Real-Time Systems using Redis & Kafka

Sandeep Kumar (CS24M112)

Ashant Kumar (CS24M113)

Dharmendra Chauhan (CS24M115)

Seminar: CS639G

Department of Computer Science and Engineering

Indian Institute of Technology Tirupati

Instructor: **Dr. G Ramakrishna**

November 13, 2025

Outline

Introduction

- Modern applications demand **real-time communication** — from order tracking and chat systems to live dashboards.
- Traditional request–response systems often fail to handle high data velocity and low-latency requirements.
- To overcome this, modern architectures use:
  - **Apache Kafka** – for reliable, high-throughput event streaming between backend services.
  - **Redis** – for fast, in-memory data updates and live broadcasting to users.
  - **SSE / WebSocket** – for delivering instant updates to the frontend.
- Together, these technologies form the foundation of real-time, event-driven systems.

Apache Kafka

A Distributed Event Streaming Platform

Why Apache Kafka ?

- Modern systems like Zomato, Uber, and Discord generate massive streams of real-time data.
- Traditional databases cannot handle such high-speed read/write operations.
- Every few seconds, thousands of updates (e.g., driver location, chat messages) overload the database.
- We need a system that can:
  - Handle millions of messages per second.
  - Stream data to multiple services at once.
- **Solution: Apache Kafka – built for real-time, distributed event streaming.**

What is Apache Kafka?

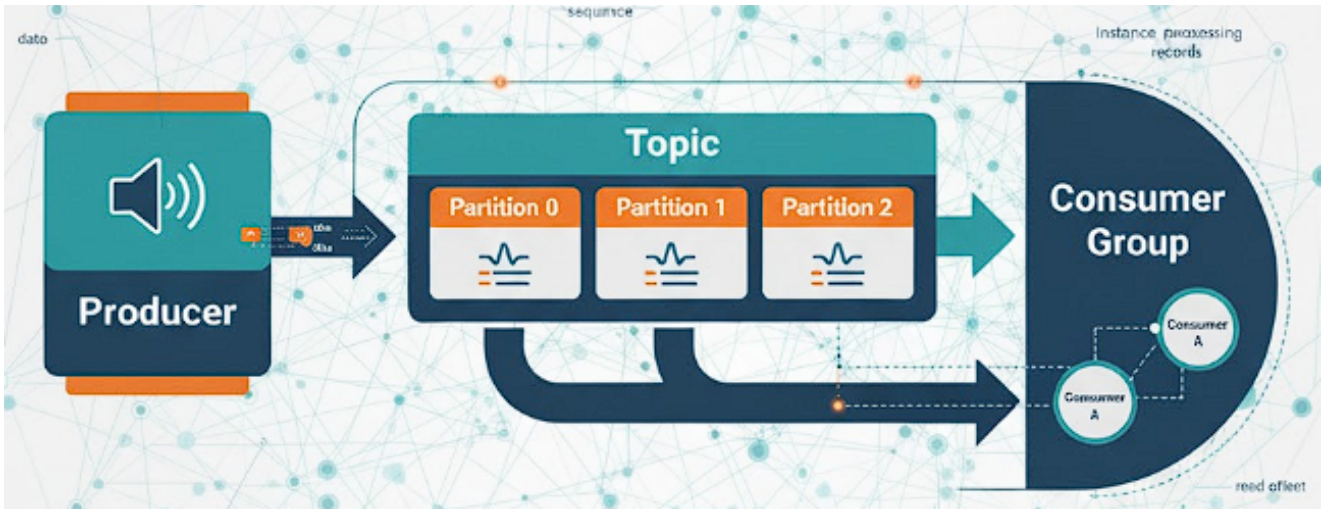
- **Apache Kafka** is an open-source, distributed event streaming platform.
- Originally developed by **LinkedIn**, now part of the Apache Software Foundation.
- Works as a **message broker** that decouples producers and consumers.
- Designed for high throughput, reliability, and scalability.

Why Use Kafka?

- **High Throughput:** Handles millions of events per second efficiently.
- **Durable:** Data persisted to disk for a configurable period.
- **Scalable:** Easy to add brokers, partitions, or consumers.
- **Fault-Tolerant:** Data replicated across brokers for reliability.
- **Decoupled:** Producers and consumers can work independently.

Kafka Architecture Overview

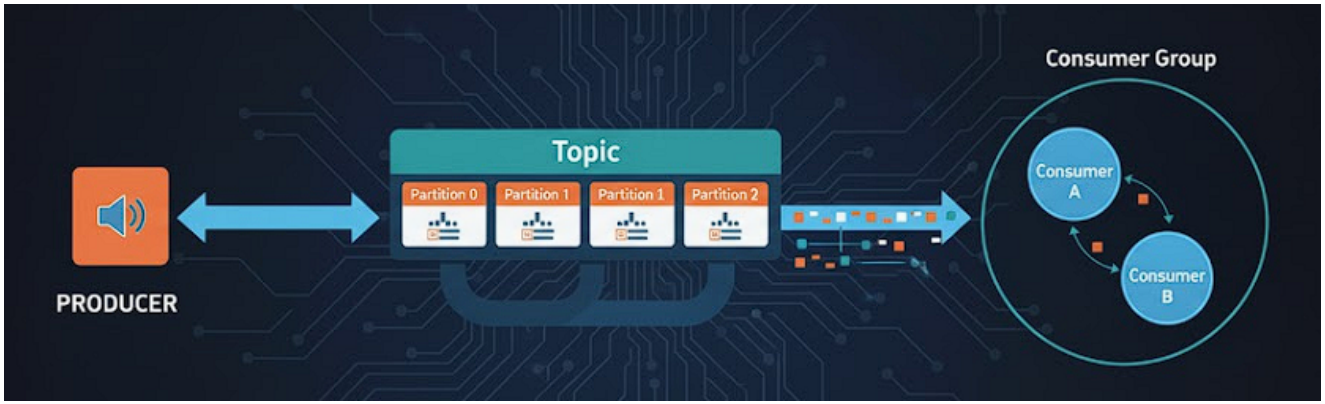
- **Producer:** Sends real-time data (messages) to Kafka topics.
- **Broker:** Kafka server that stores and manages message streams.
- **Consumer:** Reads and processes messages from topics.
- **ZooKeeper:** Coordinates brokers and manages cluster metadata.



