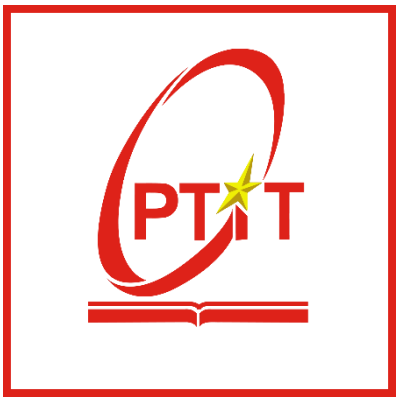
HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA ĐÀO TẠO SAU ĐẠI HỌC



TIỂU LUẬN

MÔN: CÁC HỆ THỐNG PHÂN TÁN

ĐỀ TÀI: MESSAGE QUEUE TRONG HỆ THỐNG PHÂN TÁN ĐẶT VÉ TRỰC TUYẾN

Giảng viên: TS. Kim Ngọc Bách

Sinh viên: Nguyễn Thị Hảo - B24CHHT069

Vũ Hồng Ngọc - B24CHHT087

Nguyễn Phương Đông - B24CHHT062

Mã Lớp: M24CQHT02-B

Hà Nội, 2025

MỤC LỤC

PHÂN CÔNG NHIỆM VỤ

STT	Nội dung công việc	Người phụ trách
1	Phân tích nghiệp vụ, hệ thống	Vũ Hồng Ngọc
2	Xây dựng chương trình	Nguyễn Phương Đông
3	Xây dựng script hiệu năng, thực hiện kiểm tra tải hệ thống	Nguyễn Thị Hảo

DANH MỤC CÁC KÝ HIỆU, CÁC CHỮ VIẾT TẮT

Từ viết tắt	Giải thích
Message Queue	Hệ thống trung gian để truyền thông điệp (message) giữa các thành phần phần mềm.
JMeter	Apache JMeter là công cụ mã nguồn mở để kiểm thử hiệu năng
Kubernetes	Nền tảng mã nguồn mở để điều phối container trong hệ thống phân tán.
Kafka (Apache Kafka)	Nền tảng streaming dữ liệu phân tán (Distributed Streaming Platform)
Distributed System	Hệ thống phân tán gồm nhiều máy tính hoặc node kết nối qua mạng, phối hợp để thực hiện tác vụ chung.
Microservice	Kiến trúc phần mềm chia ứng dụng thành nhiều dịch vụ nhỏ, mỗi dịch vụ đảm nhiệm một chức năng riêng biệt.
Monolithic	Kiến trúc phần mềm trong đó toàn bộ ứng dụng được xây dựng thành một khối duy nhất.

CHƯƠNG 1. GIỚI THIỆU ĐỀ TÀI

1. Tính cấp thiết của đề tài

Trong bối cảnh hiện nay, nhu cầu sử dụng các hệ thống đặt vé trực tuyến ngày càng trở nên phổ biến và thiết yếu, đặc biệt trong các lĩnh vực vận tải (vé máy bay, xe khách, tàu hỏa), giải trí (vé xem phim, ca nhạc, thể thao) và du lịch. Các chiến dịch bán vé theo thời gian thực thường thu hút một lượng lớn người dùng truy cập cùng lúc, tạo ra áp lực lớn lên hạ tầng và dịch vụ backend.

Nếu hệ thống không được thiết kế tối ưu, sẽ dễ xảy ra các vấn đề như:

- Quá tải hệ thống dẫn đến phản hồi chậm hoặc gián đoạn dịch vụ.
- Mất dữ liệu trong quá trình xử lý giao dịch.
- Overbooking – bán vượt số lượng vé thực tế, gây thiệt hại cho doanh nghiệp và trải nghiệm người dùng.

Một trong những giải pháp công nghệ nổi bật để giải quyết vấn đề này là Message Queue (MQ) – cơ chế giao tiếp không đồng bộ giúp giảm độ phụ thuộc giữa các thành phần, hỗ trợ xử lý song song và nâng cao độ tin cậy của hệ thống. MQ cho phép lưu trữ tạm thời các yêu cầu đặt vé, phân phối đến các dịch vụ xử lý theo thứ tự, đảm bảo tính nhất quán dữ liệu ngay cả khi hệ thống chịu tải cao.

Với tính cấp thiết này, đề tài "Ứng dụng Message Queue trong hệ thống phân tán đặt vé trực tuyến" có ý nghĩa thực tiễn cao, giúp giải quyết một bài toán nóng trong phát triển hệ thống phân tán hiện đại.

2. Tổng quan nghiên cứu

Các nghiên cứu và triển khai thực tế trên thế giới cho thấy MQ đóng vai trò quan trọng trong:

- Xử lý lưu lượng lớn: Apache Kafka, RabbitMQ được nhiều hãng thương mại điện tử, hãng hàng không và nền tảng streaming sử dụng để xử lý hàng triệu sự kiện mỗi giây.
- Tối ưu khả năng mở rộng: MQ hỗ trợ scaling theo chiều ngang, phù hợp với kiến trúc microservices.
- Đảm bảo tính nhất quán: Cơ chế commit log và lưu bền dữ liệu của MQ giúp tránh mất yêu cầu trong quá trình xử lý.

Tại Việt Nam, các hệ thống đặt vé trực tuyến vẫn chủ yếu dựa vào mô hình REST API đồng bộ, dẫn đến hạn chế khi gặp tải cao. Việc ứng dụng MQ trong bối cảnh này chưa được phổ biến rộng rãi và vẫn còn thiếu tài liệu hướng dẫn triển khai theo chuẩn công nghiệp.

3. Mục tiêu nghiên cứu

Tập trung chứng minh rằng kiến trúc dựa trên Message Queue (MQ) có thể giúp một hệ thống đặt vé trực tuyến duy trì tính ổn định, hiệu năng cao và khả năng mở rộng tuyến tính trong các đợt cao điểm. Cụ thể:

- ❖ Chức năng và luồng nghiệp vụ
    - Hiện thực hoá chu trình Booking dưới mô hình event-driven sử dụng Message Queue.
    - Bảo đảm đặt chỗ duy nhất (exact-once seat reservation) nhờ cơ chế khoá ghế (*seat-lock*) kết hợp *acknowledgement* của MQ và xử lý trùng lặp có kiểm soát (*idempotent consumer*).
- ❖ Hiệu năng & Khả năng mở rộng
    - Đạt ≥ 10 000 yêu cầu/giây trong kịch bản cao điểm, với độ trễ đầu cuối < 300 ms.
    - Cho phép mở rộng ngang chỉ bằng cách tăng số lượng *consumer* (scale-out), áp dụng chiến lược back-pressure để bảo vệ hệ thống khi tải vượt ngưỡng.
- ❖ Độ tin cậy, Sẵn sàng & An toàn dữ liệu
    - Bảo đảm 99,99 % thời gian sẵn sàng thông qua cơ chế replication, *persistent log,* và dead-letter queue không mất dữ liệu.
- ❖ Bảo mật, Giám sát và Đánh giá thực nghiệm
    - Mã hoá đường truyền TLS, áp dụng token-based authentication cho producer/consumer.
    - Tích hợp Prometheus + Grafana (giám sát) và ELK (log tập trung) nhằm theo dõi throughput, latency, lỗi, CPU/RAM theo thời gian thực.
    - Triển khai proof-of-concept trên Docker/Kubernetes với RabbitMQ (AMQP) và Apache Kafka (log-based); benchmark so sánh hai công

nghệ về thông lượng, độ trễ, mức tiêu thụ tài nguyên và hành vi khi rớt nút, từ đó đề xuất khuyến nghị tối ưu cho từng tình huống sử dụng.

