Indian Premier League

Producer

CSK vs RCB

All updates per ball

{ "match_id": 101, "over": 12.3, "batsman": "Kohli", "runs": 4 }

Consumer System

Consumers

Scoreboard Service
Commentary Engine
ML prediction system
Analytics Service
Notifications

Problems:

Each new service means modifying producer code.
Tight coupling — any consumer failure blocks the producer.
Retry logic becomes messy (what if 1 service is down?).

Scalability Nightmare, No Replay Capability,
No async processing,

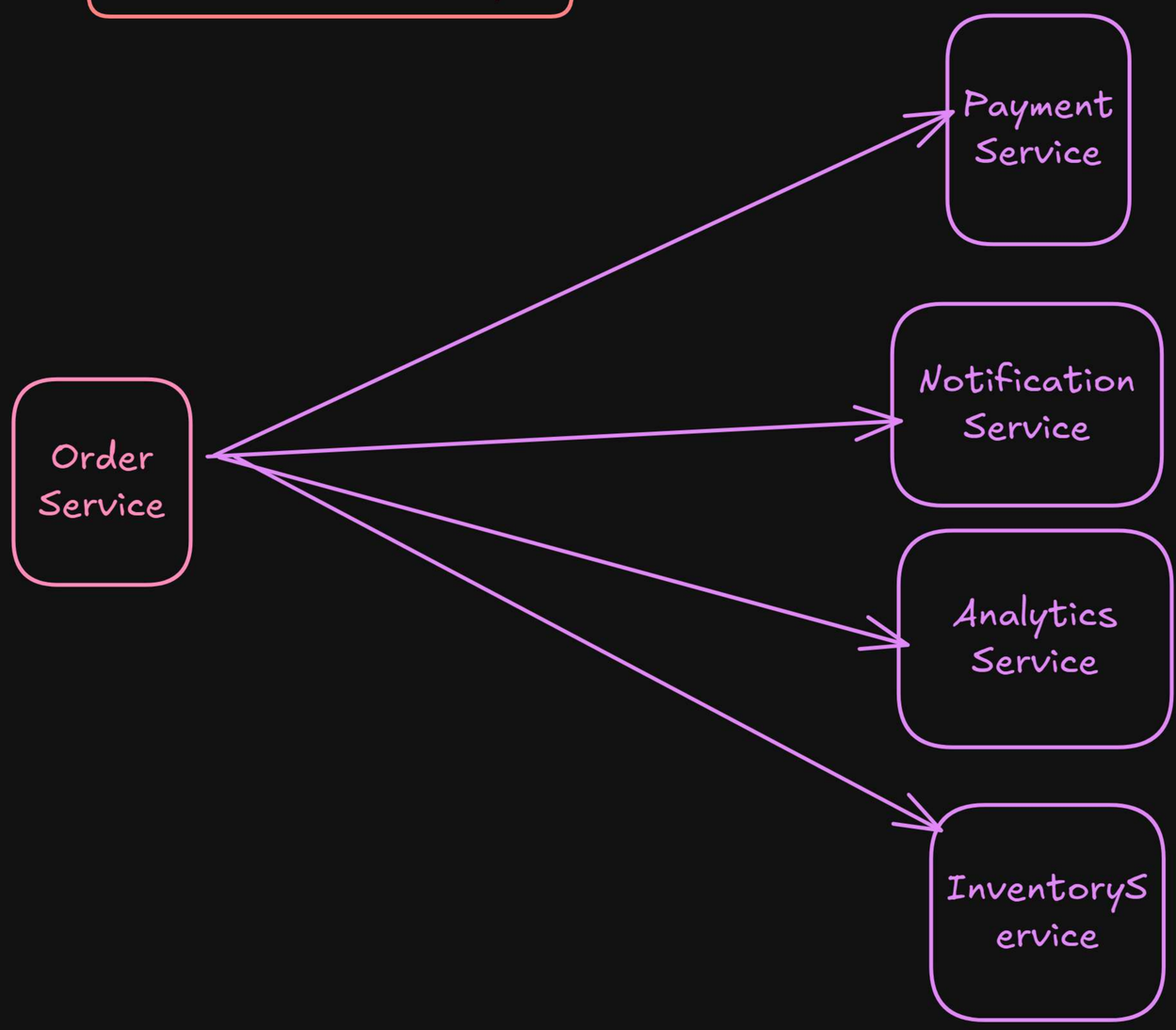
Ordering,
Data loss
Retry failures
Backpressure