Producer → Topic → Partition → Consumer Group

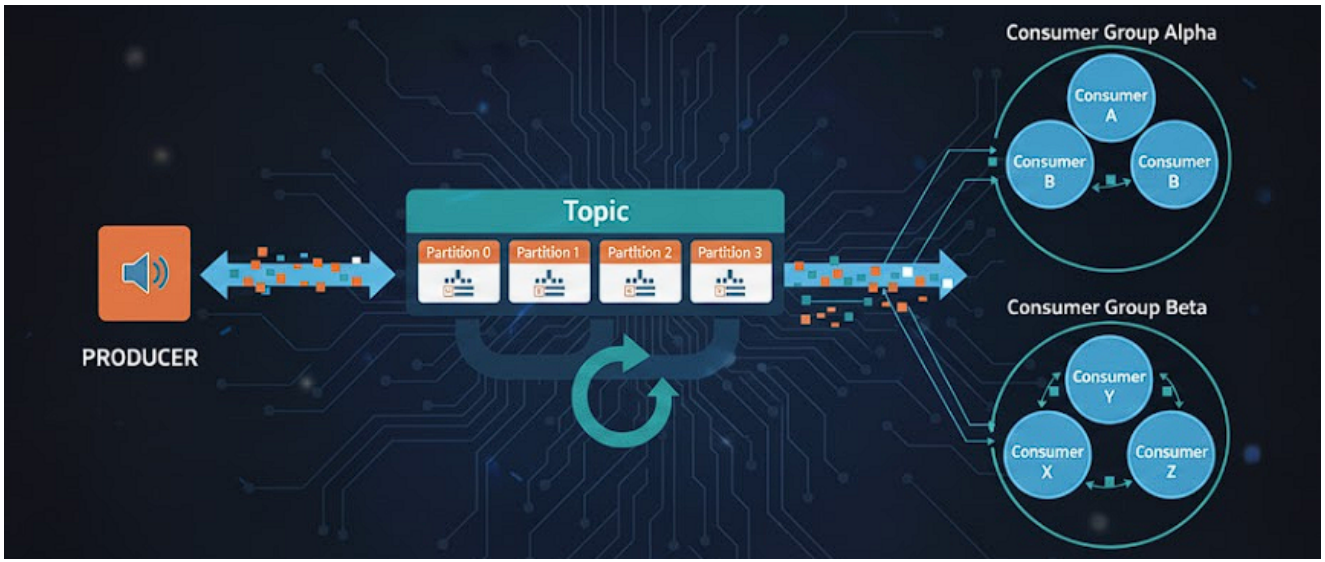
Topics and Partitions

- **Topic:** Logical stream of messages (e.g., "OrderUpdates").
- **Partition:** Physical split of a topic for parallel processing.
- Each message has an ordered ID called an **offset**.
- Enables high throughput and load balancing.
- Example:
  - Partition 0 → North India data
  - Partition 1 → South India data



Consumer Groups and Balancing

- Kafka automatically assigns partitions to consumers in a group.
- **Rule:** One partition → one consumer (within the same group).
- Examples:
  - 1 Consumer → 4 Partitions → handles all.
  - 2 Consumers → 4 Partitions → 2 each.
  - 4 Consumers → 4 Partitions → 1 each.
  - 5 Consumers → 4 Partitions → one idle.
- This ensures parallelism and fault-tolerant workload distribution.

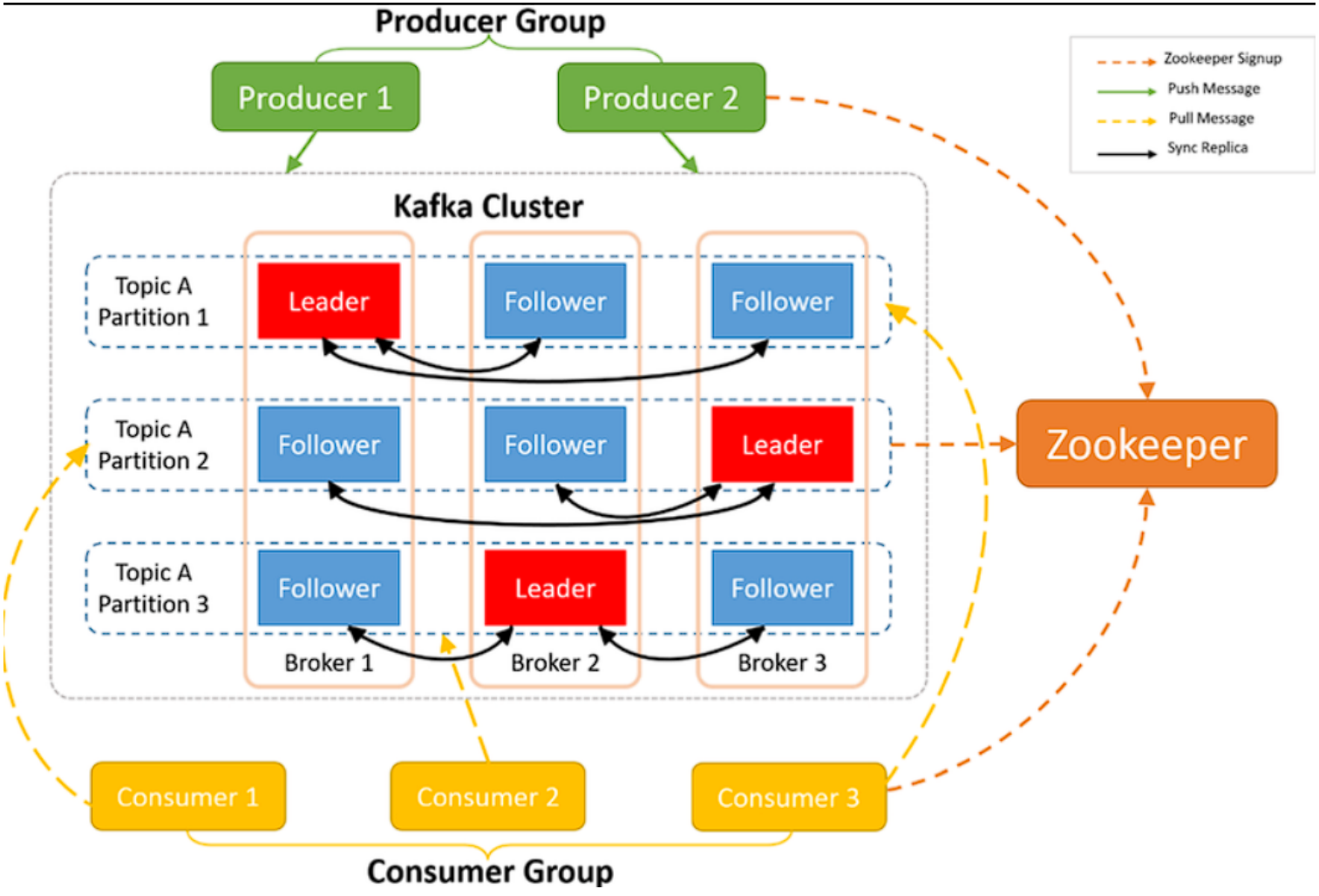


Queue vs. Publish–Subscribe Model

- **Queue Model (FIFO):**
  - One consumer group.
  - Each message processed only once.
- **Publish/Subscribe Model:**
  - Multiple consumer groups.
  - Each group receives a copy of the same stream.
- Kafka supports both models at the same time.
- Makes it flexible for different system designs.