4. Đối tượng và phạm vi nghiên cứu

- **Đối tượng nghiên cứu**:
  - Các nền tảng MQ phổ biến như Apache Kafka, RabbitMQ.
  - Cơ chế giao tiếp không đồng bộ trong hệ thống phân tán.
  - Quy trình xử lý đơn hàng/đặt vé trong môi trường nhiều người dùng đồng thời.
- **Phạm vi nghiên cứu**:
  - Đề tài không triển khai đầy đủ một cổng thanh toán thực tế, cũng như không tích hợp các cơ chế bảo mật nâng cao (PCI-DSS, tokenization, 3-D Secure…).
  - Trọng tâm phân tích đặt vào vai trò của Message Queue trong kiến trúc hệ thống phân tán, thay vì mở rộng sang các thành phần phụ trợ khác.
  - Các khía cạnh phi chức năng như bảo mật toàn diện, trải nghiệm người dùng (UX/UI) và tối ưu chi phí vận hành chỉ được đề cập ở mức tổng quan, không đi sâu trong tiểu luận.

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

2.1 Hệ thống phân tán

Hệ thống phân tán (Distributed System) là một tập hợp các máy tính độc lập, kết nối với nhau thông qua mạng truyền thông và phối hợp hoạt động để thực hiện một nhiệm vụ chung. Mặc dù bao gồm nhiều thành phần vật lý khác nhau, hệ thống phân tán được thiết kế sao cho người dùng cảm nhận như đang tương tác với một hệ thống thống nhất. Khái niệm này ngày càng trở nên quan trọng trong kỷ nguyên số hiện nay, khi nhu cầu xử lý dữ liệu lớn, tính sẵn sàng cao và khả năng mở rộng linh hoạt là yêu cầu thiết yếu của các hệ thống phần mềm hiện đại.

Một hệ thống phân tán có nhiều đặc trưng nổi bật. Trước hết là tính trong suốt (transparency), tức là người dùng không cần quan tâm đến việc tài nguyên được lưu trữ ở đâu, hệ thống sẽ tự động xử lý để cung cấp kết quả như mong muốn. Các khía cạnh khác của tính trong suốt bao gồm trong suốt vị trí, trong suốt truy cập, trong suốt lỗi và trong suốt sao chép. Bên cạnh đó, tính mở (openness) cho phép hệ thống dễ dàng mở rộng, tích hợp thêm các dịch vụ mới hoặc thay đổi thành phần mà không ảnh hưởng đến toàn bộ hệ thống. Tính mở rộng (scalability) là khả năng duy trì hiệu suất khi hệ thống gia tăng về quy mô người dùng, dữ liệu hoặc khu vực địa lý. Ngoài ra, tính chịu lỗi (fault tolerance) cũng là một yếu tố quan trọng, giúp hệ thống tiếp tục hoạt động ngay cả khi một số thành phần bị lỗi.

Kiến trúc của hệ thống phân tán có thể được tổ chức theo nhiều mô hình, trong đó phổ biến nhất là kiến trúc client-server, peer-to-peer (P2P) và microservices. Kiến trúc client-server đơn giản nhưng dễ bị tắc nghẽn tại phía server. Ngược lại, P2P phân tán đều tải nhưng khó kiểm soát và đồng bộ dữ liệu. Mô hình microservices, với việc chia nhỏ các chức năng thành nhiều dịch vụ độc lập, hiện đang là xu hướng trong phát triển hệ thống quy mô lớn nhờ vào khả năng dễ mở rộng và dễ bảo trì.

Các thành phần cơ bản trong một hệ thống phân tán bao gồm các nút (node) xử lý độc lập, mạng truyền thông, phần mềm trung gian (middleware) giúp các dịch vụ giao tiếp với nhau, các giao thức truyền thông (protocol) như HTTP,

gRPC, hoặc AMQP, và các cơ chế hỗ trợ như hàng đợi thông điệp (Message Queue). Message Queue đóng vai trò quan trọng trong việc tổ chức truyền thông không đồng bộ, nâng cao hiệu suất và độ tin cậy cho hệ thống.

Mặc dù hệ thống phân tán mang lại nhiều lợi ích như khả năng mở rộng, hiệu năng cao, và tăng tính sẵn sàng, nó cũng tiềm ẩn nhiều thách thức. Các vấn đề như đồng bộ trạng thái, xử lý lỗi phức tạp, độ trễ mạng và đảm bảo an toàn dữ liệu đòi hỏi hệ thống phải được thiết kế kỹ lưỡng. Ngoài ra, việc đảm bảo bảo mật trong môi trường phân tán – nơi dữ liệu di chuyển qua nhiều node và mạng khác nhau – cũng là một bài toán khó cần giải quyết.

Trong thực tế, hệ thống phân tán được ứng dụng rộng rãi trong các nền tảng lớn như Google, Facebook, Amazon, Netflix... Các ứng dụng đặt vé trực tuyến, hệ thống thương mại điện tử, ngân hàng số, dịch vụ phát trực tuyến, mạng xã hội đều sử dụng kiến trúc phân tán để đảm bảo khả năng phục vụ hàng triệu người dùng cùng lúc với độ tin cậy và tốc độ cao. Với sự phát triển mạnh mẽ của công nghệ điện toán đám mây và hạ tầng viễn thông, hệ thống phân tán không chỉ là xu hướng mà đã trở thành nền tảng thiết yếu cho mọi hệ thống phần mềm hiện đại.

2.2 So sánh Microservice và Monolithic

Tiêu chí	Monolithic (Nguyên khối)	Microservices
Cấu trúc	Một khối duy nhất	Nhiều dịch vụ nhỏ, độc lập
Triển khai	Deploy toàn bộ ứng dụng	Deploy từng service riêng
Mở rộng	Scale toàn hệ thống	Scale từng phần riêng lẻ
Chịu lỗi	Một lỗi có thể sập toàn hệ thống	Lỗi từng phần, hệ thống vẫn chạy được

Bảng 2.1: So sánh Microservice và Monolithic

2.3. Messege Queue

Message Queue (MQ) là một cơ chế hàng đợi thông điệp cho phép các thành phần trong hệ thống phân tán giao tiếp không đồng bộ với nhau bằng cách gửi và nhận các thông điệp thông qua một hàng đợi trung gian. Thay vì giao tiếp trực tiếp, các thành phần gọi là producer (bên gửi thông điệp) và consumer (bên nhận và xử lý thông điệp) trao đổi dữ liệu gián tiếp qua hàng đợi. Điều này giúp giảm sự phụ thuộc chặt chẽ giữa các dịch vụ, tăng khả năng chịu lỗi và mở rộng của hệ thống.