-> Polling Delay (Not Real-Time)
-> High Load on Database
-> Ordering Problems
-> Databases handle writes per second in the thousands. Throughput

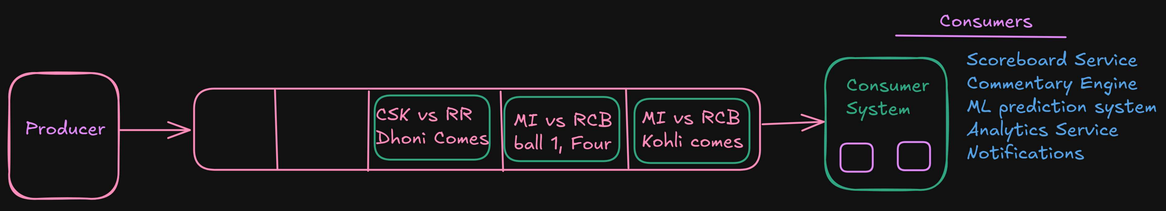
what if millions of writes/sec ?

DB

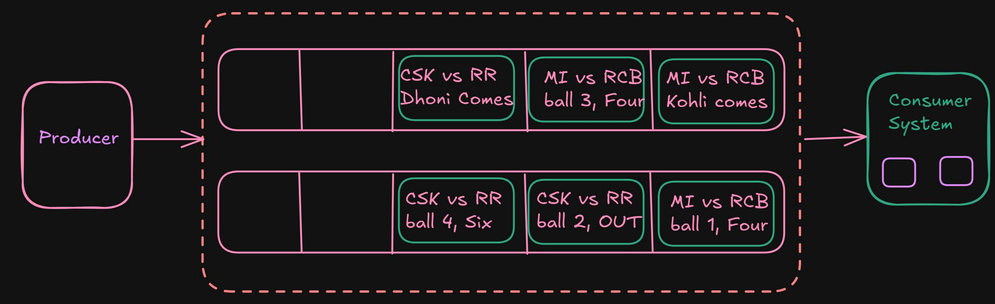
Amazon Order Example



IPL Example — Solution

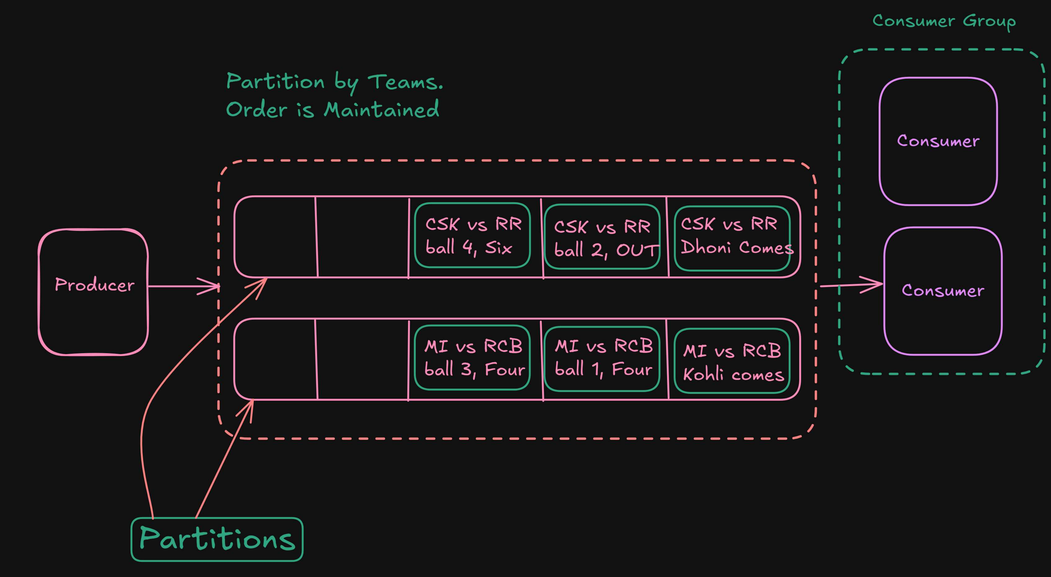


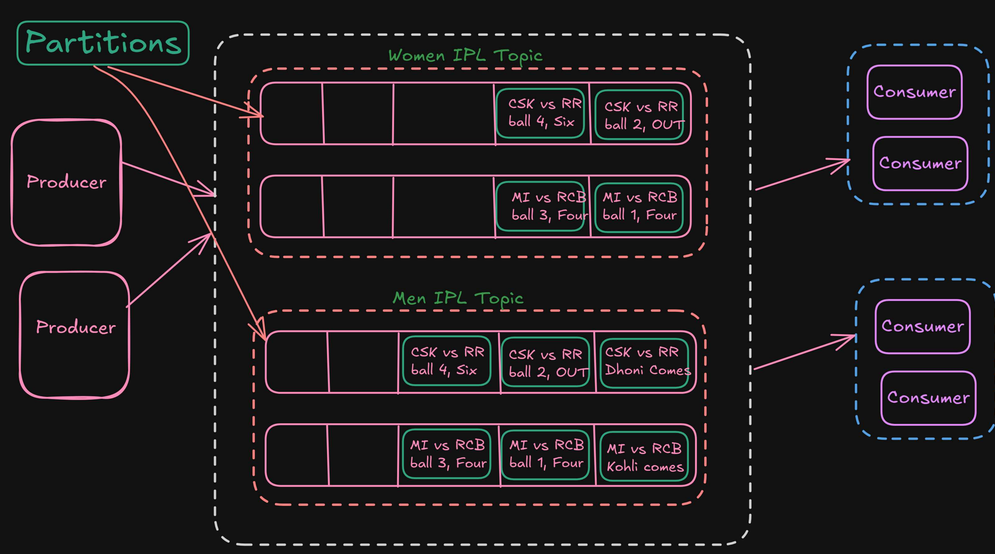
How to scale ?



How to manage Ordering ?

Partition





What is Kafka?

Apache Kafka is an open-source distributed event streaming platform.
It can be used as a message queue, stream processing system

It's designed for building real-time data pipelines and streaming applications —
that means it helps systems send, store, and process data continuously in real time.

Why Kafka?

Before Kafka, systems used point-to-point messaging (like RabbitMQ) or polling-based APIs.
But in large-scale systems like LinkedIn, Uber, Netflix, or Amazon, these patterns break down.

Kafka solves key problems:

Scalability: Handles millions of messages per second.

Durability: Data persisted to disk (and replicated).

Fault tolerance: Survives broker failures.

High throughput + low latency.

Apache Kafka — Core Components

Producer

Sends messages (called events) to Kafka topics.

Each event is a record:

```
{
  "key": "match_id_1",
  "value": "Virat Kohli hits a 4",
  "timestamp": "2025-11-02T18:00:00Z"
}
```

Important producer details:

-> Key determines which partition the message will go to.
Example: key = match_id_1 → ensures all events of the same match go to the same partition (ordering preserved).

-> acks (Acknowledgment levels):

acks=0: Fire and forget.

acks=1: Wait for leader ack.

acks=all: Wait for all replicas — safest.

-> Idempotent producers: Prevent duplicate events.

Topic

A topic is like a folder/category where messages of a specific type are stored.
Basically logical grouping of partitions

Example:

match-events
user-notifications
orders
payments

Each topic is divided into multiple partitions for scalability.

Example:

```
Topic: match-events
├── Partition 0
├── Partition 1
└── Partition 2
```

Partition

Each partition is an ordered, immutable sequence of messages.
Messages are continuously appended like a log file

Each message in a partition gets an offset — a unique incremental ID.

Ordering is guaranteed only within a partition, not across the topic.