Kafka and ZooKeeper – Legacy Coordination

- Many versions of Apache Kafka relied on **ZooKeeper** for managing cluster state and metadata.
- ZooKeeper tracked:
  - Brokers, topics, and partition configurations.
  - Controller elections and leader assignments.
- Each topic was divided into multiple partitions distributed across brokers.
- This design provided fault tolerance and scalability but added operational complexity.
- Managing and synchronizing data through ZooKeeper often caused delays and coordination overhead.



Kafka Architecture with ZooKeeper – external coordination of brokers, topics, and leaders.

How Kafka Works – Step 1: Data Flow from Producer to Broker

- **1. Producer:**
  - Applications (e.g., order service, tracking service) act as producers.
  - They publish messages such as order updates or sensor data to a **Kafka topic**.
- **2. Broker:**
  - Kafka brokers receive and store these messages.
  - Each topic is divided into **partitions**, which distribute data across multiple brokers.
  - Partitions ensure parallelism and scalability for high message throughput.
- **3. Replication:**
  - Each partition is replicated to other brokers for fault tolerance.
  - One broker acts as the **leader**, while others are **followers**.

How Kafka Works – Step 2: Data Consumption

- **4. Consumer:**
  - Consumers subscribe to one or more topics.
  - Each consumer reads data from assigned partitions independently.
  - Multiple consumers can work together as a **Consumer Group**.
- **6. Real-Time Processing:**
  - Data consumed can be processed, stored in a database, or pushed to Redis or dashboards.
  - The flow is continuous — enabling real-time analytics and event-driven applications.

Advantages of Using Kafka

- **Scalable:** Efficiently handles data at any scale.
- **Reliable:** Guarantees message delivery.
- **High Performance:** Very low latency even under heavy load.
- **Flexible:** Works with multiple producers and consumers.
- Ideal for:
  - Real-time analytics and dashboards
  - Order tracking systems
  - IoT data streaming
  - Centralized log aggregation

Summary

- Apache Kafka enables real-time, fault-tolerant data streaming.
- It overcomes database throughput limitations in distributed systems.
- Core concepts: Topics, Partitions, Brokers, and Consumer Groups.
- Principle: **Decouple producers and consumers for scalability.**
- Modern Kafka (KRaft) simplifies management and boosts performance.

Redis

In-Memory Data Store for Real-Time Applications

What is Redis?

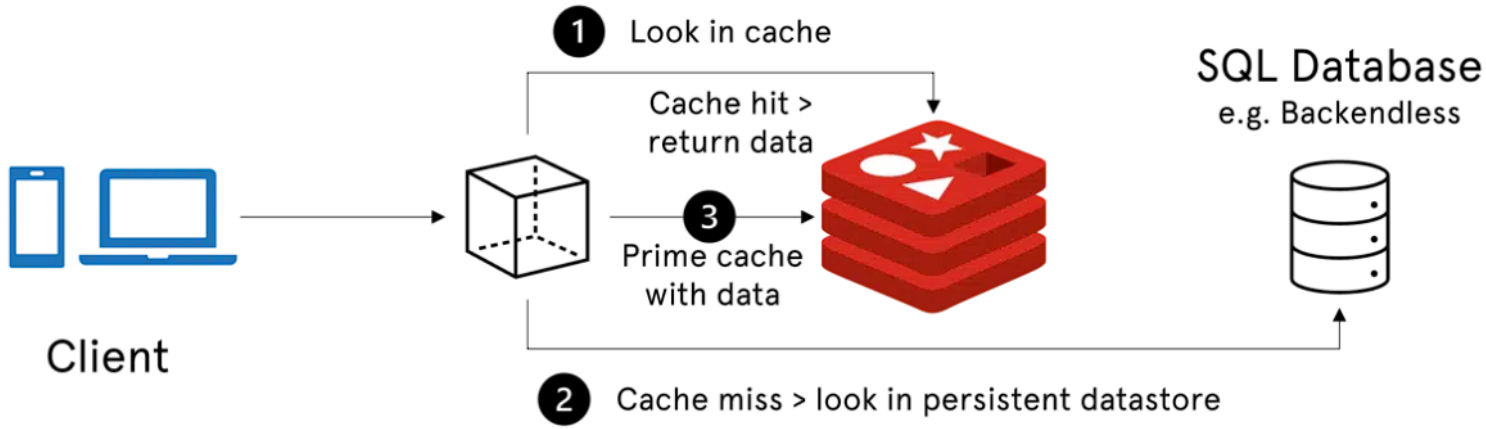
- **Redis (Remote Dictionary Server)** is an open-source, in-memory data store.
- Supports data types like **strings, lists, sets, hashes, streams, and sorted sets**.
- Used as:
  - A **cache** to speed up database queries.
  - A **message broker** for Pub/Sub systems.
  - A **real-time data store** for dashboards and analytics.
- Known for its **microsecond latency** and extremely high performance.

Why Use Redis?

- **Speed:** All data is stored in-memory, making reads/writes extremely fast.
- **Versatility:** Works as cache, queue, and Pub/Sub system.
- **Persistence:** Offers snapshot (RDB) and Append-Only File (AOF) backups.
- **Scalable:** Supports clustering and replication for high availability.
- **Lightweight:** Ideal for microservice architectures and real-time systems.

Redis Architecture Overview

- **Client:** Application that connects and performs read/write operations.
- **Redis Server:** Stores all data in memory for ultra-fast access.
- **Persistence:**
  - **RDB (Snapshot):** Periodic memory dump to disk.
  - **AOF (Append-Only File):** Logs every operation for durability.
- **Replication:** Supports master–replica setup for failover and scalability.



