VIVEKANAND EDUCATION SOCIETY'S INSTITUTE OF TECHNOLOGY

(An Autonomous Institute Affiliated to University of Mumbai)

Department of Computer Engineering

Project Report on

Optimizing Kafka-Based Supply Chain Architectures: A Comprehensive Performance Analysis

In partial fulfilment of the Fourth Year, Bachelor of Engineering (B.E.) Degree in Computer Engineering at the University of Mumbai
Academic Year 2024 - 25

By

Simran Ahuja D17C / 02

Jesica Bijju D17C / 10

Sejal Datir D17C / 14

Sania Khan D17C / 36

Project Mentor

Dr. (Mrs.) Nupur Giri

(A.Y. 2024 - 25)

VIVEKANAND EDUCATION SOCIETY'S INSTITUTE OF TECHNOLOGY

(An Autonomous Institute Affiliated to University of Mumbai)

Department of Computer Engineering

Certificate

This is to certify that **Simran Ahuja (D17C - 02), Jesica Bijju (D17C - 10), Sejal Datir (D17C - 14), & Sania Khan (D17C - 36)** of Fourth Year Computer Engineering studying under the University of Mumbai have satisfactorily completed the project on **"Optimizing Kafka-Based Supply Chain Architectures: A Comprehensive Performance Analysis"** as a part of their coursework of PROJECT - II for Semester - VIII under the guidance of their mentor **Dr. (Mrs.) Nupur Giri** in the year 2024 - 25.

This thesis/dissertation/project report entitled **"Optimizing Kafka-Based Supply Chain Architectures: A Comprehensive Performance Analysis"** by Simran Ahuja, Jesica Bijju, Sejal Datir & Sania Khan is approved for the degree of **Bachelor of Engineering** in **Computer Engineering.**

Programme Outcomes	Grade
PO1, PO2, PO3, PO4, PO5, PO6, PO7, PO8, PO9, PO10, PO11, PO12 PSO1 & PSO2	

Date: 28th April, 2025

Project Guide: Dr. (Mrs.) Nupur Giri

Project Report Approval
For
B. E (Computer Engineering)

This project report entitled **"Optimizing Kafka-Based Supply Chain Architectures: A Comprehensive Performance Analysis"** by **Simran Ahuja (D17C - 02), Jesica Bijju (D17C - 10), Sejal Datir (D17C - 14), & Sania Khan (D17C - 36)** is approved for the degree of **Bachelor of Engineering** in **Computer Engineering.**

Examiners

1. ……

(Internal Examiner name & sign)

2. ……

(External Examiner name & sign)

3. ……

(Head of Department)

4. ……

(Principal)

Date: 28th April, 2025

Place: Chembur, Mumbai

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Simran Ahuja - D17C / 02)

(Jesica Bijju - D17C / 10)

(Sejal Datir - D17C / 14)

(Sania Khan - D17C / 36)

Date: 28th April, 2025

Acknowledgement

We are thankful to our college Vivekanand Education Society's Institute of Technology for considering our project and extending help at all stages needed during our work of collecting information regarding the project.

It gives us immense pleasure to express our deep and sincere gratitude to the Head of the Computer Department **Dr. (Mrs.) Nupur Giri** (Project Guide) for her kind help and valuable advice during the development of project synopsis and for her guidance and suggestions.

We are deeply indebted to our Principal **Dr. (Mrs.) J.M. Nair**, for giving us this valuable opportunity to do this project.

We express our hearty thanks to them for their assistance without which it would have been difficult to finish this project synopsis and project review successfully.

We convey our deep sense of gratitude to all teaching and non-teaching staff for their constant encouragement, support and selfless help throughout the project work. It is a great pleasure to acknowledge the help and suggestion, which we received from the Department of Computer Engineering.

We wish to express our profound thanks to all those who helped us in gathering information about the project. Our families too have provided moral support and encouragement several times.

COURSE OUTCOMES FOR B.E. PROJECT

Learners will be able to,

Course Outcome	Description of the Course Outcome
CO1	Able to apply the relevant engineering concepts, knowledge and skills towards the project.
CO2	Able to identify, formulate and interpret the various relevant research papers and to determine the problem.
CO3	Able to apply the engineering concepts towards designing solutions for the problem.
CO4	Able to interpret the data and datasets to be utilized.
CO5	Able to create, select and apply appropriate technologies, techniques, resources and tools for the project.
CO6	Able to apply ethical, professional policies and principles towards societal, environmental, safety and cultural benefit.
CO7	Able to function effectively as an individual, and as a member of a team, allocating roles with clear lines of responsibility and accountability.
CO8	Able to write effective reports, design documents and make effective presentations.
CO9	Able to apply engineering and management principles to the project as a team member.
CO10	Able to apply the project domain knowledge to sharpen one's competency.
CO11	Able to develop a professional, presentational, balanced and structured approach towards project development.
CO12	Able to adopt skills, languages, environment and platforms for creating innovative solutions for the project.

Index

List Of Figures

List Of Tables

Abstract

This study explores the impact of Apache Kafka parameter tuning on the performance and scalability of a microservices-based supply chain management system. It evaluates the effects of key Kafka parameters such as batch size, compression type, partition count, and replication factor on crucial performance metrics, including throughput, latency, and fault tolerance. The experimental results show that fine-tuning Kafka configurations can significantly reduce processing delays, enhance throughput, and improve system resilience under varying workloads.

The study highlights the trade-off between replication overhead and performance efficiency, particularly at high message volumes, where lower replication factors lead to better processing efficiency. By adjusting Kafka's parameters, the system's ability to handle large-scale event-driven workloads is optimized, resulting in more efficient data processing and reduced latency. Additionally, the evaluation of different parameter configurations provides insights into how Kafka can be tailored to meet the specific needs of supply chain systems.

The findings offer actionable recommendations for Kafka performance optimization, helping organizations improve the reliability and scalability of their event-driven supply chain systems. These guidelines are valuable for architects and developers aiming to maximize Kafka's potential while balancing performance and fault tolerance in large-scale applications.

Chapter 1: Introduction

1.1 Introduction

Supply Chain Management (SCM) is an essential aspect of modern business operations, responsible for the efficient and timely movement of goods from manufacturers to end consumers. Traditionally, SCM systems were built using monolithic architectures, where various core components such as order processing, inventory control, logistics, and tracking were tightly integrated into a single application. While this setup worked for simpler workflows, it has become increasingly inadequate for handling the scale and complexity of today's dynamic supply chains.

Monolithic architectures present several challenges in large-scale systems. They often experience performance issues when processing high transaction volumes, leading to degraded efficiency. A fault in one component can affect the entire application, causing system-wide failures. Moreover, scaling becomes inefficient, as the entire application needs to be replicated even if only one part of it demands more resources. This leads to unnecessary resource consumption and increased costs. These systems also lack the flexibility required to adapt to frequent business changes, making the integration of new features or third-party services difficult.

To address these limitations, modern SCM solutions are transitioning toward microservices-based architectures. In this project, the proposed system is composed of five loosely coupled microservices: Manufacturer, Hospital, Transport, Monitor, and Notification. Each of these services is containerized using Docker, enabling consistent deployment, better portability, and simplified management. The services communicate through Apache Kafka, a distributed event-streaming platform that supports fast and reliable message exchange.

Docker ensures that each microservice operates in its own isolated environment, enhancing fault isolation, ease of maintenance, and scalability. Kafka plays a crucial role in maintaining communication efficiency between services, but its performance is heavily influenced by specific configurations. Parameters such as batch size, partition count, and replication factor have a significant impact on throughput, latency, and system reliability.

Through the fine-tuning of these Kafka parameters, the system achieves faster message processing, reduced latency, and higher fault tolerance. Partitioning allows parallel message processing across brokers, improving scalability. Replication ensures data durability in case of node failures. Batch size tuning increases throughput by minimizing transmission overhead.

Overall, this microservices-based SCM system, built with Docker and optimized Kafka configurations, demonstrates enhanced performance, scalability, and resilience—making it a robust solution for modern supply chain challenges.

1.2 Motivation for the Project

In today's globalized and fast-paced market, supply chains have become increasingly complex, involving multiple stakeholders, real-time coordination, and the need for timely decision-making. Traditional supply chain management systems, which rely on monolithic architectures, struggle to keep up with the dynamic nature and growing demands of modern businesses. These systems face significant limitations in terms of scalability, fault tolerance, and adaptability, especially when dealing with high volumes of transactions or integrating new functionalities.

The motivation behind this project arises from the need to design a more efficient, reliable, and scalable solution for managing supply chain operations. By leveraging modern software architecture principles such as microservices and containerization, the project aims to overcome the limitations of traditional systems. Microservices allow independent deployment and scaling of different functional components, improving system resilience and maintainability.

Another key motivator is the integration of Apache Kafka, a distributed event streaming platform, which enables real-time communication between services. Kafka supports high-throughput, low-latency message processing, making it ideal for event-driven applications like supply chain systems. However, Kafka's performance is highly dependent on proper configuration of parameters such as batch size, partition count, and replication factor.