Về mặt cơ chế, khi một producer gửi một thông điệp vào hàng đợi, thông điệp đó sẽ được lưu tạm thời trong hệ thống MQ cho đến khi một consumer sẵn sàng nhận và xử lý nó. Việc gửi và nhận có thể diễn ra tại các thời điểm khác nhau, nên MQ giúp đảm bảo rằng thông điệp không bị mất ngay cả khi dịch vụ nhận tạm thời bị gián đoạn. Cơ chế này rất hữu ích trong các hệ thống có lượng

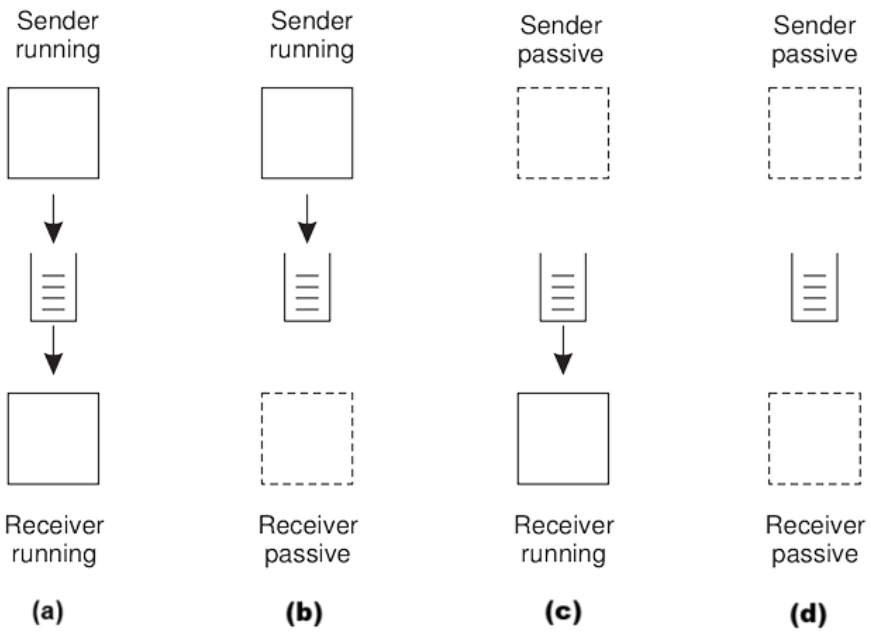
truy cập lớn hoặc cần xử lý theo thứ tự, như hệ thống đặt vé trực tuyến, thanh toán điện tử, hay xử lý đơn hàng.

Một trong những mô hình giao tiếp phổ biến của MQ là mô hình point-to-point (work queue), trong đó mỗi thông điệp được xử lý bởi một consumer duy nhất, đảm bảo không có trùng lặp trong quá trình xử lý. Ngoài ra còn có mô hình publish-subscribe, nơi một producer phát thông điệp và nhiều consumer có thể nhận được bản sao của thông điệp đó, phù hợp với các trường hợp thông báo trạng thái hệ thống hoặc sự kiện.

Để tăng độ tin cậy, hầu hết các hệ thống MQ đều hỗ trợ các cơ chế như acknowledgment (xác nhận), retry (thử lại nếu consumer thất bại), và dead-letter queue (luu trữ thông điệp lỗi không xử lý được để theo dõi sau). Ngoài ra, một số hệ thống MQ còn cung cấp tính năng ưu tiên thông điệp, trì hoãn xử lý hoặc kiểm soát tốc độ tiêu thụ (rate-limiting).

MQ đặc biệt phù hợp với các kiến trúc microservices hoặc event-driven trong hệ thống phân tán, giúp tăng khả năng mở rộng, tách rời chức năng và tăng tính chịu lỗi. Trong hệ thống đặt vé trực tuyến, MQ được dùng để ghi nhận yêu cầu đặt vé, xử lý tồn kho, gửi thông báo và kết nối đến dịch vụ thanh toán một cách linh hoạt và ổn định, ngay cả trong trường hợp có hàng nghìn người dùng truy cập đồng thời.

Hiện nay có nhiều hệ thống MQ phổ biến được sử dụng rộng rãi như RabbitMQ (hỗ trợ AMQP, dễ triển khai), Apache Kafka (phù hợp cho luồng dữ liệu lớn và event streaming), ActiveMQ, Amazon SQS, hay Google Pub/Sub. Mỗi hệ thống có điểm mạnh riêng, nhưng đều hướng đến mục tiêu chung là giúp hệ thống giao tiếp ổn định, hiệu quả và dễ mở rộng trong môi trường phân tán.



Hình 1.1: Các trường hợp của hàng đợi

(Nguồn: Figure 4.26 - Distributed Systems Third edition Version 3.03 (2020) by Maarten van Steen and Andrew S. Tanenbaum

2.4 Kubenetes

Kubernetes – Nền tảng điều phối container trong hệ thống phân tán.

Kubernetes là một nền tảng mã nguồn mở do Google phát triển, hiện được duy trì bởi Cloud Native Computing Foundation (CNCF), dùng để tự động hóa việc triển khai, mở rộng và quản lý các ứng dụng container. Trong bối cảnh hiện nay, khi kiến trúc microservices và container (đặc biệt là Docker) trở nên phổ biến, Kubernetes đóng vai trò trung tâm trong việc điều phối các container trong môi trường phân tán hoặc điện toán đám mây.

Kubernetes giúp quản lý các pod – đơn vị triển khai nhỏ nhất chứa một hoặc nhiều container – và phân phối chúng trên các node trong một cụm máy chủ (cluster). Các node có thể là máy vật lý hoặc máy ảo. Hệ thống này cung cấp các chức năng mạnh mẽ như cân bằng tải (load balancing), tự động scale, tự phục hồi (self-healing) khi container lỗi, và rolling updates để cập nhật dịch vụ không gián đoạn. Điều này giúp đảm bảo tính sẵn sàng cao, linh hoạt và dễ mở rộng cho hệ thống.

Một số thành phần quan trọng trong kiến trúc Kubernetes bao gồm:

- ❖ Control Plane (bao gồm API Server, Scheduler, Controller Manager, v.v.) chịu trách nhiệm điều phối toàn bộ hệ thống.
- ❖ Worker Nodes (gồm Kubelet và kube-proxy) thực thi và quản lý các pod.
- ❖ etcd là kho dữ liệu phân tán dùng để lưu trữ trạng thái của cụm (cluster state).
- ❖ Kubelet là tác nhân trên mỗi node để quản lý container cục bộ.
- ❖ Kubernetes API là giao diện chính để giao tiếp và điều khiển các thành phần khác.

Một điểm mạnh khác của Kubernetes là khả năng tích hợp với các hệ thống CI/CD (Continuous Integration/Continuous Deployment), giúp tự động triển khai phiên bản mới của ứng dụng một cách linh hoạt, kiểm soát tốt lỗi, rollback khi cần thiết. Bên cạnh đó, Kubernetes hỗ trợ tích hợp với các công cụ quản lý cấu hình, giám sát và lưu trữ như Helm, Prometheus, Grafana, Fluentd.

