kafka
Legacy system / Monolith	Use Debezium CDC (reads DB log and publishes row-level changes)

Checklist:

✨ 2. Kafka Layer: Event Transport & Buffer

Kafka acts as the core message bus.

Checklist:

🔻 3. Downstream: Data Consumers / Sinks

Sink Type	Examples	Purpose
Microservices	FraudService, EmailService	Trigger actions based on events
Kafka topic (new)	`enriched.orders`, `alerts`	Chaining pipelines or materializing new views
Stream processors	Flink, Spark	Enrichment, aggregation, windowing, joins
Data Warehouse	BigQuery, Snowflake	Analytics, dashboards, reporting
Data Lake	S3, Delta, Iceberg	ML pipelines, audit, replay
NoSQL DB	MongoDB, Cassandra	Lookup tables, API backends
Relational DB	PostgreSQL, MySQL	Audit logs, history tables
Cache / Search	Redis, ElasticSearch	Dashboards, alerting, real-time search

Checklist:

🚀 Pro Tip: Think Like a Pipeline

> **Upstream (source + ingestion strategy) → Kafka (buffer + format) → Downstream (action, store, or query)**