This project is driven by the objective to explore how tuning these parameters impacts system performance and to develop a containerized, event-driven SCM system that can handle real-world workloads efficiently. The goal is to provide a practical and scalable solution that enhances processing speed, ensures system reliability, and supports the growing demands of modern supply chains.

1.3. Drawback of the Existing System

Traditional Supply Chain Management (SCM) systems are predominantly built using monolithic architecture, where all core functionalities such as order management, inventory control, logistics, and tracking are tightly integrated into a single application. While this approach was suitable for earlier, less complex supply chain models, it presents several critical drawbacks in the context of modern, large-scale systems.

One of the primary limitations is **lack of scalability**. In a monolithic setup, scaling one module (e.g., inventory) requires scaling the entire application, leading to inefficient resource utilization and higher operational costs. This becomes problematic when only specific components need additional resources to handle high loads.

Another major drawback is **reduced fault tolerance**. A failure in any one component can potentially bring down the entire application, disrupting the supply chain operations. There is minimal fault isolation, which increases the risk of system-wide outages and reduces reliability.

Monolithic systems also suffer from **poor maintainability and limited flexibility**. Even small changes require rebuilding and redeploying the entire application, making it difficult to adapt quickly to business needs or integrate with third-party services. This slows down the development cycle and increases the chances of introducing bugs into the system.

Furthermore, these systems are **not well-suited for real-time processing**, which is essential in today’s event-driven environments. Traditional architectures struggle to handle large volumes of real-time data efficiently, leading to higher latency and delays in processing critical transactions.

Due to these limitations, existing monolithic SCM systems are becoming obsolete and inefficient for managing the evolving demands of modern supply chains. This highlights the need for a more modular, scalable, and fault-tolerant solution, which this project aims to address using microservices, Docker, and Apache Kafka.

1.4. Problem Definition

Drug Supply Chain Management (SCM) systems face mounting challenges in maintaining the efficiency, transparency, and resilience needed to support the complex demands of modern pharmaceutical distribution. Traditional monolithic SCM systems are ill-equipped to handle the scale and dynamism of today's interconnected networks, often resulting in bottlenecks, delayed communication, and increased operational costs.

These legacy systems suffer from:

- **Centralized architectures** that are prone to system-wide failures,
- **Limited scalability** that restricts performance under high transaction volumes,
- **Rigid structures** that hinder real-time responsiveness and modular upgrades.

To address these inefficiencies, there is a growing need for a modular, scalable, and fault-tolerant supply chain system that can support decentralized communication and real-time event synchronization across diverse stakeholders, such as manufacturers, hospitals, transport services, and monitoring agencies.

The proposed solution transitions from monolithic SCM to a Kafka-based microservices architecture to achieve the following:

- **Real-Time Event Streaming with Apache Kafka**: Enables asynchronous communication between independently developed services such as Manufacturer, Hospital, Transport, Monitor, and Notification, ensuring fast and reliable data exchange across the supply chain.
- **Microservices Modularity**: Breaks down the system into loosely coupled services, facilitating independent scaling, fault isolation, and easier maintenance.
- **Scalability and Fault Tolerance**: Optimizes key Kafka configurations like partition count, batch size, and replication factor to improve throughput, resilience, and overall system performance.
- **Docker Containerization**: Enhances deployment flexibility, fault isolation, and scalability by containerizing each microservice, allowing for seamless updates and efficient resource utilization.

By leveraging Kafka's distributed streaming capabilities and a microservices-based design, the system addresses the critical limitations of legacy SCM systems—enabling a robust, scalable, and efficient solution tailored to the evolving demands of modern drug supply chain operations.

1.5. Relevance of the Project

The Kafka-Based Supply Chain Management (SCM) project is highly relevant in today's interconnected and rapidly evolving pharmaceutical ecosystem. With increasing demand for real-time transparency, error-free logistics, and scalable infrastructure, this project addresses pressing challenges in healthcare supply chains, especially in the wake of global events such as pandemics and disruptions to critical drug deliveries.

By replacing traditional monolithic systems with a distributed, event-driven architecture, the project offers a highly efficient, resilient, and responsive solution for managing the end-to-end drug supply chain.

Key aspects that enhance the relevance of this project include:

- **Microservices Architecture for Scalability and Modularity**
  Each functional entity of the supply chain—such as Manufacturer, Hospital, Transport, Monitor, and Notification—is implemented as an independent microservice. This allows seamless scaling, efficient resource allocation, and fault isolation, making the system highly adaptable to growing demands.
- **Real-Time Communication with Apache Kafka**
  Apache Kafka enables asynchronous, high-throughput communication between microservices. It supports real-time event streaming, ensuring that updates on orders, inventory, and shipments are instantly synchronized across the network of stakeholders.
- **Scalability and Portability with Docker Containerization**
  Using Docker containers, each microservice can be deployed, scaled, and maintained independently across different environments. This supports faster updates, simplified testing, and cloud-native deployment capabilities.
- **Supply Chain Resilience and Visibility**
  With real-time tracking and communication, the system improves coordination among stakeholders, reduces stockouts and delays, and increases the overall resilience of the drug supply network.

Designed with future scalability and fault tolerance in mind, this architecture can be applied across a variety of sectors where logistics and multi-node communication are critical. It directly contributes to:

- **SDG 3:** Good Health and Well-being – By strengthening healthcare infrastructure through efficient drug delivery systems.
- **SDG 9:** Industry, Innovation, and Infrastructure – Promoting innovation in digital supply chain technologies and scalable IT infrastructure.
- **SDG 12:** Responsible Consumption and Production – Reducing waste and inefficiencies by optimizing inventory and distribution through real-time data analytics.

1.6. Methodology Used

The Kafka-Based Supply Chain Management project adopts a suite of modern technologies and architectural methodologies to ensure real-time synchronization, high scalability, and resilience across the drug supply chain. The system is designed to streamline the flow of data and communication between manufacturers, hospitals, transport systems, and monitoring services. The following methodologies form the backbone of the solution.

Microservices Architecture

The platform is designed using a modular microservices-based approach, where independent services such as **Manufacturer, Hospital, Transport, Monitor, and Notification** are loosely coupled and independently deployable. This enables fault isolation, system flexibility, and horizontal scalability across services without affecting the overall infrastructure.

Asynchronous Communication with Apache Kafka

Apache Kafka is used as a distributed event streaming platform, acting as the communication layer between microservices. Kafka ensures asynchronous, real-time data flow with high throughput and durability. Each microservice communicates by publishing and subscribing to Kafka topics, allowing seamless coordination and real-time updates across the supply chain.

Docker-Based Containerization

All services are packaged and deployed using Docker containers, promoting portability and consistency across development, testing, and production environments. Containerization also simplifies scalability and maintenance by allowing individual services to be updated or scaled without affecting others.

Kafka Configuration Tuning

The performance and resilience of the architecture are enhanced by optimizing Kafka parameters such as:

1. Partition Count – Enables parallel processing and load distribution.
2. Batch Size – Reduces message overhead and boosts throughput.
3. Replication Factor – Ensures data availability and fault tolerance during node failures.

These configurations are fine-tuned based on system load, throughput demands, and fault tolerance requirements.

Real-Time Monitoring and Logging

The system incorporates centralized logging and monitoring tools to track the health and performance of Kafka brokers and microservices. This helps in detecting anomalies, visualizing traffic patterns, and proactively managing failures.

Service-Oriented Task Handling

Each service performs clearly defined responsibilities:

1. Manufacturer Service: Publishes production and dispatch updates.
2. Transport Service: Tracks location and delivery status in real-time.
3. Hospital Service: Subscribes to inventory changes and places orders.
4. Monitor Service: Aggregates system status and performance metrics.
5. Notification Service: Sends alerts on shipment delays, inventory thresholds, and critical failures.

Security and Fault Isolation

By isolating services within containers and ensuring all communication flows through Kafka, the architecture is inherently fault-tolerant. Services can recover or restart independently in the event of failure, minimizing system-wide disruptions.

Chapter 2: Literature Survey

2.1 Research Papers

2.1.1 Microservices: yesterday, today, and tomorrow.

Authors: Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. - 2017

a. Abstract of the Research Paper

Microservices is an architectural style inspired by service-oriented computing that has recently started gaining popularity. Before presenting the current state-of-the-art in the field, this chapter reviews the history of software architecture, the reasons that led to the diffusion of objects and services first, and microservices later. Finally, open problems and future challenges are introduced. This survey primarily addresses newcomers to the discipline, while offering an academic viewpoint on the topic. In addition, we investigate some practical issues and point out some potential solutions.

b. Inference Drawn from the Paper

The chapter *"Microservices: Yesterday, Today, and Tomorrow"* provides a comprehensive analysis of the evolution and current state of microservices architecture. It traces the progression from monolithic systems to service-oriented architectures, culminating in the adoption of microservices. The authors highlight the advantages of microservices, such as enhanced flexibility, modularity, and scalability, while also acknowledging challenges like increased network complexity and security concerns. They emphasize the need for robust tools and methodologies to manage these complexities effectively. The chapter concludes by identifying open problems and future challenges, suggesting that ongoing research and development are essential to fully leverage the potential of microservices in software engineering.