Trong bối cảnh triển khai các hệ thống phân tán hiện đại như hệ thống đặt vé trực tuyến, Kubernetes có thể được sử dụng để triển khai các dịch vụ như Booking Service, Inventory, Notification, Payment,... dưới dạng các pod độc lập, giao tiếp qua mạng nội bộ (service), và tích hợp thêm các giải pháp như Message Queue (RabbitMQ, Kafka) để xử lý không đồng bộ. Nhờ đó, hệ thống có thể tự động co giãn tài nguyên khi lưu lượng thay đổi, xử lý lỗi nhanh chóng, và đảm bảo uptime cao.

Hiện tại, Kubernetes đã trở thành tiêu chuẩn công nghiệp cho việc quản lý và triển khai các ứng dụng container trong môi trường đám mây hoặc tại chỗ (on-premise), được hỗ trợ mạnh mẽ bởi các nhà cung cấp lớn như Google Kubernetes Engine (GKE), Amazon EKS, Azure AKS, Red Hat OpenShift,... Điều này cho thấy tầm quan trọng và vai trò không thể thiếu của Kubernetes trong thiết kế và vận hành các hệ thống phân tán quy mô lớn ngày nay.

2.5. Kafka

Apache Kafka – Nền tảng truyền thông điệp phân tán và xử lý luồng dữ liệu thời gian thực

Apache Kafka là một nền tảng truyền thông điệp (message streaming platform) mã nguồn mở, được phát triển ban đầu bởi LinkedIn và hiện được duy trì bởi Apache Software Foundation. Kafka được thiết kế để xử lý luồng dữ liệu theo thời gian thực với hiệu suất cao, độ bền vững tốt và khả năng mở rộng mạnh mẽ, phù hợp cho các hệ thống phân tán quy mô lớn, nơi có nhu cầu trao đổi thông tin giữa các dịch vụ hoặc xử lý dữ liệu lớn theo thời gian thực.

Khác với các hệ thống Message Queue truyền thống như RabbitMQ hay ActiveMQ, Kafka được xây dựng theo mô hình log phân tán (distributed commit log), trong đó các thông điệp (message) được ghi theo thứ tự thời gian vào các topic – mỗi topic được chia nhỏ thành nhiều partition giúp hệ thống có thể xử lý song song và phân tán dữ liệu trên nhiều máy chủ (broker). Producer sẽ gửi dữ liệu vào topic, và consumer sẽ đăng ký (subscribe) để đọc dữ liệu từ các topic đó. Mỗi consumer có thể tùy chọn vị trí đọc (offset) trong topic, từ đó mang lại tính linh hoạt và mạnh mẽ trong việc xử lý dữ liệu quá khứ, đồng thời hỗ trợ khả năng replay dữ liệu khi cần.

Một trong những ưu điểm nổi bật của Kafka là khả năng xử lý hàng triệu thông điệp mỗi giây với độ trễ thấp, nhờ vào kiến trúc phân tán và cơ chế ghi đĩa tuần tự (sequential disk write). Kafka cũng hỗ trợ replication để đảm bảo độ bền của dữ liệu (durability) và khả năng chịu lỗi (fault tolerance), nhờ đó dữ liệu vẫn được duy trì ngay cả khi một số broker gặp sự cố. Kafka gồm ba thành phần chính:

- Producer: gửi thông điệp đến các topic.
- Consumer: nhận và xử lý thông điệp từ topic.
- Broker: máy chủ chịu trách nhiệm lưu trữ và phân phối dữ liệu. Ngoài ra, Kafka còn hỗ trợ các consumer group – cho phép chia tải dữ liệu giữa nhiều consumer mà không bị trùng lặp, và Kafka Connect – giúp tích

hợp Kafka với các hệ thống bên ngoài như cơ sở dữ liệu, hệ thống lưu trữ hoặc hệ thống giám sát. Một thành phần mở rộng quan trọng là Kafka Streams, dùng để xử lý luồng dữ liệu ngay bên trong Kafka mà không cần hệ thống xử lý ngoài.

Trong hệ thống phân tán hiện đại, Kafka thường được dùng làm "xương sống" để truyền dữ liệu giữa các dịch vụ độc lập theo mô hình event-driven. Đặc biệt trong các hệ thống như đặt vé trực tuyến, Kafka có thể giữ vai trò làm hàng đợi thông điệp chính để tiếp nhận các sự kiện như "đặt vé thành công", "thanh toán hoàn tất", "gửi thông báo", v.v. Khi có lượng người dùng lớn truy cập đồng thời, Kafka giúp tách biệt luồng xử lý, đảm bảo hệ thống không bị quá tải cục bộ, đồng thời ghi lại toàn bộ luồng sự kiện phục vụ việc giám sát, xử lý lỗi hoặc phân tích dữ liệu sau này.

Hiện nay, Apache Kafka được sử dụng bởi nhiều công ty công nghệ hàng đầu như LinkedIn, Uber, Netflix, Airbnb, và trong nhiều lĩnh vực khác nhau từ tài chính, thương mại điện tử đến công nghiệp IoT. Sự phổ biến và sức mạnh của Kafka khiến nó trở thành lựa chọn hàng đầu cho các hệ thống truyền thông điệp phân tán có yêu cầu cao về hiệu năng, độ tin cậy và xử lý theo thời gian thực.

2.6 Java spring boot

Spring Boot – Nền tảng phát triển ứng dụng web và microservices bằng Java

Spring Boot là một framework mã nguồn mở thuộc hệ sinh thái Spring Framework, được phát triển bởi Pivotal Software (nay là VMware). Spring Boot ra đời nhằm đơn giản hóa quá trình phát triển các ứng dụng Java – đặc biệt là các ứng dụng web và microservices – bằng cách cung cấp các thiết lập mặc định, tự động cấu hình và các công cụ tích hợp sẵn để giúp lập trình viên xây dựng ứng dụng một cách nhanh chóng, linh hoạt và dễ triển khai.

Không giống với Spring truyền thống yêu cầu cấu hình XML phức tạp, Spring Boot áp dụng nguyên tắc "convention over configuration" (ưu tiên quy ước thay vì cấu hình) để rút ngắn thời gian phát triển. Khi sử dụng Spring Boot,

lập trình viên có thể khởi tạo một ứng dụng web RESTful chỉ trong vài dòng lệnh nhờ vào sự hỗ trợ của Spring Initializr – công cụ tạo project với cấu hình sẵn. Ngoài ra, Spring Boot tích hợp chặt chẽ với các thành phần phổ biến trong hệ sinh thái Spring như Spring MVC, Spring Data JPA, Spring Security, và Spring Cloud.