Redis Data Structures

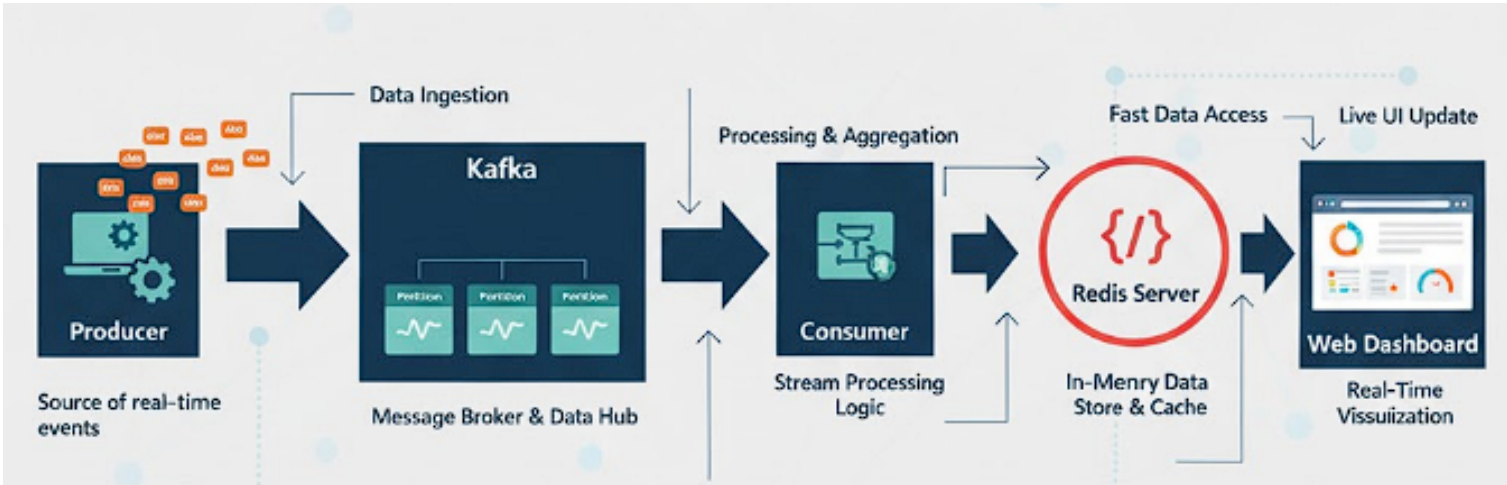
- **String:** Basic key-value pair (e.g., user token, counter).
- **List:** Ordered sequence, ideal for queues.
- **Set:** Unordered unique elements.
- **Hash:** Store key–value pairs (e.g., user profiles).
- **Stream:** Append-only log for real-time event tracking.

Redis Pub/Sub Model

- **Publisher:** Sends messages to a specific **channel**.
- **Subscriber:** Listens to one or more channels.
- **Redis Server:** Instantly delivers messages from publishers to all subscribers.
- Ideal for:
  - Real-time notifications.
  - Chat systems.
  - Live dashboards or monitoring tools.

How Redis Complements Kafka

- **Kafka:** Manages event streaming between backend services.
- **Redis:** Serves as a real-time cache and Pub/Sub system for frontend updates.
- Combined Workflow:
  1. Kafka consumer processes events and stores latest data in Redis.
  2. Redis broadcasts these updates to all connected clients.
  3. Frontend (via WebSocket or SSE) shows live updates instantly.



Kafka → Redis → WebSocket → Frontend = Full Real-Time Pipeline.

Advantages of Using Redis

- **Ultra-fast:** Operates entirely in memory.
- **Reliable:** Optional disk persistence with RDB or AOF.
- **Simple:** Easy to integrate with Python, Node.js, or Java.
- **Scalable:** Cluster support for distributed caching.
- Common Use Cases:
  - Real-time dashboards.
  - Leaderboards and counters.
  - Caching frequent database queries.
  - Pub/Sub messaging and streaming.

Without Redis Caching

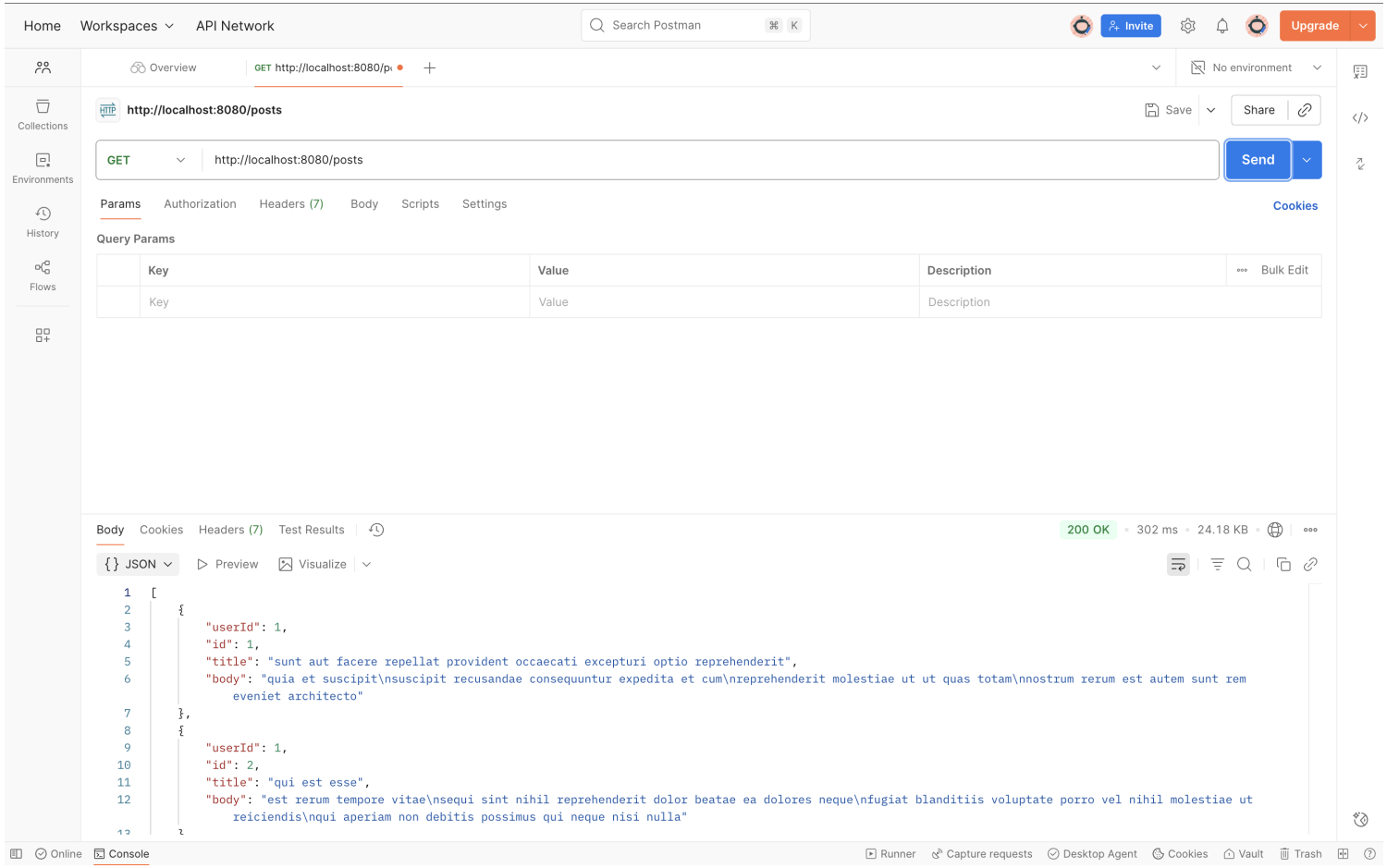


Figure: Every request hits the DB → higher latency.