Parallelism: Consumers read different partitions in parallel.

Broker

A Kafka Broker is a server that stores data and handles producer/consumer requests.

Kafka cluster = multiple brokers.

Example:

Broker 1 → Partitions 0, 3
Broker 2 → Partitions 1, 4
Broker 3 → Partitions 2, 5

Each partition has:

Leader replica: Handles all reads/writes.

Follower replicas: Copy data from leader.

If leader fails → one follower becomes leader (automatic failover).

Each partition can be replicated across multiple brokers

Consumer

It pulls messages from Kafka (not pushed).
You can control offset (where to start reading):

auto offset (latest or earliest)

manual commit (for precise control)

Example:

Consumer reads from:

match-events topic
→ Partition 0 (offset 1000 to 1200)

Consumer Group

A Consumer Group is a set of consumers working together on a topic.

Kafka guarantees that each partition is consumed by only one consumer in a group.

Relationship Overview

Relationship	Meaning
Topic → Partitions	One topic has multiple partitions (for parallelism)
Partition → Consumer (within one group)	One partition is consumed by exactly one consumer in a consumer group
Partition → Consumers (across different groups)	consumer groups can read the same partition independently
Consumer Group → Partitions	A group collectively consumes multiple partitions
Consumer → Partition	One consumer can read from multiple partitions

Offset

Each message in a partition has an offset — a monotonically increasing number.

Consumers track offsets to know where they left off.

Replication

To ensure durability, each partition has replicas (leader + followers).

Example:

Replication factor = 3

Broker 1 → Leader (Partition 0)
Broker 2 → Follower (Partition 0)
Broker 3 → Follower (Partition 0)

If Broker 1 dies → Kafka promotes one follower to leader.

Delivery Semantics

Kafka provides three delivery guarantees:

At most once: Messages may be lost but never redelivered.

At least once: No loss, but duplicates possible.

Exactly once: No loss, no duplicates (requires idempotent producers + transactional writes).

Kafka — Message Flow Lifecycle (Full End-to-End Journey)

Producer → Broker (Leader Partition) → Replication → Consumer Group → Consumer

1. Producer Sends a Message (called a record)

```
{
  "topic": "match-events",
  "key": "match_id_101",
  "value": "Virat Kohli hits a SIX!",
  "timestamp": "2025-11-02T18:00:00Z"
}
```

2. Partition Selection (Where to Store It)

Each Kafka topic has multiple partitions.
Kafka decides which partition gets the message based on key.

Partitioning Logic:

Case	Strategy	Description
Key present	hash(key) % num_partitions	All same keys → same partition (ordering preserved)
Key absent	Round robin	Spread evenly across partitions

Example:

hash("match_id_101") % 3 = 1 → Partition 1

Ordering guaranteed only within the same partition.

3. Broker Assignment

Once the partition is determined, Kafka identifies which broker holds that partition.

The mapping of partitions → brokers is managed by Kafka cluster metadata,
which is maintained by the Kafka controller (a role within the broker cluster).

The producer uses this metadata to send the message directly to the
broker that hosts the leader of the target partition.

Earlier:
Kafka used ZooKeeper to store this cluster metadata and help elect the controller.

Now (Modern Kafka):
Kafka uses its internal KRaft (Kafka Raft) system — no ZooKeeper needed.
KRaft manages all metadata and leader elections inside Kafka itself, using the Raft consensus protocol.

4. Broker (Leader Partition) Writes the Message

Each partition has:

Leader Replica → handles all reads/writes
Follower Replicas → copy data from leader

Steps:

Producer sends the message to the leader broker.

Broker appends it to the partition's commit log.

The message is given an offset (unique ID).

The broker acknowledges the producer based
on acks configuration.

acks=0 No confirmation (fastest, risky)
acks=1 Leader only confirms
acks=all All in-sync replicas confirm (safest)

5. Replication

6. Consumers Read Messages

Consumers belong to a Consumer Group.
Each group reads from the topic's partitions,
and each partition is read by exactly one consumer
in that group.

Steps:

Consumer sends Fetch Request to the
broker (leader of the partition).