2.1.2 Kafka: a Distributed Messaging System for Log Processing

Authors: Jay Kreps, Neha Narkhede, and Jun Rao

a. Abstract of the Research Paper

Log processing has become a critical component of the data pipeline for consumer internet companies. We introduce Kafka, a distributed messaging system that we developed for collecting and delivering high volumes of log data with low latency. Our system incorporates ideas from existing log aggregators and messaging systems, and is suitable for both offline and online message consumption. We made quite a few unconventional yet practical design choices in Kafka to make our system efficient and scalable. Our experimental results show that Kafka has superior performance when compared to two popular messaging systems. We have been using Kafka in production for some time and it is processing hundreds of gigabytes of new data each day.

b. Inference Drawn from the Paper

The paper *"Kafka: A Distributed Messaging System for Log Processing"* presents Kafka as a high-throughput, low-latency system designed to efficiently process large volumes of log data. Developed at LinkedIn, Kafka addresses the limitations of traditional log aggregation and messaging systems by offering a distributed, scalable architecture capable of handling real-time and

batch data processing. It achieves performance efficiency through sequential disk I/O, partitioned topics for parallel processing, and consumer groups that balance load across readers. Kafka also employs offset management to ensure reliable message delivery and replay capabilities. Overall, the system is positioned as a robust solution for modern data pipelines that demand speed, reliability, and scalability.

2.1.3 Development of Kafka Messaging System and its Performance Test Framework using Prometheus

Authors: Jay Kreps, Neha Narkhede, and Jun Rao

a. Abstract of the Research Paper

Log processing has become a critical component of the data pipeline for consumer internet companies. We introduce Kafka, a distributed messaging system that we developed for collecting and delivering high volumes of log data with low latency. Our system incorporates ideas from existing log aggregators and messaging systems, and is suitable for both offline and online message consumption. We made quite a few unconventional yet practical design choices in Kafka to make our system efficient and scalable. Our experimental results show that Kafka has superior performance when compared to two popular messaging systems. We have been using Kafka in production for some time and it is processing hundreds of gigabytes of new data each day.

b. Inference Drawn from the Paper

The paper *"Kafka: A Distributed Messaging System for Log Processing"* presents Kafka as a high-throughput, low-latency system designed to efficiently process large volumes of log data. Developed at LinkedIn, Kafka addresses the limitations of traditional log aggregation and messaging systems by offering a distributed, scalable architecture capable of handling real-time and batch data processing. It achieves performance efficiency through sequential disk I/O, partitioned topics for parallel processing, and consumer groups that balance load across readers. Kafka also employs offset management to ensure reliable message delivery and replay capabilities. Overall, the system is positioned as a robust solution for modern data pipelines that demand speed, reliability, and scalability.

2.1.4 Optimising Kafka for stream processing in latency sensitive systems

Authors: Roman Wiatr, Renata Słota, Jacek Kitowski

a. Abstract of the Research Paper

Many problems, like recommendation services, sensor networks, anti-crime protection, sophisticated AI services, need online data processing coming from the environment in the form of data streams consisting of events. The novelty of the approach in the field of stream processing lies in a synergistic effort toward optimization of such systems and additionally needed client components working as a whole. Building a message passing system for gathering information from mission-critical systems can be beneficial, but it is required to pay close attention to the impact it has on these systems. In this paper, we present the Apache Kafka optimization process for usage of Kafka as a messaging system in latency sensitive systems. We propose a set of performance tests that can be used to measure Kafka impact on the system and performance test results of KafkaProducer Java API. KafkaProducer has almost no impact on system overall latency and it has a severe impact on resource consumption in terms of CPU. Optimising Kafka for stream

processing in latency sensitive systems we reduce KafkaProducer negative impact by 75%. The tests are performed on an isolated production system.

b. Inference Drawn from the Paper

The paper titled *"Optimising Kafka for Stream Processing in Latency Sensitive Systems"* discusses the optimization of Apache Kafka for use as a messaging system in environments where low latency is critical. Apache Kafka is a distributed event streaming platform commonly used for building real-time data pipelines and streaming applications. In latency-sensitive systems, minimizing the time taken for data to travel through the system is essential to meet performance requirements. The paper outlines strategies and techniques for configuring and tuning Kafka to reduce latency, ensuring that messages are processed and delivered with minimal delay. These optimizations are crucial for applications where timely data processing is vital, such as financial services, telecommunications, and real-time analytics. By implementing the recommended optimizations, organizations can enhance the performance of their stream processing systems, achieving lower latency and improved responsiveness in their applications.

2.1.5 On Efficiently Partitioning a Topic in Apache Kafka

Authors: Theofanis P. Raptis, Andrea Passarella

a. Abstract of the Research Paper

Apache Kafka addresses the general problem of delivering extreme high volume event data to diverse consumers via a publish-subscribe messaging system. It uses partitions to scale a topic across many brokers for producers to write data in parallel, and also to facilitate parallel reading of consumers. Even though Apache Kafka provides some out of the box optimizations, it does not strictly define how each topic shall be efficiently distributed into partitions. The well-formulated fine-tuning that is needed in order to improve an Apache Kafka cluster performance is still an open research problem. In this paper, we first model the Apache Kafka topic partitioning process for a given topic. Then, given the set of brokers, constraints and application requirements on throughput, OS load, replication latency and unavailability, we formulate the optimization problem of finding how many partitions are needed and show that it is computationally intractable, being an integer program. Furthermore, we propose two simple, yet efficient heuristics to solve the problem: the first tries to minimize and the second to maximize the number of brokers used in the cluster. Finally, we evaluate its performance via large-scale simulations, considering as benchmarks some Apache Kafka cluster configuration recommendations provided by Microsoft and Confluent. We demonstrate that, unlike the recommendations, the proposed heuristics respect the hard constraints on replication latency and perform better w.r.t. unavailability time and OS load, using the system resources in a more prudent way.

b. Inference Drawn from the Paper

The paper titled *"On Efficiently Partitioning a Topic in Apache Kafka"* by Theofanis P. Raptis and Andrea Passarella addresses the challenge of optimizing topic partitioning within Apache Kafka clusters. While Kafka allows for extensive partitioning, it doesn't provide explicit guidelines on efficient partition distribution, which is crucial for performance. The authors model the partitioning process and formulate an optimization problem to determine the optimal number of partitions, considering factors like throughput, OS load, replication latency, and system availability. They demonstrate that this problem is computationally intractable. To address this, they propose two heuristics: one minimizing and the other maximizing broker usage. Through large-scale

simulations, these heuristics are shown to better adhere to constraints and outperform existing recommendations from Microsoft and Confluent in terms of replication latency, unavailability time, and OS load, leading to more efficient resource utilization.

2.1.6 How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka

Authors: Theofanis P. Raptis, Andrea Passarella

a. Abstract of the Research Paper

Message brokers see widespread adoption in modern IT landscapes, with Apache Kafka being one of the most employed platforms. These systems feature well-defined APIs for use and configuration and present flexible solutions for various data storage scenarios. Their ability to scale horizontally enables users to adapt to growing data volumes and changing environments. However, one of the main challenges concerning message brokers is the danger of them becoming a bottleneck within an IT architecture. To prevent this, knowledge about the amount of data a message broker using a specific configuration can handle needs to be available. In this paper, we propose a monitoring architecture for message brokers and similar Java Virtual Machine-based systems. We present a comprehensive performance analysis of the popular Apache Kafka platform using our approach. As part of the benchmark, we study selected data ingestion scenarios with respect to their maximum data ingestion rates. The results show that we can achieve an ingestion rate of about 420,000 messages/second on the used commodity hardware and with the developed data sender tool.

b. Inference Drawn from the Paper

The paper titled *"How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka"* by Guenter Hesse, Christoph Matthies, and Matthias Uflacker presents a comprehensive analysis of Apache Kafka's data ingestion capabilities. The authors developed a monitoring architecture tailored for Java Virtual Machine-based systems to facilitate this evaluation. Their findings indicate that, on commodity hardware, Apache Kafka can achieve ingestion rates of approximately 420,000 messages per second. This performance was observed under specific configurations, including the use of a custom data sender tool. The study underscores the importance of system configuration and hardware capabilities in optimizing Kafka's performance for high-throughput scenarios.

2.1.7 Designing High-performance, Scalable Kafka Clusters for Real-time Data Streaming

Authors: Chandrakanth Lekkala

a. Abstract of the Research Paper

Due to exponential evolution of data from many sources, real-time data streaming is necessary for many organizations. Apache Kafka, an open-source, highly available distributed event streaming platform, has become famous for developing streaming data pipelines in less than time. Nevertheless, a complex, high-performance, scalable Kafka cluster construction is not easy because of large volumes of big data and the quality of significant performers' requirements. In this paper, I will establish the main aspects and top practices for creating and utilizing Kafka clusters that can be applied with low latency data streaming at big-time scales. The topic of this lecture contains the constituent elements of cluster architecture, data partitioning, replication, and configuration tuning.