With Redis Caching

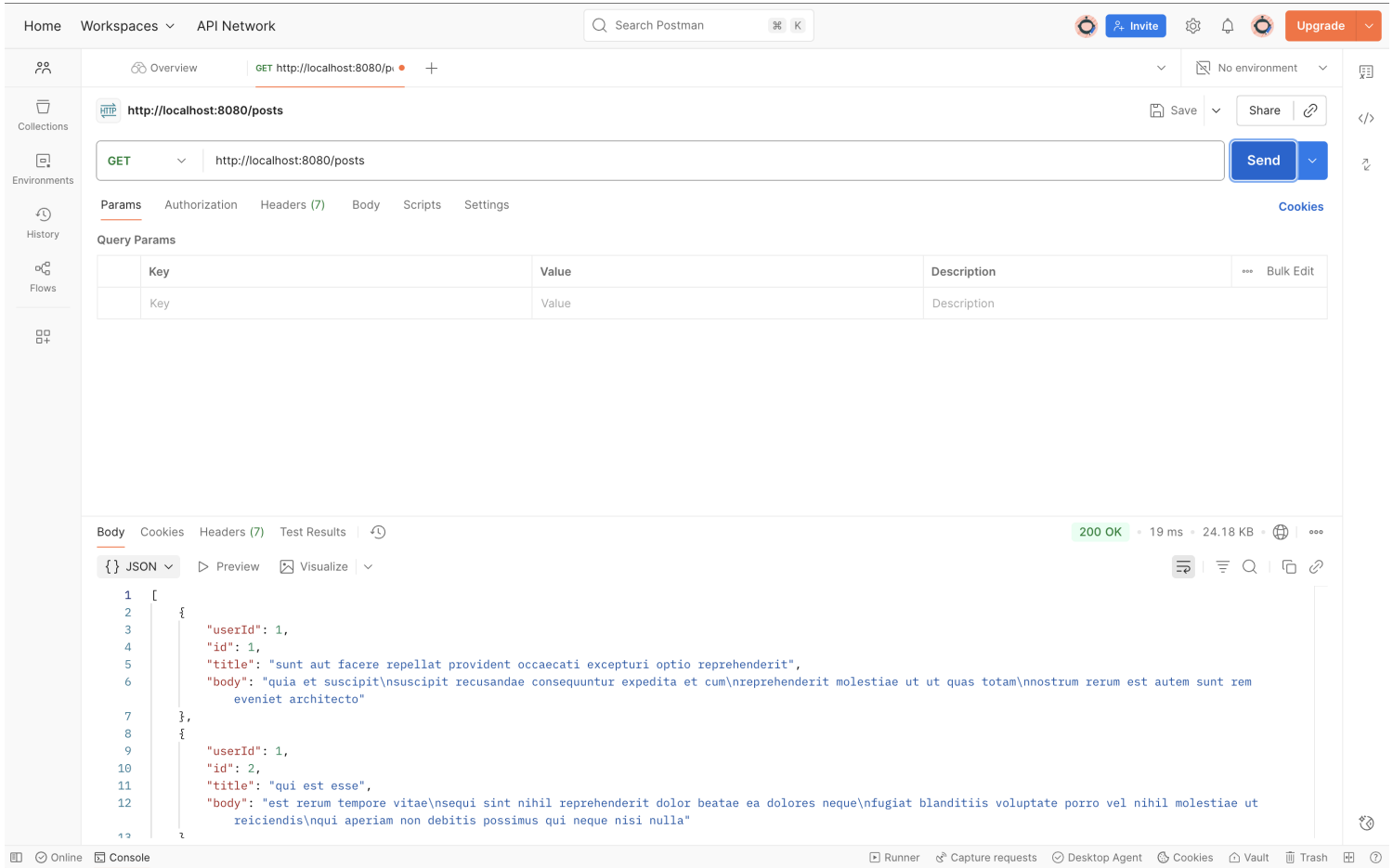


Figure: Data served from Redis → lower latency, reduced DB load.

Summary

- Redis is a high-speed, in-memory data store ideal for real-time systems.
- It can act as a cache, message broker, or real-time event store.
- Complements Kafka by pushing live data to the frontend.
- Together, Kafka and Redis form the backbone of real-time analytics and monitoring systems.

Real-Time Communication Technologies

Long Polling, SSE, Web Sockets

Why Real-Time Communication?

- Real-time communication allows servers to send updates instantly to connected clients.
- **Use Cases:**
  - Real-Time Dashboards (e.g., analytics, KPIs, IoT metrics)
  - Live Chats, Order Tracking, Notifications
  - Trading Platforms (stocks, crypto)
  - Fantasy Leagues & Live Sports Updates
- Core Goal: Deliver data instantly without refreshing the page.

Long Polling – The Old but Reliable Technique

- **Concept:** Client repeatedly requests data; server holds the request open until data is available.
- When server responds, client immediately re-sends another request.

Advantages:

- Simple REST-based approach; easy to implement.
- Works across all browsers and proxies.

Disadvantages:

- High latency (server waits before responding).
- Wastes resources due to repeated requests.

Long Polling simulates real-time updates using traditional HTTP.

Server-Sent Events (SSE) – Simpler Real-Time Push

- **Concept:** Client establishes one persistent connection; server pushes updates over time.
- Unidirectional – from Server → Client only.
- Works on HTTP/1.1; supports auto-reconnect natively.

Advantages:

- Lightweight and efficient.
- Simple to use; built-in browser support (EventSource API).
- Reliable through proxies.

Disadvantages:

- One-way communication (server → client only).
- Not suitable for bidirectional applications like chat.

Ideal for live dashboards and order tracking updates.

WebSockets – Full-Duplex Real-Time Channel

- **Concept:** Creates a single, persistent TCP connection after an HTTP handshake.
- Enables full-duplex (bi-directional) communication.

Advantages:

- True real-time, instant updates both ways.
- Best suited for chats, multiplayer games, notifications.
- Highly scalable when used with brokers (e.g., Kafka, Redis).

Disadvantages:

- Complex to manage connections and events.
- Some proxies/firewalls block WebSocket traffic.

The most powerful but complex real-time technology.