Broker returns messages (batches).

Consumer processes them (e.g., updates database, UI, etc.).

Consumer commits offset after processing.

7. Offset Management

Each message in a Kafka partition is assigned a unique offset,
which is a sequential identifier indicating the message's position within that partition.
Offsets are monotonically increasing and start from 0.

This offset is what consumers use to track their progress while reading messages.

As consumers read messages, they maintain their current offset.

They periodically commit this offset back to
Kafka (to the internal topic __consumer_offsets).

If a consumer fails or restarts, it can resume reading from the
last committed offset, instead of starting from the beginning.

Producer
↓
Partition Determination
↓
Broker Assignment (ZooKeeper old / KRaft new)
↓
Leader writes message
↓
Followers replicate
↓
Consumers fetch
↓
Offsets committed

Fault-tolerant
Ordered within partition
Real-time & scalable

When to Use Kafka

Apache Kafka can serve two major purposes — it can act as a message queue or as a streaming platform.
The difference depends on how data is consumed by your services:

In a message queue, consumers pull messages when they're ready, one at a time
— like filling a bucket from a tap, where you decide when to collect water.

In a streaming model, data flows continuously, and consumers process it in real time
— much like standing under a waterfall, where water keeps coming nonstop.

Kafka as a Message Queue

Kafka behaves like a traditional message queue when you want
to handle asynchronous tasks, preserve event ordering, or
decouple producers and consumers.

Unlike RabbitMQ, Kafka is built for very high throughput
and massive scalability.

If your system needs to handle hundreds of thousands
or millions of messages per second, Kafka is the right fit.

For smaller workloads or low-latency, simple background tasks,
RabbitMQ or SQS might be lighter and easier to manage.

Use Kafka as a Message Queue When:

1. You have tasks that can be processed asynchronously

Example: Amazon Order Processing System
When a user places an order, you confirm it instantly, but publish an event to a Kafka topic for background work —
such as sending an email, charging the card, or updating inventory.
These tasks are handled asynchronously by different consumers.

2. You need to process events in strict order

Example: Cricket Ball-by-Ball Feed
Each ball event — "Ball 1: Dot", "Ball 2: FOUR", "Ball 3: OUT" —
must be processed in the exact sequence they occur for that match.
Kafka ensures ordering within a partition, so all events for a given Match ID
are processed sequentially and correctly.

3. You want to decouple producers and consumers

Example: E-commerce Microservices
The Order Service may produce events much faster than the
Shipping Service can handle them.
Kafka acts as a buffer between the two, letting
each service scale independently,
preventing one slow consumer from affecting the rest of the system.

Throughput Note:

For very high throughput or scalable,
distributed processing, use Kafka.

For lightweight, low-volume,
low-latency workloads,
a simple queue like RabbitMQ
or SQS might be better suited.

Kafka as a Streaming Platform

Kafka shines as a real-time data streaming platform
when you need to process information continuously
as it arrives.

Use Kafka as a Stream When:

1. You need continuous, real-time processing

Example: Ride-sharing Platform (like Uber)
Drivers send location updates every few seconds.
Kafka Streams or Flink can consume these events to
calculate ETAs, match riders with nearby drivers,
and adjust surge pricing — all in real time.

2. Multiple consumers need the same data simultaneously

Example: Online Food Delivery App (like Swiggy/Zomato)
Every "order status update" — food prepared,
out for delivery, delivered —
is published to a Kafka topic.
Multiple systems — the customer app, restaurant dashboard,
and delivery tracker — can all consume this
same stream of updates independently and simultaneously.

Interview Tip

When asked "When would you use Kafka?", you can answer:

Kafka can act as both a message queue
and a streaming system.

As a queue, it's great for asynchronous
or ordered processing — like handling Amazon orders
or ball-by-ball cricket events
— especially when you need high throughput.

As a stream, it's ideal for real-time
use cases — like tracking drivers or
broadcasting live order updates to multiple systems.

Scalability

Let's first understand how Kafka scales.

A Kafka broker is just one server in a Kafka cluster that stores data and handles reads/writes.