Spring Boot hỗ trợ tự động cấu hình (auto-configuration), giúp hệ thống tự động hiểu và cài đặt các thành phần cần thiết dựa trên các thư viện có trong classpath. Ví dụ, nếu có thư viện web trong project, Spring Boot sẽ tự động cấu hình embedded server như Tomcat hoặc Jetty, và thiết lập các endpoint HTTP mà không cần can thiệp thủ công. Tính năng này giúp giảm đáng kể khối lượng công việc liên quan đến cấu hình hệ thống.

Một ưu điểm lớn của Spring Boot là khả năng tạo ra các ứng dụng độc lập (standalone) – tức là không cần cài đặt server ngoài như Apache Tomcat – vì ứng dụng có thể tự chạy như một file .jar hoặc .war nhúng web server. Điều này giúp đơn giản hóa việc triển khai, đóng gói và phân phối ứng dụng, đặc biệt thuận tiện trong các môi trường như Docker, Kubernetes hoặc CI/CD pipelines.

Trong kiến trúc microservices, Spring Boot thường được sử dụng để xây dựng từng dịch vụ nhỏ (microservice), nhờ vào khả năng tách biệt rõ ràng các module, hỗ trợ giao tiếp qua HTTP (REST), gRPC hoặc messaging (RabbitMQ, Kafka), cùng với khả năng tích hợp dễ dàng với Spring Cloud để quản lý cấu hình phân tán, service discovery, load balancing, failover, và các chức năng nâng cao như circuit breaker hoặc distributed tracing.

Đối với các hệ thống đặt vé trực tuyến hoặc thương mại điện tử, Spring Boot cho phép xây dựng nhanh chóng các thành phần như: dịch vụ đặt vé, dịch vụ kiểm tra tồn kho, dịch vụ thanh toán, và dịch vụ thông báo, mỗi thành phần được triển khai như một microservice riêng biệt. Nhờ sự hỗ trợ của Spring Boot với các message broker như Kafka, RabbitMQ, và các công cụ giám sát như Actuator, Prometheus, Spring Boot tạo nên một hệ sinh thái hoàn chỉnh để xây dựng và vận hành các hệ thống phân tán có độ tin cậy cao, hiệu năng tốt và dễ mở rộng.

Hiện nay, Spring Boot là một trong những framework Java phổ biến nhất, được sử dụng bởi hàng triệu lập trình viên và hàng nghìn công ty trên toàn thế giới, từ startup đến các tập đoàn lớn. Sự kết hợp giữa tính đơn giản, hiệu quả và khả năng mở rộng khiến Spring Boot trở thành lựa chọn hàng đầu cho việc phát triển các hệ thống web và microservices trong môi trường hiện đại.

2.7 Test hiệu năng Jmeter

Apache JMeter là một công cụ mã nguồn mở do Apache Software Foundation phát triển, được sử dụng phổ biến trong kiểm thử hiệu năng của các hệ thống phần mềm. Ban đầu, JMeter được thiết kế để kiểm thử các ứng dụng web, nhưng hiện nay nó đã hỗ trợ nhiều loại ứng dụng khác nhau như API REST, SOAP, cơ sở dữ liệu, FTP, SMTP/POP3, JMS, và cả WebSocket thông qua các plugin mở rộng.

Mục đích chính của JMeter là thực hiện các loại kiểm thử như kiểm thử tải (load testing), kiểm thử stress (stress testing), kiểm thử hiệu năng (performance testing) và cả kiểm thử chức năng (functional testing). Nó cho phép mô phỏng hàng trăm hoặc hàng nghìn người dùng ảo gửi yêu cầu đến máy chủ để đánh giá khả năng đáp ứng và hiệu suất của hệ thống.

JMeter hoạt động dựa trên cấu trúc Test Plan – kế hoạch kiểm thử gồm nhiều thành phần khác nhau. Thread Group định nghĩa số lượng người dùng mô phỏng và kịch bản họ sẽ thực hiện. Các Sampler dùng để gửi yêu cầu tới server như HTTP Request, JDBC Request hoặc các giao thức khác. Listener là nơi hiển thị kết quả kiểm thử dưới dạng bảng, biểu đồ hoặc file log. Ngoài ra, JMeter còn hỗ trợ Assertions để kiểm tra kết quả phản hồi, Timers để tạo độ trễ giữa các yêu cầu, và các thành phần tiền/xử lý sau (Pre/Post-Processors) để can thiệp vào luồng kiểm thử.

JMeter có thể được sử dụng dưới dạng giao diện đồ họa (GUI) để thiết kế kịch bản kiểm thử dễ dàng, hoặc chạy ở chế độ không giao diện (Non-GUI) để kiểm thử ở quy mô lớn, đặc biệt hữu ích trong môi trường CI/CD. Ví dụ, để kiểm thử một API REST với JMeter, bạn có thể tạo một Thread Group mô phỏng 100

người dùng, thêm một HTTP Request để gửi GET request tới một URL cụ thể, và sử dụng Listener để theo dõi kết quả.

Kết quả từ JMeter rất đa dạng và chi tiết, bao gồm số lượng yêu cầu thành công/thất bại, thời gian phản hồi trung bình, nhỏ nhất, lớn nhất, số người dùng đồng thời và biểu đồ hiệu suất theo thời gian. Nhờ đó, JMeter là một công cụ mạnh mẽ, linh hoạt và dễ mở rộng, rất phù hợp cho các nhóm phát triển phần mềm cần kiểm thử hiệu năng hệ thống một cách hiệu quả.

CHƯƠNG 3: XÂY DỰNG HỆ THỐNG

3.1 Tổng quan hệ thống

3.1.1 Mô tả hệ thống

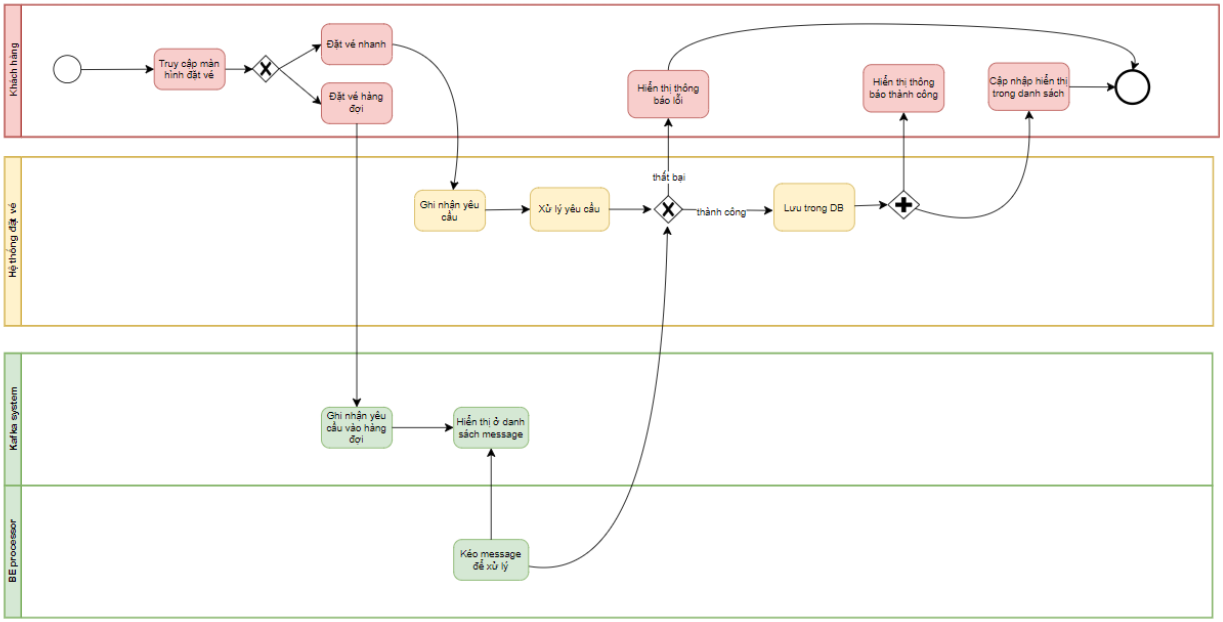
Hệ thống được thiết kế nhằm mô phỏng và đánh giá hiệu quả của việc áp dụng Message Queue (cụ thể là Apache Kafka) trong xử lý giao dịch đặt vé trực tuyến. Mục tiêu chính là so sánh hai cách thức tiếp nhận và xử lý yêu cầu:

- Phương thức đồng bộ (Mua vé nhanh) – xử lý yêu cầu trực tiếp.
- Phương thức không đồng bộ (Mua vé qua Kafka) – xử lý thông qua hàng đợi thông điệp.

Việc xây dựng song song hai phương thức này trên cùng một nền tảng cho phép kiểm chứng tác động của MQ tới hiệu suất, khả năng chịu tải và độ ổn định của hệ thống phân tán trong điều kiện lưu lượng truy cập cao. Qua đó, hệ thống đóng vai trò như một mô hình thực nghiệm giúp phân tích ưu – nhược điểm của từng phương pháp, từ đó rút ra các khuyến nghị áp dụng trong thực tế.

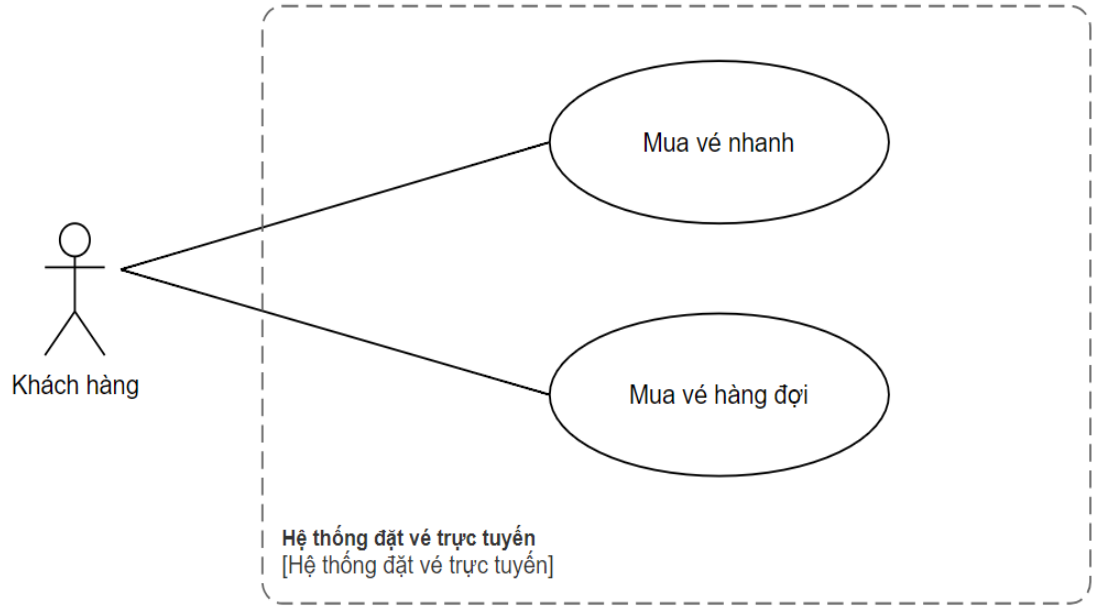
3.1.2 Cấu trúc và chức năng

3.1.2.1 BPMN luồng nghiệp vụ tổng quan hệ thống đặt vé trực tuyến



Hình 3.1: BPMN luồng nghiệp vụ tổng quan hệ thống đặt vé trực tuyến

3.1.2.2 Use case tổng quan hệ thống đặt vé trực tuyến



Hình 3.2: Use case tổng quan hệ thống đặt vé trực tuyến

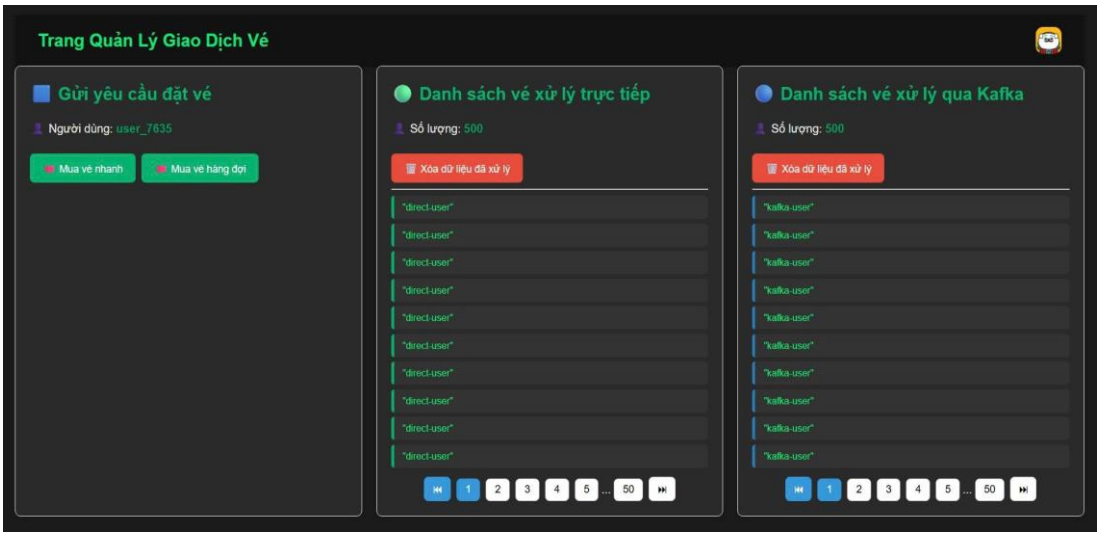
3.1.2.2 Chức năng của hệ thống

Hệ thống gồm hai chức năng chính:

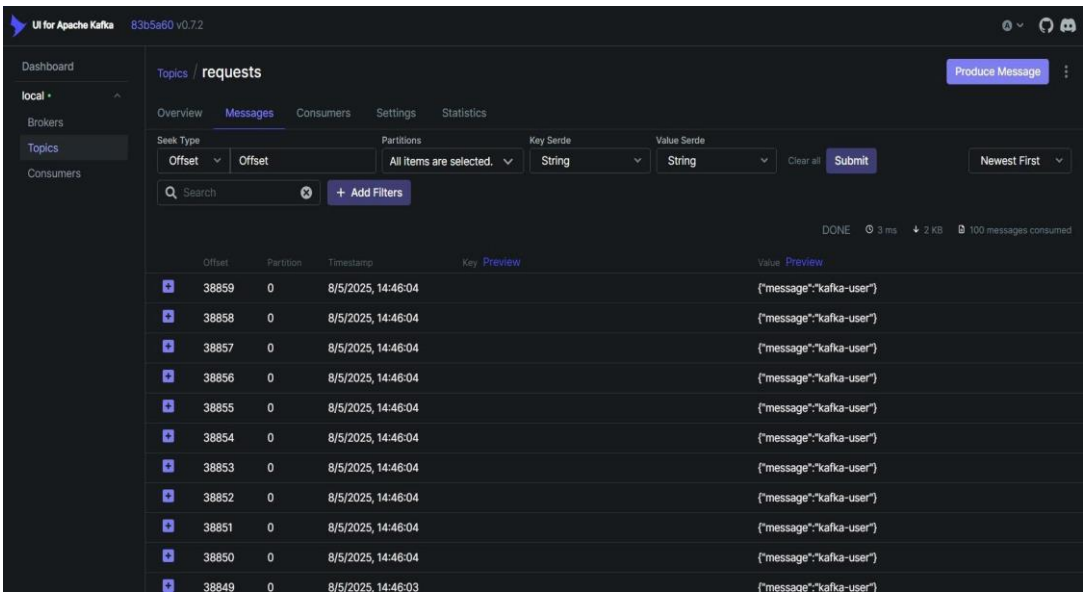
- **Mua vé nhanh** (Synchronous Processing)
  - Quy trình: Người dùng gửi yêu cầu → Backend xử lý ngay lập tức → Trả kết quả trực tiếp.
  - Đặc điểm: Thời gian phản hồi nhanh khi tải thấp, nhưng dễ bị quá tải và giảm hiệu năng khi nhiều người dùng truy cập đồng thời.
- **Mua vé qua Kafka** (Asynchronous Processing)
  - Quy trình: Người dùng gửi yêu cầu → Kafka lưu vào hàng đợi → Dịch vụ xử lý backend lấy yêu cầu ra và thực hiện giao dịch.
  - Đặc điểm: Giảm tải trực tiếp cho backend, xử lý song song hiệu quả, đảm bảo tính nhất quán dữ liệu và hạn chế mất dữ liệu khi hệ thống chịu áp lực lớn.

Thông qua hai chức năng này, hệ thống cho phép đo lường các chỉ số như thời gian phản hồi, tỷ lệ thành công, khả năng chịu tải và độ ổn định, tạo cơ sở dữ liệu thực nghiệm để phân tích và so sánh hai mô hình.