Kafka vs WebSocket vs SSE

Comparing Real-Time Communication Technologies

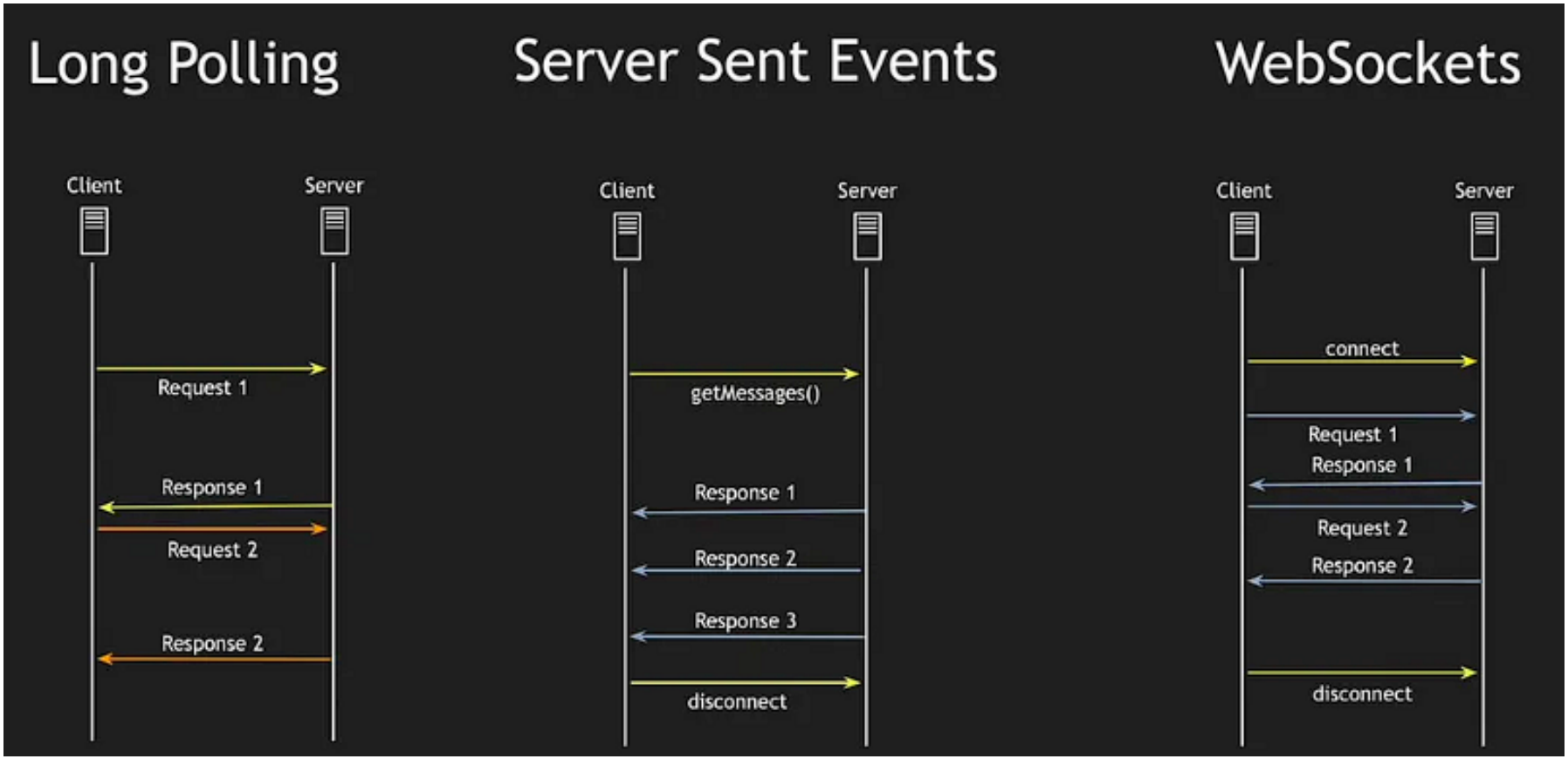
Technology	Layer	Purpose	Who Uses It
Kafka	**Backend-to-Backend**	Distributed event streaming for reliable, asynchronous message delivery between microservices.	Microservice systems, backend pipelines
WebSocket / SSE	**Backend-to-Frontend**	Pushes live updates to browsers – WebSocket for bi-directional, SSE for one-way communication.	Frontend applications, dashboards, live chats

Kafka powers backend data flow; WebSocket/SSE bring it to the UI.

Choosing the Right Technology

- **Long Polling:** Legacy fallback; works everywhere but inefficient.
- **SSE:** Simple, one-way updates (ideal for dashboards).
- **WebSockets:** True real-time, bidirectional, best for chats and games.
- **Kafka:** Backend event streaming; integrates with WebSockets or SSE for UI delivery.

Comparison of Real-Time Communication Protocols



Long Polling vs SSE vs WebSockets — visual comparison of client–server communication models.

Summary

- Real-Time Communication is essential for responsive, data-driven applications.
- **Kafka** handles reliable event delivery between services.
- **WebSockets/SSE** deliver updates to the user interface.
- **Long Polling** remains a fallback when modern methods fail.
- Together, these technologies form the backbone of modern, real-time systems.

Problem Statement - 1

Real-Time Order Tracking System

using Apache Kafka and Redis

Problem Statement

- Traditional order tracking systems provide only periodic or static updates.
- These systems lack **scalability** and **instant synchronization** between admin and users.
- There is a need for a **real-time, event-driven architecture** capable of handling:
  - Multiple concurrent users.
  - Fast propagation of status and location updates.

Goal: Build a system that updates all users instantly with minimal latency.

Motivation

- **Why use Redis & Kafka for this system?**
  - Redis provides ultra-fast, in-memory data access and caching.
  - Kafka enables scalable, asynchronous event streaming.
- **Real-Time User Expectations**
  - Platforms like Swiggy, Amazon, and Zomato rely on instant updates.
  - Users expect delivery tracking to be accurate and live.
- **System Requirements**
  - Handle thousands of concurrent updates efficiently.
  - Support a robust, event-driven communication pipeline.

Introduction

- **What is Real-Time Order Tracking?**
  - A continuous monitoring process that updates order status and location instantly.
- **Importance in Logistics & E-Commerce**
  - Ensures transparency and customer trust.
  - Minimizes delivery uncertainty and support overhead.
- **Key Goals of the Project**
  - Provide **live status updates** for each order.
  - Enable **live location visualization** on an interactive map.
  - Maintain **reliable event streaming** with high consistency.

Project Objectives

- **Implement a Kafka-based event pipeline**
  - Kafka handles the continuous stream of order and location updates between system components.
- **Store the latest data in Redis (in-memory)**
  - Redis maintains real-time order status and location data for quick retrieval and low latency.
- **Display live map and order status on the client side**
  - React + Leaflet visualize order movement and current delivery positions dynamically.
- **Enable instant updates to multiple users simultaneously**
  - SSE broadcasting ensures that all connected clients receive updates in real time.

Objective: Deliver a scalable, low-latency, event-driven order tracking solution.

System Overview

- The project implements a real-time order tracking architecture using Redis and Kafka.
- **Major Components:**
  - **Event Producer:** Sends new order and location updates.
  - **Kafka Broker:** Manages event topics and distributes updates to consumers.
  - **Redis Cache:** Stores the latest status and location data in memory.
  - **WebSocket Server:** Pushes instant updates to all connected clients.
  - **Frontend (React + Leaflet Map):** Displays live order status and map visualization.