Besides, the paper encompasses Kafka performance optimization strategies, fault tolerance techniques, and ways to integrate them into other big data technology frameworks. By industry organizations following the proposals presented in this paper, the high availability, scalability, and throughput performance of Kafka clusters can be achieved for real-time data streaming.

b. Inference Drawn from the Paper

In *"Designing High-performance, Scalable Kafka Clusters for Real-time Data Streaming,"* Chandrakanth Lekkala outlines essential practices for constructing Apache Kafka clusters optimized for high availability, scalability, and low-latency data streaming. The paper delves into critical components such as cluster architecture, data partitioning, replication, and configuration tuning. Additionally, it discusses performance optimization strategies, fault tolerance techniques, and integration methods with other big data frameworks. By implementing these best practices, organizations can effectively manage real-time data streaming at large scales.

2.2 Articles Referred

2.2.1 Docker: lightweight Linux containers for consistent development and deployment Linux Journal, 2014.

Authors: Dirk Merkel

a. Summary of the Article

This article introduces Docker as a transformative tool in the realm of software development and deployment. Docker enables developers to package applications along with their dependencies into isolated containers, ensuring consistent behavior across various environments. The author delves into the architecture of Docker, highlighting its use of Linux kernel features like namespaces and control groups (cgroups) to achieve process isolation and resource management. By leveraging these features, Docker containers can run multiple applications on the same host without interference, leading to efficient resource utilization. The article also discusses the advantages of Docker over traditional virtual machines, emphasizing its lightweight nature, faster startup times, and ease of integration into continuous integration and deployment pipelines.

b. Inference Drawn from the Article

From the article, it can be inferred that Docker significantly streamlines the development-to-deployment workflow by providing a consistent environment for applications. This consistency reduces the "it works on my machine" problem, as the same container can be deployed across different stages of development and production without modification.Furthermore, Docker's lightweight containers offer a more efficient alternative to full-fledged virtual machines, leading to better resource utilization and scalability. The adoption of Docker can thus lead to improved developer productivity, faster deployment cycles, and more reliable application performance across diverse environments.

2.2.2 Benchmarking Apache Kafka: 2 million writes per second on three cheap machines LinkedIn Engineering Blog, 2016.

Authors: Jay Kreps et al.

a. Summary of the Article

In his benchmarking study, Jay Kreps evaluates Apache Kafka's performance using a cluster of three general-purpose machines, each equipped with a 2.5 GHz Xeon processor, 32GB RAM, six 7200 RPM SATA drives, and a 1Gb Ethernet connection. The tests reveal Kafka's impressive throughput capabilities: a single producer thread without replication achieves approximately 821,557 records per second (78.3 MB/sec); introducing 3x asynchronous replication slightly reduces throughput to around 786,980 records per second (75.1 MB/sec); and 3x synchronous replication yields about 421,823 records per second (40.2 MB/sec). Scaling up to three producers with 3x asynchronous replication, Kafka handles over 2 million records per second (193.0 MB/sec). On the consumption side, three consumers manage to process approximately 2.6 million records per second (249.5 MB/sec), and combined producer-consumer operations sustain around 795,064 records per second (75.8 MB/sec). These results underscore Kafka's ability to maintain high throughput rates even with increased replication factors and concurrent producer-consumer operations.

b. Inference Drawn from the Article

The benchmark results indicate that Apache Kafka is highly efficient and scalable, capable of handling millions of messages per second using relatively inexpensive hardware. The system's architecture allows for consistent performance across various configurations, including different replication strategies and concurrent processing scenarios. This efficiency makes Kafka a suitable choice for organizations seeking a reliable and cost-effective solution for real-time data streaming and processing needs.

2.2.3 How to choose the number of topics/partitions in a Kafka cluster?, March 2015.

Authors: J. Rao.

a. Summary of the Article

In this article, Jun Rao emphasizes that partitions are the fundamental unit of parallelism in Kafka. Increasing the number of partitions can enhance throughput by enabling parallel processing on both the producer and consumer sides. To estimate the required number of partitions, the article suggests measuring the throughput achievable per partition for both production (p) and consumption (c), then calculating the maximum of t/p and t/c, where t represents the target throughput. This calculation helps determine the minimum number of partitions needed to meet performance goals.

The article also discusses the implications of using keys in messages. When messages are keyed, Kafka ensures that all messages with the same key are routed to the same partition, preserving their order. However, increasing the number of partitions after deployment can disrupt this ordering guarantee. To mitigate this risk, it's advisable to initially over-partition based on anticipated future throughput, allowing for scalability without compromising message ordering.

Additionally, the article warns against excessive partitioning, as it can lead to increased overhead and resource consumption. Therefore, while partitioning is a powerful tool for scaling Kafka, it must be applied judiciously to balance performance gains against potential drawbacks.

b. Inference Drawn from the Article

The article underscores the importance of strategic partitioning in Kafka to achieve desired throughput while maintaining system integrity. By carefully estimating the number of partitions based on throughput requirements and considering the impact of message keys on partitioning, one can design a Kafka cluster that is both efficient and scalable. Over-partitioning in anticipation of future growth can provide flexibility, but it must be balanced against the potential for increased system overhead. Thus, thoughtful planning and measurement are crucial in configuring Kafka partitions to meet both current and future demands.

2.2.4 Cloudera Inc. Tuning Apache Kafka Performance, 2024.

a. Summary of the Article

The Cloudera documentation titled "Tuning Apache Kafka Performance" offers a comprehensive guide to optimizing Kafka's performance through various configurations and system-level adjustments. It emphasizes the importance of appropriately sizing clusters based on network and disk throughput requirements, determining the optimal number of partitions to achieve desired parallelism, and tuning broker settings such as JVM configurations, garbage collection, and log cleaner parameters. Additionally, it highlights the necessity of system-level tuning, including adjusting file descriptor limits, selecting suitable filesystems, managing virtual memory settings, and configuring networking parameters. The document also addresses the handling of large messages by recommending message size reductions or configuring broker and consumer properties to accommodate larger messages. Overall, the guide underscores that achieving optimal Kafka performance requires a holistic approach that considers both application-level configurations and underlying system settings.

b. Inference Drawn from the Article

The document underscores that achieving optimal Kafka performance requires a holistic approach, encompassing both application-level configurations and underlying system settings. By meticulously tuning broker parameters and ensuring the operating system is configured to support Kafka's demands, organizations can significantly enhance throughput, reduce latency, and maintain system stability. This comprehensive tuning strategy is essential for deploying Kafka in production environments where performance and reliability are critical.

Author	Title	Description
Peddireddy, Kiran, IARJSET (2023)	Enhancing Inventory Management Systems through Kafka Data Stream Integration	Integration of Apache Kafka into inventory management systems enhances real-time data streaming, improves supplier communication, increases operational efficiency, ensures scalability and reliability, and reduces costs and risks associated with inventory mismanagement

V Ganesh Tejas, Dr. Hemavathy, IRJET(2020)	Development of Kafka Messaging System and its Performance Test Framework using Prometheus	This paper suggests a method for developing, deploying, and securing a Kafka-based messaging system using container technology. The study assesses Kafka's efficiency by varying the number of records produced and consumed, with test results demonstrating system performance under different conditions.
J. Rao. March 2015, Accessed 06-April-2022	How to choose the number of topics/partitions in a kafka cluster?	This article suggests that increasing partitions in a Kafka cluster improves throughput by enabling parallelism but also increases resource usage, latency, and potential unavailability during failures. Proper partitioning strategies balance performance, scalability, and system limitations to optimize Kafka's efficiency.
Roman Wiatr, Renata Słota, Jacek Kitowski, ScienceDirect(2018)	Optimising Kafka for stream processing in latency sensitive systems	The paper presents an optimized approach to using Apache Kafka for stream processing in latency-sensitive systems, introducing performance tests to measure Kafka's impact. The study identifies KafkaProducer's high CPU consumption as a challenge and proposes optimization techniques that reduce its negative impact by 75%, improving resource efficiency while maintaining low latency.
Theofanis P. Raptis and Andrea Passarella, IIT-CNR(2022)	On Efficiently Partitioning a Topic in Apache Kafka	The paper presents a novel approach to optimizing Apache Kafka topic partitioning by modeling the process, formulating it as an optimization problem, and proposing two heuristic algorithms that improve resource efficiency, reduce replication latency, and enhance fault tolerance compared to existing industry recommendations.

Table 2.1: Summary of Literature Survey

2.3 Inference Drawn

The literature reviewed in this chapter highlights the evolution and growing significance of microservices architecture and Apache Kafka in the development of modern distributed systems. From the foundational understanding of microservices presented in *"Microservices: Yesterday, Today, and Tomorrow"*, we observe that modularity, scalability, and flexibility are core benefits that continue to drive widespread adoption in industry applications. However, they also come with new challenges such as increased system complexity and the need for more sophisticated orchestration tools.