3.2 Giao diện hệ thống



Hình 3.2a: Giao diện hệ thống đặt vé



Hình 3.2b: Message Queue GUI

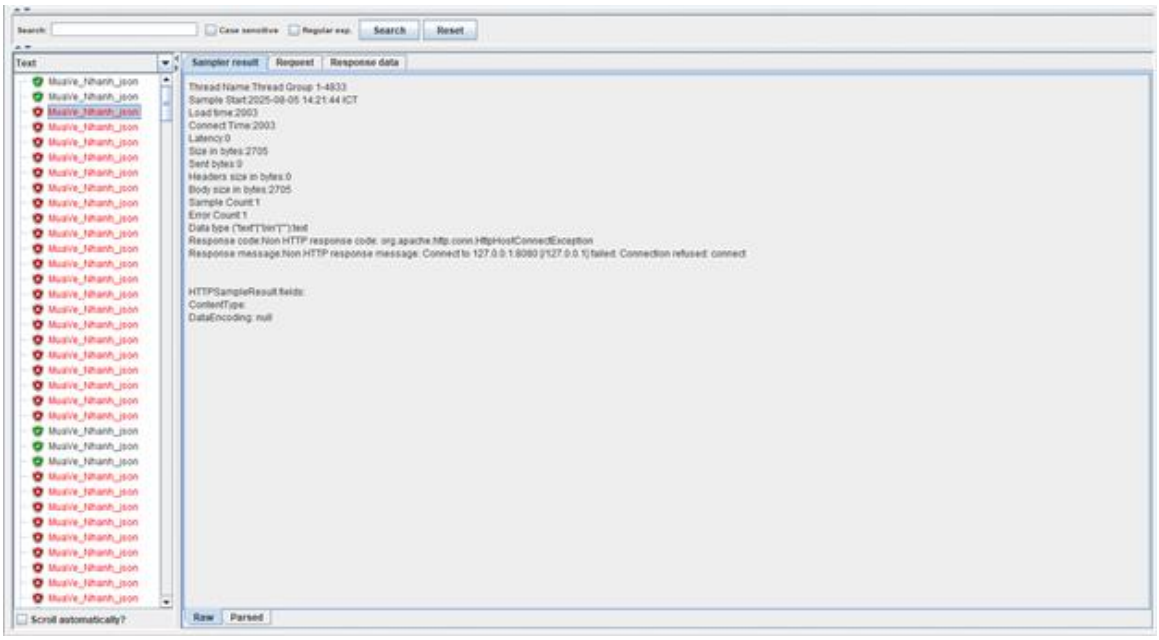
CHƯƠNG 4: KẾT LUẬN & BÀI HỌC RÚT RA

4.1 Các kịch bản đổ tải trên Jmeter

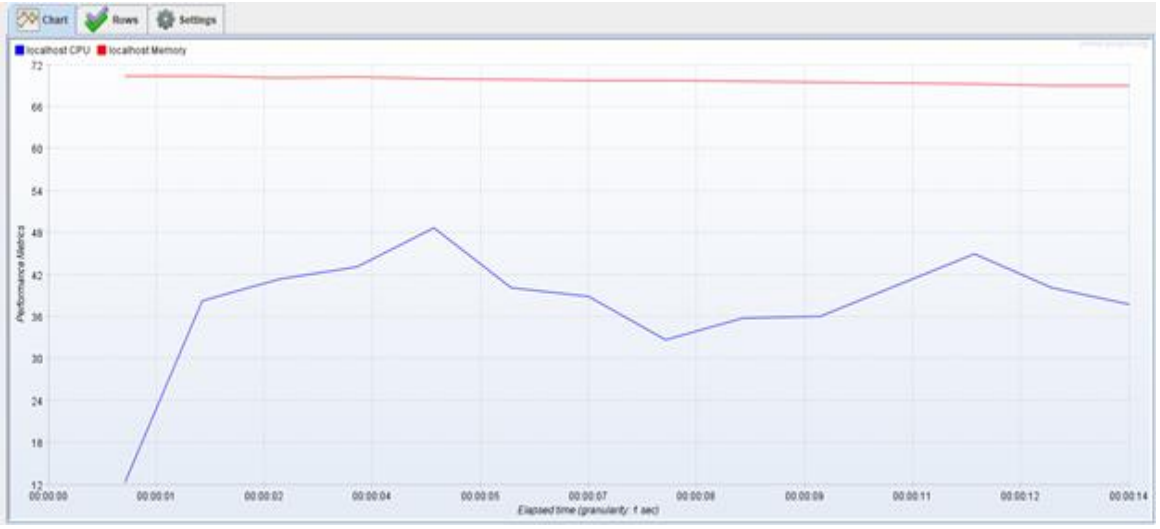
4.1.1. Test tải lần 1

- Số lượng: 5000 CCU
- Thời gian: từ đầu đến khi đủ 5000 người 15s/request/người

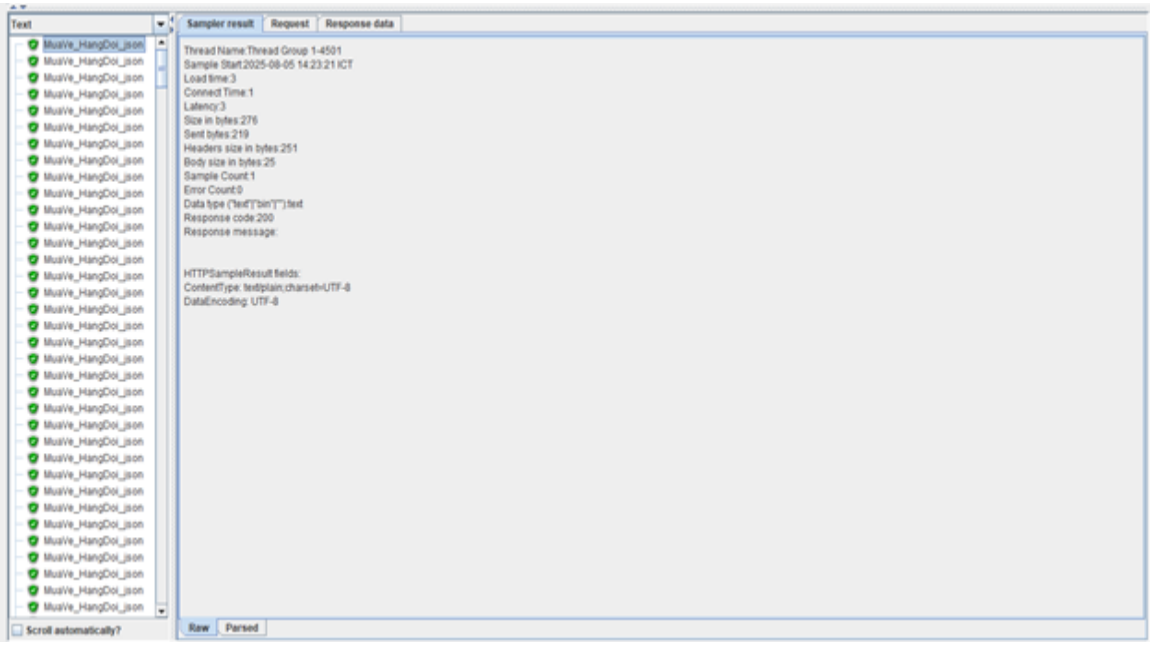
Hình 4.1: Kết quả đổ tải khi gửi trực tiếp 5000 CCU



Hình 4.2: Kết quả tải khi gửi trực tiếp 5000 CCU



Hình 4.3: Kết quả tải khi gửi qua Kafka 5000 CCU



Hình 4.4: Kết quả tải khi gửi qua Kafka 5000 CCU

4.1.1.1 Bảng so sánh sự khác nhau khi thực hiện gửi 5000 CCU

Tiêu chí đánh giá	Gửi trực tiếp	Gửi qua Kafka
Avg. Response Time	3 359 ms (rất cao)	3 ms
Throughput (thời gian xử lý avg request/s)	179.9 req/s	333.43 req/s
Error	86.04%	0 %
Độ ổn định	Quá tải, lỗi kết nối, timeout	Mượt, ổn định, không lỗi
Khả năng scale	Khó mở rộng, giới hạn backend	Dễ scale, backend xử lý async

Bảng 4.1: So Sánh tải hệ thống với 5000 CCU

4.1.2 Test tải lần 2

- Số lượng: 500 CCU
- Thời gian: từ đầu đến khi đủ 500 người 15s/request/người

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
MuaVe_Nhanh_json	500	4427	260	10968	2530.87	0.00%	24.5/sec	6.53	5.28	273.0
TOTAL	500	4427	260	10968	2530.87	0.00%	24.5/sec	6.53	5.28	273.0

Hình 4.5: Kết quả tải khi gửi trực tiếp với 500 CCU

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
MuaVe_HangDoi_json	500	4	3	19	1.53	0.00%	33.4/sec	9.00	7.14	276.0
TOTAL	500	4	3	19	1.53	0.00%	33.4/sec	9.00	7.14	276.0

Hình 4.6: Kết quả tải khi gửi qua kafka với 500 CCU

4.1.2.1. Bảng so sánh sự khác nhau khi thực hiện gửi 500 CCU

Tiêu chí đánh giá	Gửi trực tiếp	Gửi qua Kafka
Avg. Response Time	~4427 ms	4 ms
Throughput (thời gian xử lý avg request/s)	24.5 req/sec	33.4 req/sec
Error	0 %	0 %
Độ ổn định	Độ trễ cao, lệch chuẩn lớn	Độ trễ thấp, ổn định
Khả năng scale	Khó mở rộng, giới hạn backend	Dễ scale, backend xử lý async

Bảng 4.2: So Sánh tải hệ thống với 5000 CCU

KẾT LUẬN

- **Ưu điểm:**
  - **Giảm tải tức thời cho hệ thống backend**: Kafka đóng vai trò hàng đợi trung gian giúp hấp thụ lượng lớn yêu cầu người dùng, tránh gây "nghẽn cổ chai" tại API chính.
  - **Tăng độ tin cậy**: Dữ liệu không bị mất nhờ cơ chế ghi bền (*durable logs*) và kiểm soát phân phối thông điệp (*acknowledgement, retry*).
  - **Hiệu năng vượt trội**: Độ trễ trung bình thấp
  - **Dễ mở rộng**: Có thể scale-out bằng cách tăng số lượng consumer mà không ảnh hưởng đến producer.
- **Nhược điểm và thách thức:**
  - **Tăng độ phức tạp hệ thống**: Việc tích hợp Kafka yêu cầu thêm thành phần (broker, topic, consumer group), cần kiến thức vận hành.
  - **Độ trễ đầu cuối phụ thuộc vào consumer**: Nếu consumer chậm hoặc lỗi, hệ thống sẽ xử lý chậm hơn mong đợi.
  - **Chi phí triển khai cao hơn ban đầu**: Đặc biệt với Kafka cluster nhiều node, cần giám sát và bảo trì riêng biệt.

TÀI LIỆU THAM KHẢO

[1] https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture

[2] https://topdev.vn/blog/kafka-la-gi/

[3] https://jmeter.apache.org/

[4] https://minikube.sigs.k8s.io/docs/