This pipeline ensures low-latency and reliable real-time order updates across all users.

Technologies Used

- **Apache Kafka** – Event streaming backbone for handling asynchronous updates.
- **Redis** – High-speed, in-memory data store for storing the latest order and location state.
- **Node.js + Express.js** – Backend server managing Kafka producers, consumers, and APIs.
- **React.js** – Frontend application for user dashboards and customer tracking views.
- **Leaflet** – OpenStreetMap integration for real-time delivery location visualization.

The Roles: Redis and Kafka in Project

Component	Role in our Project	Why It's Important
Kafka	Acts as the *real-time message broker* between services (Admin → Backend → Customer).	Handles event streams efficiently (e.g., order status or location updates). Ensures **asynchronous**, **reliable**, and **scalable** message delivery.
Redis	Serves as the *in-memory data store / cache* for the most recent order and location data.	Provides **instant access** to current state without querying databases. Also used for **Pub/Sub** or WebSocket triggers to notify clients instantly.

Project Flow Using Kafka & Redis

1. **Admin** updates order status or live coordinates.
2. **Backend** publishes the update to **Kafka** (topics: `order-updates`, `location-updates`).
3. **Kafka Consumer** reads the event and updates **Redis** with the latest information.
4. **Redis** stores:
   - `order:ORD-49915` → `{status:"Packed"}`
   - `location:admin` → `{lat:28.61, lng:77.20}`
5. **Redis or Backend** triggers a **SSE broadcast** to all connected clients.
6. **Customer Frontend (React)** updates:
   - Order status text
   - Marker on live map (Leaflet / OpenStreetMap)

Services and Docker Setup

Containerized Services Used in the Project

Service Name	Docker Image	Version Used	Description
ZooKeeper	`confluentinc/cp-zookeeper`	7.5.0	Coordinates and manages Kafka brokers. Handles leader election, configuration, and cluster metadata.
Kafka	`confluentinc/cp-kafka`	7.5.0	Kafka broker responsible for real-time event streaming. Depends on ZooKeeper for coordination in legacy mode.
Redis	`redis`	7.2	In-memory key-value store used for caching and Pub/Sub messaging. Provides ultra-fast updates to frontend dashboards.

Each service runs in a separate Docker container, enabling isolated, reproducible, and scalable deployment.

Kafka – Event Streaming Layer

- **Role:** Handles continuous message flow between producers and consumers.
- **Topics Used:**
  - `order-updates`
  - `location-updates`

Sample Kafka Message:

JSON Format

```
{
  "orderId": "ORD-49915",
  "status": "Packed",
  "timestamp": 1762424052948
}
```

Kafka ensures reliable, real-time propagation of order and location updates.

Redis – Real-Time Data Store

- **Role:** Maintains the most recent order and location data for instant access.

Example Redis Data Snapshot:

CLI Output (Part 1)

```
Key: global:location
   Position: (13.0827, 80.2707)
   Full Data: {
     "name": "Warehouse A",
     "lat": 13.0827,
     "lng": 80.2707
   }
```

Redis provides real-time access to frequently changing data.

Redis – Real-Time Data Store

CLI Output (Part 2)

```
Key: order:ORD-J7818
   Full Data: {
     "orderId": "ORD-J7818",
     "customerName": "Sandeep",
     "items": ["a", "b"],
     "status": "Packed",
     "timestamp": 1762495668474
   }
```

- Redis acts as the central in-memory data store linking backend events to frontend updates.
- Enables rapid synchronization between Kafka consumers and live dashboards.

Redis stores and serves the latest real-time state for every order and location.

SSE Hook Integration in React Frontend

- **Hook Used:** Custom `useSSE()` for real-time event handling.
- **Purpose:** Simplifies SSE setup and state updates in React.

Custom React Hook – `useSSE.js`

```
import { useEffect } from "react";

export default function useSSE(onMessage) {
  useEffect(() => {
    const sse = new EventSource("http://localhost:5000/events");
    sse.onmessage = (e) => onMessage(JSON.parse(e.data));
    return () => sse.close();
  }, [onMessage]);
}
```

The useSSE hook abstracts connection logic, enabling clean real-time updates.

SSE Backend Example (1/2)

- **Goal:** Register clients and keep SSE connections alive.
- **Key Logic:** `registerClient()` maintains open HTTP streams.

Node.js – `sseManager.js`

```
let clients = [];
function registerClient(res) {
  res.setHeader("Content-Type", "text/event-stream");
  res.setHeader("Cache-Control", "no-cache");
  res.setHeader("Connection", "keep-alive");
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.flushHeaders();

  clients.push(res);
  const keepAlive = setInterval(() => res.write(":ping\n\n"), 20000);

  res.on("close", () => {
    clearInterval(keepAlive);
    clients = clients.filter(c => c !== res);
  });
}
```

SSE Backend Example (2/2)

- **Goal:** Send updates to all active clients.
- **Key Logic:** `broadcast()` pushes JSON events to each connection.

Node.js – `sseManager.js`

```
function broadcast(event) {
  const data = JSON.stringify(event);
  clients.forEach(c => c.write(`data: ${data}\n\n`));
  console.log(`Sent update to ${clients.length} clients`);
}

module.exports = { registerClient, broadcast };
```

All connected SSE clients receive updates instantly.

Key Differences and Purpose

Comparing `text/event-stream` (SSE) vs Common HTTP Content Types

Feature	`text/event-stream`	**Common Content Types (e.g., `application/json`, `text/plain`)**
Purpose	Used for **Server-Sent Events (SSE)**, where the server continuously pushes real-time updates to the client over a single, long-lived HTTP connection.	Used for **standard HTTP requests/responses**, where each request receives one complete response from the server.
Connection	Connection remains **open** until explicitly closed by the client or server, enabling a continuous stream of updates.	Connection is typically **closed** after the response is delivered (though it may be reused via keep-alive).
Data Format	Data must follow the **SSE format**: UTF-8 encoded plain text, each message prefixed with `data:` and separated by two newlines (`\n\n`).	Data can be structured (e.g., JSON, XML) or plain text, with no strict streaming format within a single response.

`text/event-stream` enables live, continuous updates — unlike regular one-time HTTP responses.

Redis Monitoring Script (checkRedis.js)

- **Purpose:** Display all Redis keys with their current values.
- **Use:** Helps verify synchronization between Kafka, Redis, and frontend.
- **Output:** Shows each key's name, position, and status.

Node.js – checkRedis.js

```
const redis = require("./redis/redisClient");

(async () => {
  const keys = await redis.keys("*");
  console.log(`Found ${keys.length} keys:\n`);
  for (const key of keys) {
    const value = await redis.get(key);
    console.log(` ${key}:`, value);
  }
  process.exit(0);
})();
```

Quickly checks live Redis data to confirm consistency across the system.