Apache Kafka emerges as a recurring solution to the challenges of real-time data streaming, log aggregation, and reliable message delivery. The various studies examined delve into Kafka's architecture, performance benchmarks, optimization strategies, and scalability. Notably:

- Kafka's low-latency, high-throughput capabilities make it ideal for handling real-time event streams and log processing.
- Several papers focus on performance optimization in latency-sensitive environments, highlighting the impact of producers, brokers, and system resource allocation.
- Research on topic partitioning and efficient cluster design reveals that Kafka's performance heavily relies on system configuration and partition strategies.
- Benchmarks indicate that Kafka can support ingestion rates of over 400,000 messages per second, demonstrating its capacity for high-volume workloads.

Together, these insights inform the development of scalable and resilient real-time systems. They also underscore the importance of customized Kafka tuning and careful architectural decisions for specific use cases, such as healthcare, e-commerce, or financial services. This comprehensive understanding will guide the design and implementation of the proposed system in the forthcoming chapters.

2.4 Comparison with the existing system

Traditional monolithic systems, which form the foundation of many existing architectures, exhibit several limitations in scalability, maintainability, and flexibility. In contrast, the microservices-based architecture proposed in recent literature offers significant advantages in addressing these issues.

The following table outlines a comparative analysis of key features between traditional monolithic systems and the proposed microservices-based system:

Feature	Existing Monolithic System	Proposed Microservices-Based System
Architecture	Tightly coupled, single-tiered	Loosely coupled, modular services
Scalability	Difficult to scale specific components independently	Individual services can be scaled based on load
Maintainability	Complex and time-consuming updates	Easier maintenance with isolated service updates

Deployment	Entire application must be redeployed	Continuous deployment of independent services
Failure Impact	Failure in one module can affect the whole system	Failures are isolated to individual services
Technology Stack	Limited to one tech stack	Allows polyglot programming (multiple tech stacks)
Data Communication	Synchronous, often tightly integrated	Asynchronous communication using Kafka
Real-time Data Handling	Minimal or absent	Real-time processing enabled via Kafka streaming

Table 2.2 Comparison of existing systems

One of the key enablers of this transition is the adoption of Apache Kafka as the backbone of asynchronous communication. Kafka acts as a distributed event streaming platform, decoupling the services from one another while ensuring high-throughput, low-latency, and fault-tolerant data exchange. This is particularly advantageous in scenarios that require real-time data ingestion and processing—such as in supply chain tracking, drug inventory monitoring, and alert systems.

Kafka enables services to communicate through publish-subscribe mechanisms, eliminating the need for services to be directly aware of each other. This not only promotes service independence but also improves the system's ability to gracefully handle failures. For example, if a downstream service is temporarily unavailable, Kafka can retain the events and ensure delivery once the service is back online.

Moreover, this architecture facilitates continuous integration and continuous delivery (CI/CD) pipelines, allowing teams to release new features or bug fixes independently and frequently without impacting other parts of the system. This rapid delivery cycle is essential for meeting the evolving needs of users and responding to market dynamics in a timely manner.

In summary, the proposed microservices-based system represents a paradigm shift from rigid, monolithic design to a more dynamic and scalable architecture. By combining the modular nature of microservices with the robustness of Kafka's event-streaming capabilities, the system becomes more resilient, responsive, and easier to evolve. This transition is not just a technical improvement but a strategic necessity for developing modern, cloud-native applications that demand real-time insights, high availability, and agility.

Chapter 3: Requirement Gathering for the Proposed System

3.1 Introduction to requirement gathering

Requirement gathering is a critical phase in the system development life cycle, aimed at understanding the precise needs of all stakeholders involved in the Drug Inventory and Supply Chain Tracking System. This process involves identifying both functional and non-functional aspects necessary for a robust, scalable, and real-time system powered by Kafka and microservices architecture.

The goal is to streamline drug distribution by enabling secure, transparent, and efficient interactions among hospitals, manufacturers, transport providers, and monitoring services. The system leverages event-driven communication through Kafka topics to ensure real-time data flow and seamless integration between distributed components. Through interviews, brainstorming sessions, and analysis of similar systems, the following requirements have been identified.

3.2 Functional Requirements

1. **Order Placement & Processing:** Hospitals place orders via Kafka (`order-placed` topic). Manufacturers process these orders, update inventory, and send confirmations.
2. **Drug Manufacturing and Inventory Management:** Manufacturers produce drugs and update multiple Kafka topics for inventory and confirmation.
3. **Transport & Shipment Tracking:** Transport service updates shipment status via Kafka (`shipment-status` topic).
4. **Monitoring System Health:** Monitor service publishes alerts based on anomalies and Kafka metrics (`monitor-alerts` topic).
5. **Real-Time Notification:** Notification service consumes messages from various topics and sends updates to users.

3.3 Non-Functional Requirements

1. **Scalability:** System should efficiently scale with message volume via Kafka's partitioning and multiple brokers.
2. **Low Latency:** Event processing must be near real-time for responsive supply chain operations.
3. **Fault Tolerance:** Use of replication in Kafka ensures resilience against broker failures.
4. **Reliability & Consistency:** Key-based partitioning helps maintain message order and data integrity.
5. **Maintainability:** Services are loosely coupled and containerized for easy updates and debugging.
6. **Performance Monitoring:** Monitor service tracks Kafka metrics like throughput and jitter.

3.4 Technical Constraints:

- **Kafka-Based Messaging System**: All inter-service communication must be handled using Apache Kafka, which introduces dependencies on Kafka cluster setup, partitioning strategy, and message serialization formats (e.g., Avro, JSON).
- **Microservices Architecture**: Each component (e.g., order service, transport service, notification service) must be independently deployable, containerized (preferably using Docker), and orchestrated (e.g., via Kubernetes).
- **Data Format and Schema Enforcement**: All messages published to Kafka must conform to predefined schemas to ensure interoperability between services. Schema Registry may be used for validation.
- **Security and Authentication**: Communication between services must be secured using SSL/TLS, and Kafka access must be authenticated and authorized using mechanisms like SASL or OAuth.
- **Dependency on External Systems**: Integration with third-party APIs (e.g., real-time GPS for shipment tracking or external monitoring tools) introduces network and compatibility constraints.

3.5 Hardware, Software, Tools, and Techniques Utilized Hardware

1. Hardware Constraints

The system was developed and tested on a **personal computing device**, which imposes limits on the scale of testing.

a) **Host Machine Used:**
   - **Model:** MacBook Pro
   - **Processor:** Apple M1 chip (ARM-based architecture)
   - **Memory (RAM):** 16 GB
   - **Storage:** 256 GB SSD

b) **Implications:**
   - While the M1 chip is powerful for development and simulation, it's **not equivalent to a production-grade server**.
   - The **number of Docker containers and Kafka brokers** that could be run concurrently was limited by available RAM and CPU.
   - Simulated workloads (up to 50,000 messages) are **controlled environments** and may not reflect performance on high-traffic enterprise systems.

2. Software Constraints

The system relied on specific open-source tools and frameworks. Each of these comes with limitations and setup requirements:

a) **Apache Kafka:** Apache Kafka is a powerful distributed event streaming platform known for its high throughput, durability, and scalability. It is well-suited for handling large

volumes of real-time data. However, to achieve optimal performance, manual tuning is often necessary—especially when configuring partitioning strategies and replication factors. The default settings are not ideal for high-performance production environments. Additionally, Kafka relies on Zookeeper for coordination tasks, which introduces an additional layer of complexity and dependency.

b) **Zookeeper:** Zookeeper plays a crucial role in the Kafka ecosystem by managing broker metadata and facilitating leader election among brokers. While it's possible to use a single Zookeeper instance for testing purposes, doing so introduces a single point of failure. To ensure high availability and resilience in production, deploying a Zookeeper cluster is strongly recommended.

c) **Docker:** Docker was utilized to containerize the various microservices involved—namely Hospital, Manufacturer, Transport, Monitor, and Notification services. This approach streamlines deployment and isolation. However, careful attention had to be given to container resource limits to prevent crashes or performance throttling. Furthermore, on Mac systems, Docker operates inside a Linux virtual machine, which can lead to slightly reduced I/O performance compared to running natively on a Linux environment.

d) **Python Workload Generator:** A Python-based workload generator was developed to simulate real-world messaging activity across the system. This script was effective for benchmarking and stress testing, generating synthetic data with fixed formats and patterns. However, it fell short of emulating real-time API variability and dynamic payload structures, which are common in actual production environments.