Now, Kafka doesn't have a fixed message size limit — you can change it using message.max.bytes.
But for best performance, keep your messages small (under 1 MB).
Small messages → faster processing → less memory pressure → better network speed.

❌ Don't store big files or videos in Kafka.
✅ Store large data (like videos or PDFs) in storage systems such as S3, and send only a small Kafka message with the file's location.

Rough Capacity Estimate

On decent hardware:

One broker can store about 1 TB of data.

It can handle roughly 1 million messages per second (depending on hardware and message size).

If your total load fits within that, you may not even need to worry about scaling.

When You Need to Scale

If one broker isn't enough, there are two main ways to scale Kafka:

1. Add More Brokers (Horizontal Scaling)

2. Smart Partitioning Strategy

Scaling in Kafka mostly depends on how you partition your data.

Each message goes to a partition based on its key:

```
partition = hash(key) % number_of_partitions
```

Good key choice = even load
Bad key choice = "hot partitions" (one partition gets all the traffic)

Example: If you partition by ad_id and one ad goes viral, that single partition becomes overloaded.
That's called a hot partition.

How to Handle Hot Partitions

1. Compound Keys (Best Practice)

Instead of using a single key like ad_id, combine it with another field — for example: ad_id + region or ad_id + user_id.
This spreads messages more evenly while still keeping them logically grouped.
(e.g., traffic for different regions or users goes to different partitions).

2. Salting the Key

Add a random number or timestamp to your key, like "ad_id_01", "ad_id_02".
This breaks heavy traffic for one key into multiple partitions.
It's simple and effective, though it can make aggregation logic more complex later.

3. Random Partitioning (No Key)

If message ordering isn't important, send messages without a key.
Kafka will automatically assign partitions in a round-robin fashion.
This guarantees even distribution across partitions — but you lose order within a topic.

4. Backpressure (Last Resort)

If producers are overwhelming Kafka, slow them down temporarily.
Either limit the producer's send rate, or let managed services (AWS MSK) throttle automatically.
This gives brokers time to recover and prevents message lag buildup.

Fault Tolerance & Durability

Kafka is designed to never lose data.
It does this through replication.

Each partition is copied across multiple brokers:

One broker = Leader

Others = Followers

When a producer sends a message:

It's written to the leader.

The leader replicates it to all followers.

Once all replicas confirm, the message is marked as "safe".

This depends on the producer setting acks

When a Consumer Fails

Kafka is usually "always available," but consumers can fail.
Here's what happens then:

Offset Management:

Each message in a partition has an offset (like a line number).
After processing, the consumer commits
its latest offset to Kafka.

If it crashes, it restarts from that
saved offset — no data loss or duplication.

Rebalancing:

In Kafka, multiple consumers can share the work by forming a consumer group.
Each consumer in the group reads messages from a different set of partitions.

If one consumer stops working — maybe it crashes,
disconnects, or shuts down —
Kafka automatically detects this and reassigns
that consumer's partitions to the others in the group.

This way, the remaining consumers pick up the extra work,
and message processing continues smoothly
without any data being left behind.

Handling Retries & Errors

Producer Retries

Sometimes messages fail to reach Kafka
(due to network or broker issues).
Kafka supports automatic retries.

```
const producer = kafka.producer({
  retry: { retries: 5, initialRetryTime: 100 },
  idempotent: true
});
```

idempotent: true ensures no duplicates even during retries.

Consumer Retries

Kafka doesn't have built-in retries for consumers.
You can build your own system using:

A retry topic → to reprocess failed messages later.

A Dead Letter Queue (DLQ) →
to store permanently failed messages for investigation.

Performance Tips

Batch messages

Compress data

Choose balanced partition keys

Retention

Kafka doesn't store messages permanently —
it keeps them only for a limited time or until a certain amount of data is reached.

You can control this using two settings:

retention.ms → how long messages should be kept (time-based)

retention.bytes → how much data can be stored before older messages are deleted (size-based)

By default, Kafka keeps messages for 7 days.
You can increase this period if your system needs older data,
but remember — the longer you retain messages,
the more storage space and disk cost you'll need

DEMO