Redis Utility Script (flushRedis.js)

- **Purpose:** Clear all Redis keys before fresh data ingestion.
- **Use Case:** Helpful during testing or when resetting Kafka consumers.

Node.js – `flushRedis.js`

```
const redis = require("./redis/redisClient");

(async () => {
  try {
    await redis.flushall();
    console.log(" Redis flushed successfully!");
  } catch (err) {
    console.error(" Flush failed:", err.message);
  } finally {
    process.exit(0);
  }
})();
```

Used to reset the Redis store and start with a clean state.

Conclusion

- **Redis + Kafka = A real-time backbone system**
  - Combines Kafka's scalable event streaming with Redis's in-memory speed.
- **Achieved instant order & location tracking**
  - Real-time status and map updates successfully implemented.
- **Architecture is scalable and reusable**
  - Supports multi-user, multi-order, and high-throughput environments.
- **Practical solution for logistics & delivery systems**
  - Can be extended to ride-sharing, fleet management, or IoT monitoring.

Redis and Kafka together form a robust foundation for any real-time tracking platform.

Problem Statement - 2

Real-time Log Aggregation System

Using Apache Kafka-Redis

Motivation

- In modern microservice architectures, each service generates logs independently.
- Monitoring, debugging, and analyzing distributed logs becomes difficult.
- A central, real-time log aggregation system helps improve observability.

Objective

Goal

Centralize, process, and visualize logs from multiple microservices in real-time.

- Collect logs from multiple producers.
- Stream logs through Apache Kafka.
- Store and broadcast using Redis.
- Visualize logs using a live WebSocket dashboard.

System Components

- **Producers:** Microservices or simulated sources that emit logs.
- **Kafka Broker:** Streams logs through topics.
- **Consumer:** Aggregates and pushes logs to Redis.
- **Redis:** Acts as cache and Pub/Sub broadcaster.
- **Dashboard:** Displays real-time logs via WebSocket.

Tech Stack

- **Backend:** Node.js (KafkaJS, Express)
- **Message Broker:** Apache Kafka
- **Cache/Queue:** Redis
- **Frontend:** HTML, JavaScript, Chart.js
- **Containerization:** Docker & Docker Compose

Data Flow in Real-Time Log Aggregation

1. **Log Producers** generate log events and send them to the Kafka topic `service-logs`.
2. **Kafka Broker** stores these messages and streams them to subscribed consumers.
3. **Kafka Consumer Service** reads log events and forwards them to **Redis**.
4. **Redis** stores the most recent 100 logs and broadcasts new ones using its **Pub/Sub** mechanism.
5. **WebSocket Dashboard** receives live updates and visualizes logs in real-time with charts and status indicators.

Producer (KafkaJS)

- Built using the **KafkaJS** library in Node.js.
- Acts as a **log generator** simulating multiple microservices (e.g., *auth*, *orders*, *payment*).
- Randomly produces log messages every 10 seconds.
- Sends each log as a JSON object to the Kafka topic `service-logs`.
- Ensures continuous real-time data flow into the Kafka broker.

Producers continuously stream application logs to Kafka topics.

Consumer (Kafka + Redis)

- Implemented using **KafkaJS** for consuming messages and **ioredis** for data storage.
- Subscribes to the Kafka topic `service-logs`.
- Parses each received log message and pushes it to **Redis**.
- Publishes new logs to a **Redis Pub/Sub channel** for instant broadcast.
- Enables live updates for the real-time **WebSocket dashboard**.

Consumers process and forward Kafka logs to Redis for real-time visualization.

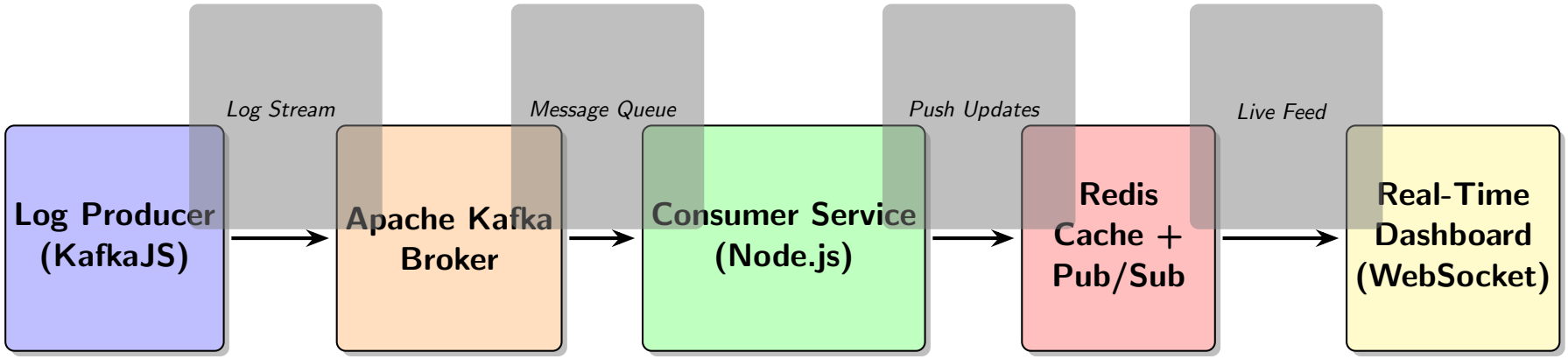
Dashboard (WebSocket + Chart.js)

- Displays real-time logs with color-coded levels.
- Uses WebSocket to receive live updates from Redis.
- Shows INFO, WARN, ERROR distribution using Chart.js.

Docker Compose Setup

- All services (Kafka, Zookeeper, Redis, Node.js app) run via Docker Compose.
- Ensures reproducibility and easy scaling.
- Uses `wait-for-it.sh` to delay app startup until Kafka and Redis are ready.

System Architecture Diagram



Conclusion

- Successfully implemented a real-time, scalable log aggregation system.
- Demonstrated end-to-end data flow using Kafka and Redis.
- Dashboard provides instant insight into distributed system behavior.

Real-time visibility, Faster debugging & Smarter monitoring

Summary

- Developed a **real-time data streaming system** using **Apache Kafka** and **Redis**.
- Implemented two use cases: **Log Aggregation** and **Order Tracking**.
- **Kafka** ensures reliable and scalable message streaming between microservices.
- **Redis** enables instant data access and real-time Pub/Sub updates.
- Built a **live dashboard** for continuous visualization of system activity.
- Deployed the entire stack via **Docker Compose** for easy orchestration and scalability.

References

- Apache Kafka Documentation — `https://kafka.apache.org/documentation/`
- Redis Documentation — `https://redis.io/docs/`
- React-Leaflet Documentation — `https://react-leaflet.js.org/`

Thank You!