Chapter 4: Proposed Design

4.1 Architecture Diagram of the System

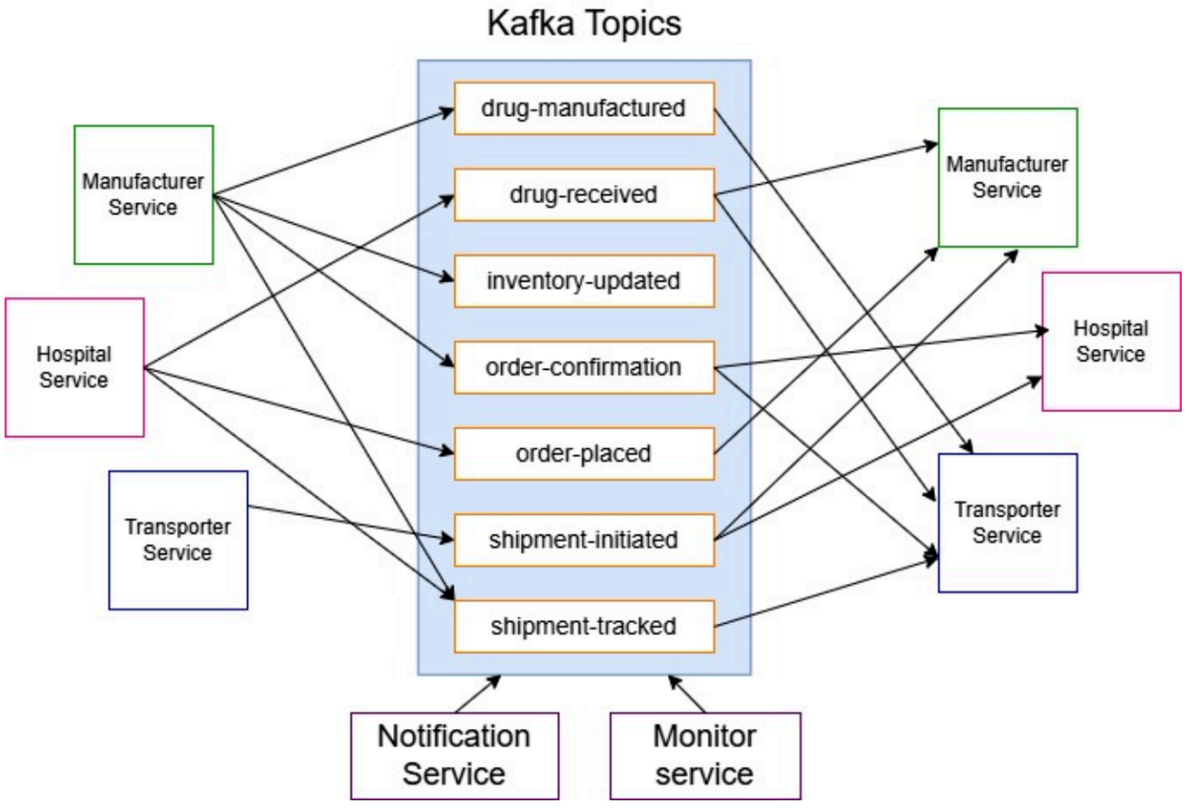


Figure 4.1: Kafka Architecture

The architecture diagram illustrates a **Kafka event-driven microservices** system for managing pharmaceutical supply chains. The system consists of several domain-specific services that communicate through Apache Kafka topics. Kafka acts as the central message broker, facilitating decoupled and asynchronous communication among services.

Core Microservices and Their Responsibilities

In the proposed Drug Inventory and Supply Chain Tracking System, microservices architecture plays a pivotal role in ensuring modularity, scalability, and maintainability. Each microservice is designed to fulfill a specific set of responsibilities while interacting with other services via Kafka, an event-streaming platform that acts as the communication backbone. The system's efficiency lies in how these services operate both independently and collaboratively to ensure seamless end-to-end drug supply chain management.

a. Hospital Service

The Hospital Service serves as the primary entry point into the system. It represents the interface through which healthcare institutions, such as hospitals or medical facilities, initiate drug orders and track their fulfillment. In its producer role, the service publishes an `order-placed` message whenever a new order is created. As the drugs are transported, it may also produce a `shipment-tracked` message to keep a real-time check on the consignment. Once the drugs are successfully delivered and verified, the hospital service can optionally produce a `drug-received` message to acknowledge the completion of the transaction.

On the consumer side, the Hospital Service listens to the `order-confirmation` topic to know whether the manufacturer has accepted and started processing the order. Additionally, it subscribes to the `shipment-initiated` topic to receive real-time updates on the shipment status. By doing so, hospitals can track their orders from initiation to final delivery, ensuring visibility throughout the process.

b. Manufacturer Service

The Manufacturer Service is central to order processing and inventory management. Upon receiving new drug requests, this service handles production, manages stock levels, and ensures that the order is fulfilled. In its consumer capacity, the service subscribes to the `order-placed` topic to receive incoming orders directly from hospitals.

Acting as a producer, the Manufacturer Service is responsible for updating multiple Kafka topics during the order lifecycle. Once the order is accepted, it publishes an `order-confirmation` message. As drugs are manufactured, it sends out `drug-manufactured` messages, followed by `inventory-updated` messages to reflect the change in stock levels. For additional confirmation or traceability, it can publish a `drug-received` message and provide shipping details through the `shipment-tracked` topic to aid in transport coordination and monitoring.

c. Transporter Service

The Transporter Service is tasked with the physical movement of drugs from the manufacturer to the hospital or medical institution. It plays a crucial role in logistics and shipment tracking. The service consumes messages from the `drug-manufactured` topic to identify when a drug batch is ready for pickup and delivery.

In its producer role, the Transporter Service emits `shipment-initiated` events once the shipment process begins. It continuously provides real-time updates through `shipment-tracked` messages, which include GPS location data and timestamps. After successfully delivering the drugs to the hospital, the service sends out a `drug-received` message as a final confirmation, ensuring the supply chain process is documented from end to end.

d. Notification Service

The Notification Service is responsible for enhancing the user experience by delivering timely and relevant updates to users across the system. It functions solely as a consumer, subscribing to multiple Kafka topics such as `order-confirmation`, `shipment-initiated`, and `shipment-tracked`. This allows the service to capture key moments in the drug delivery journey.

The service then transforms these events into user-facing notifications, which can be displayed through dashboards, web portals, or mobile apps. Technologies such as WebSocket or push notification APIs are used to relay these updates instantly, enabling users—whether hospitals, distributors, or regulatory authorities—to stay informed and make timely decisions.

e. Monitor Service

The Monitor Service ensures system transparency and operational health by observing all activities across the Kafka ecosystem. It functions exclusively as a consumer, subscribing to all major Kafka

topics to achieve full observability. This service plays a vital role in logging, auditing, and performance monitoring.

Beyond basic monitoring, the Monitor Service supports advanced analytics and incident detection. It captures data for throughput, message delays, error patterns, and can even integrate AI/ML algorithms to predict anomalies or optimize performance. This ensures that the system remains reliable, fault-tolerant, and capable of scaling as demand grows.

4.2: Project Scheduling & Tracking using Timeline / Gantt Chart

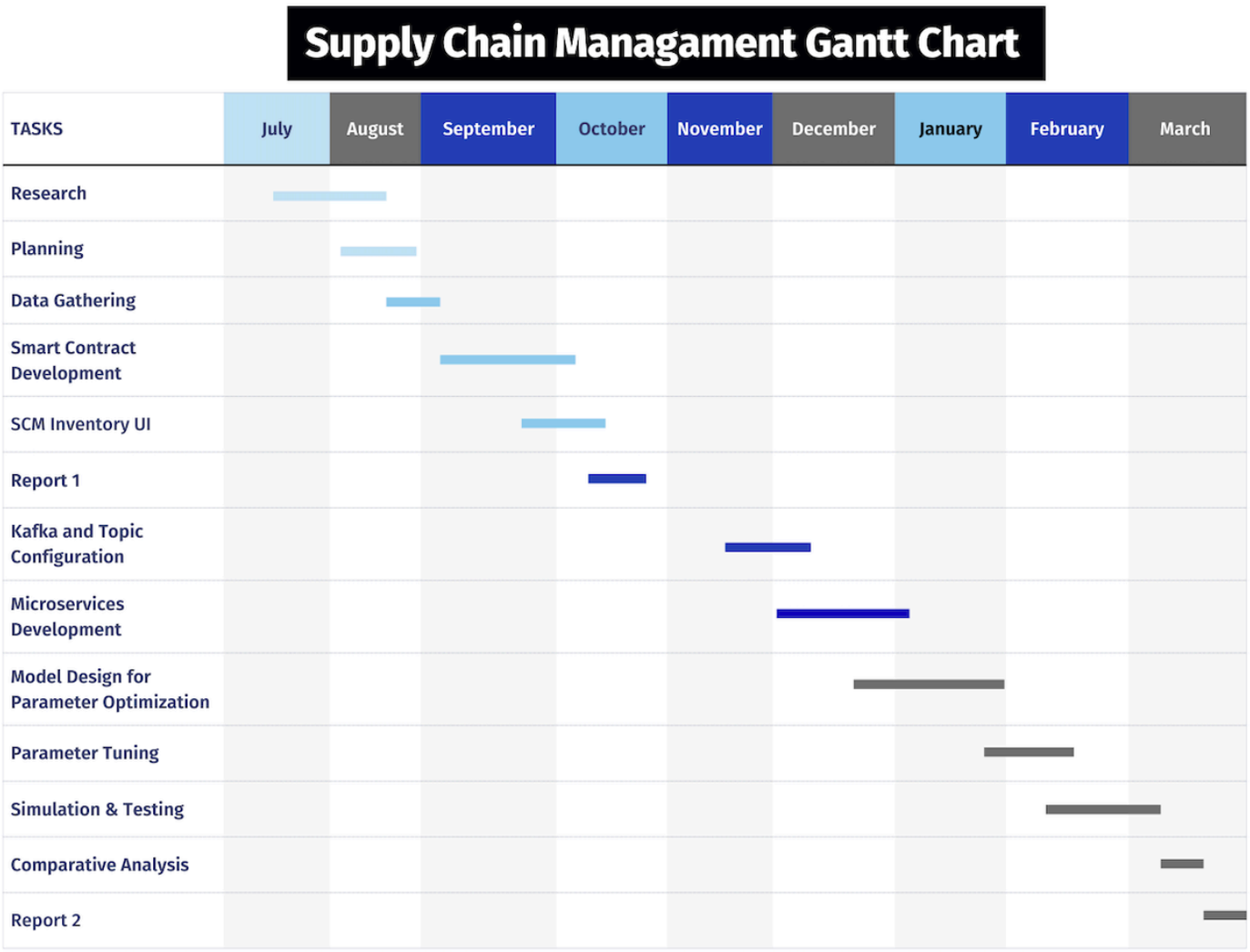


Figure 4.2: Gantt Chart of the system

Chapter 5: Implementation of the Proposed System

5.1. Methodology employed for development

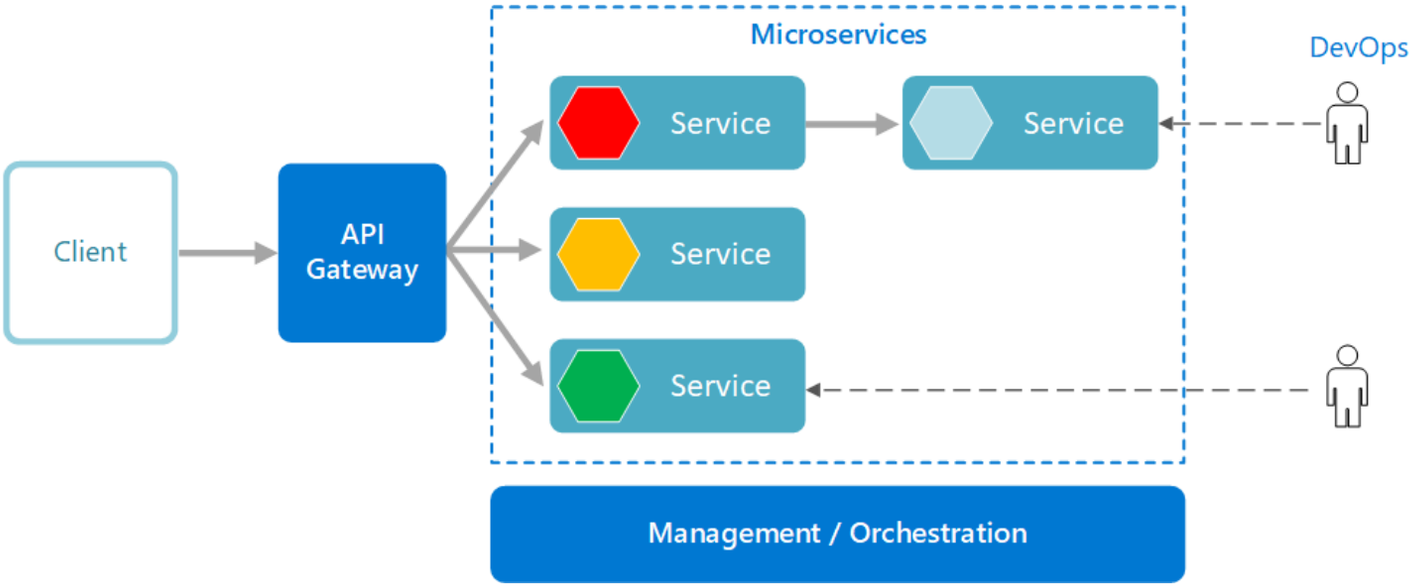


Figure 5.1: Microservice Architecture

Microservices-based architecture has been employed to develop and optimize a scalable, resilient, and maintainable supply chain platform. The methodology revolves around the decomposition of application functionality into discrete, independently deployable services, integrated through an API Gateway and governed by management/orchestration tools. This approach aligns with modern DevOps practices, ensuring continuous integration, automated deployment, and operational transparency.

This section provides a structured explanation of the methodology used, illustrated in Figure 5.1, which depicts the architectural flow from client interaction to service orchestration and DevOps integration.

1. Client Interface

The client represents the external user or system (e.g., mobile application, web portal, or enterprise software) that interacts with the backend ecosystem. Clients are abstracted from the internal complexity of service interactions and only interface with the system via the API Gateway. This improves usability, security, and maintainability.

2. API Gateway

The API Gateway acts as the single entry point for all client requests. It plays a crucial role in the following:

- Routing: Directs incoming requests to the appropriate microservice.
- Authentication & Authorization: Validates user credentials before request processing.
- Load Balancing: Distributes incoming requests to multiple instances of a microservice.
- Rate Limiting & Monitoring: Protects services from overload and provides analytics.

- Aggregation: In scenarios where data from multiple services is required, the gateway can aggregate results before sending a unified response to the client.

The use of an API Gateway abstracts the internal structure of services and enforces consistent communication standards.

3. Microservices Layer

Each service in the architecture encapsulates a distinct business function, such as inventory management, order processing, or shipment tracking. The diagram shows three such microservices, each represented with a colored hexagon (red, yellow, green), signifying modularity and potential service status indicators (e.g., warning, active, or standby).

Key characteristics of microservices in this methodology:

- Independently Deployable: Services can be updated or replaced without affecting others.
- Technology Agnostic: Teams can choose the most suitable tech stack per service.
- Scalable: Individual services can be scaled horizontally based on demand.
- Composable: Services can invoke other services as needed (e.g., the red service delegates to another internal service), forming functional chains.

This layer follows the Separation of Concerns principle, reducing complexity and promoting reusability.

4. Management / Orchestration

At the foundation of the microservices architecture lies a Management and Orchestration Layer (e.g., Kubernetes, Docker Swarm). Its responsibilities include:

- Container Management: Deploying and managing service containers.
- Service Discovery: Automatically locating service instances for communication.
- Health Monitoring: Proactively identifying and responding to service failures.
- Resource Allocation: Dynamically managing compute/storage resources for performance optimization.
- Auto-scaling: Adjusting service instances based on real-time load metrics.

This orchestration layer ensures that the microservices operate as a cohesive system despite their independent natures.

5. DevOps Integration

Each microservice is tightly integrated with DevOps pipelines, as depicted by the connections to DevOps personnel. This is critical for:

- Continuous Integration (CI): Automatically building and testing code upon changes.
- Continuous Deployment (CD): Automated deployment of changes into production.
- Logging and Monitoring: Centralized log management and real-time observability tools (e.g., ELK stack, Prometheus + Grafana).
- Rollback & Recovery: Rapid recovery from faulty deployments or service failures.

5.2 Algorithms and flowcharts for the system developed

a. Flowchart of system

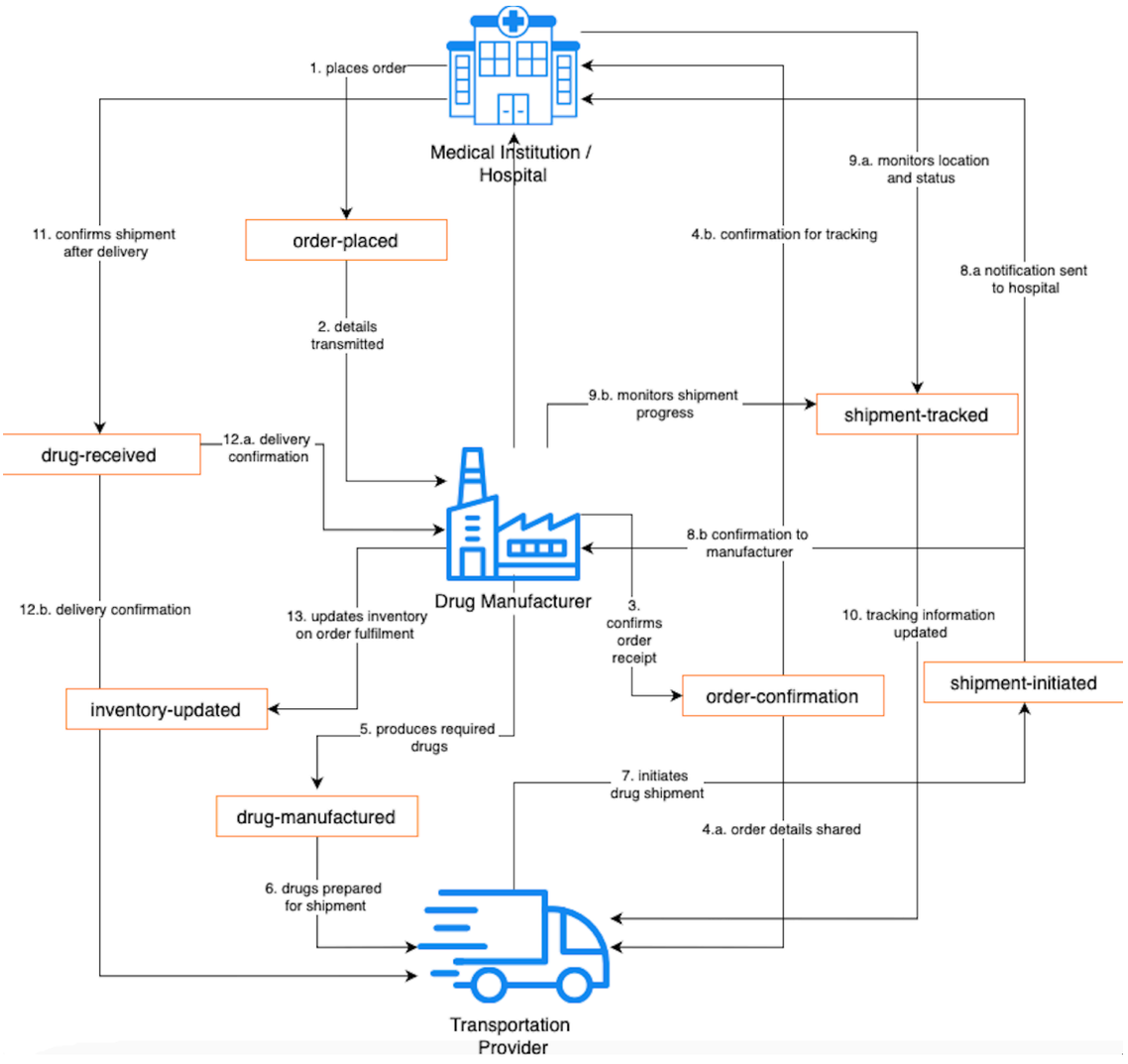


Figure 5.2: Flow of system

This diagram illustrates the complete workflow of a pharmaceutical supply chain system involving three key entities: Medical Institution/Hospital, Drug Manufacturer, and Transportation Provider. The diagram maps out information flows, events, and interactions between these entities through a series of numbered steps.

Order Initiation and Processing

1. Place Order: The hospital initiates the process by placing an order for required medications.
2. Details Transmitted: The order details (order-placed) are transmitted to the drug manufacturer.
3. Confirms Order Receipt: The manufacturer confirms receipt of the order.

Order Confirmation and Sharing

4.a. Order Details Shared: Order confirmation details are shared with the transportation provider.

4.b. Confirmation for Tracking: Order confirmation is sent to the hospital for tracking purposes.

Manufacturing Phase

5. Produces Required Drugs: The manufacturer produces the medications requested in the order.
6. Drugs Prepared for Shipment: The manufactured drugs are prepared and packaged for shipment.

Shipment Phase

7. Initiates Drug Shipment: The transportation provider initiates the shipment of drugs.

8.a. Notification Sent to Hospital: Shipment initiation notification is sent to the hospital.

8.b. Confirmation to Manufacturer: Shipment initiation confirmation is sent to the manufacturer.

Tracking Phase

9.a. Monitors Location and Status: The hospital monitors the shipment's location and status.

9.b. Monitors Shipment Progress: The manufacturer also monitors the shipment's progress.

10. Tracking Information Updated: The transportation provider updates tracking information throughout the delivery process.

Delivery and Confirmation

11. Confirms Shipment After Delivery: The hospital confirms the shipment after receiving the delivery.

12.a. Delivery Confirmation: The hospital sends delivery confirmation to the manufacturer.

12.b. Delivery Confirmation: The hospital also sends delivery confirmation to the transportation provider.

Inventory Management

13. Updates Inventory on Order Fulfillment: The manufacturer updates its inventory based on the completed order fulfillment.

5.3 Datasets source and utilization

In this project, synthetic datasets were generated to simulate real-world supply chain activities across five microservices: Manufacturer, Hospital, Transport, Monitor, and Notification. These datasets were structured to reflect typical supply chain workflows such as order creation, dispatch updates, shipment tracking, inventory changes, and alert notifications.

The messages exchanged between these microservices were generated and injected into Kafka topics using mock data generators. Each message followed a predefined JSON schema to ensure consistency across producers and consumers.

Each dataset/message included:

- Timestamp of the event
- Entity details (e.g., hospital ID, manufacturer ID, transport ID)
- Action/Event type (e.g., order-placed, drug-manufactured, etc)
- Status flags (e.g., success, failure)
- Optional metadata (e.g., GPS coordinates, temperature logs, delivery personnel)

These messages were transmitted in real-time between services through Kafka topics, simulating a production-like data flow.

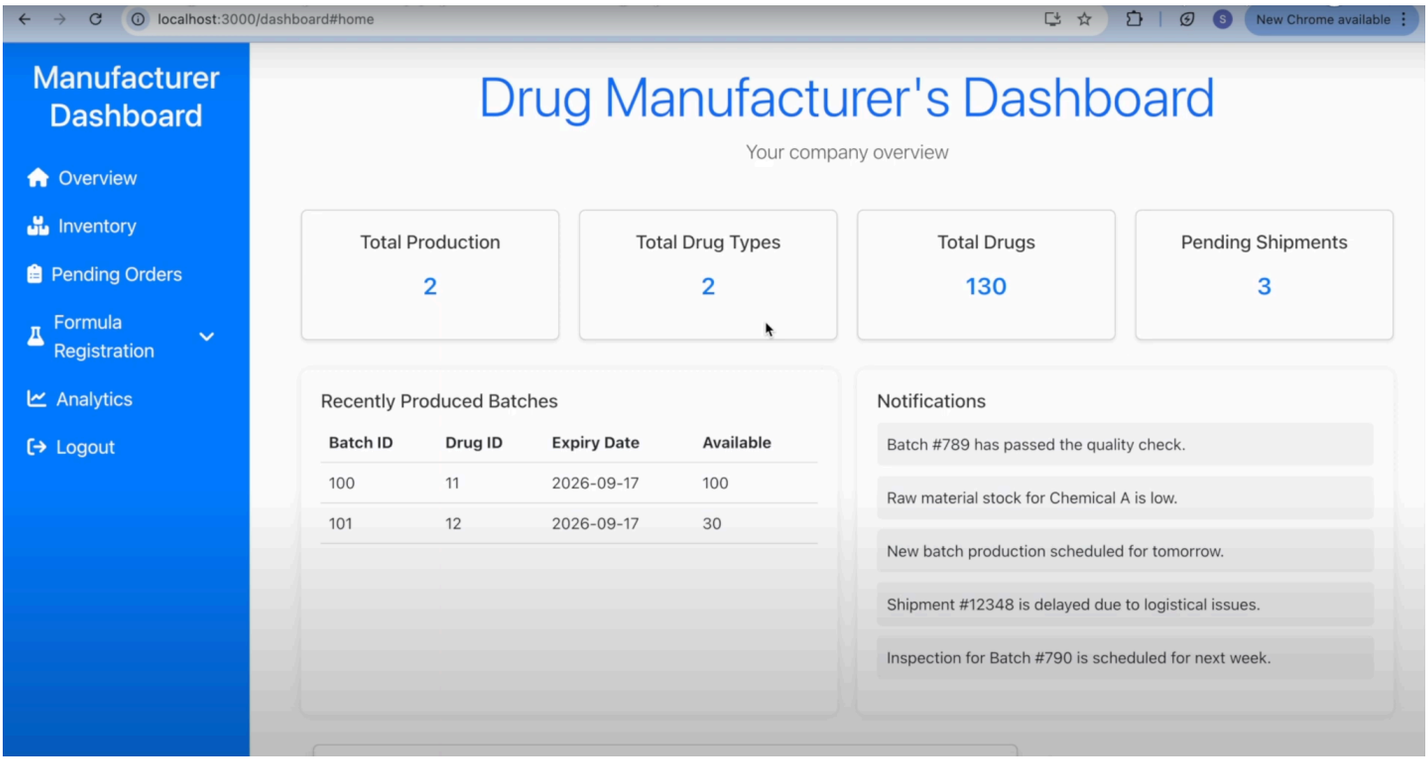
The generated datasets were used to:

- Evaluate Kafka performance under various configurations (e.g., different batch sizes, partition counts, replication factors).
- Measure **throughput, latency, jitter and kafka request time** across different workloads.
- Stress tests the microservices architecture, particularly under heavy traffic and simulated faults.

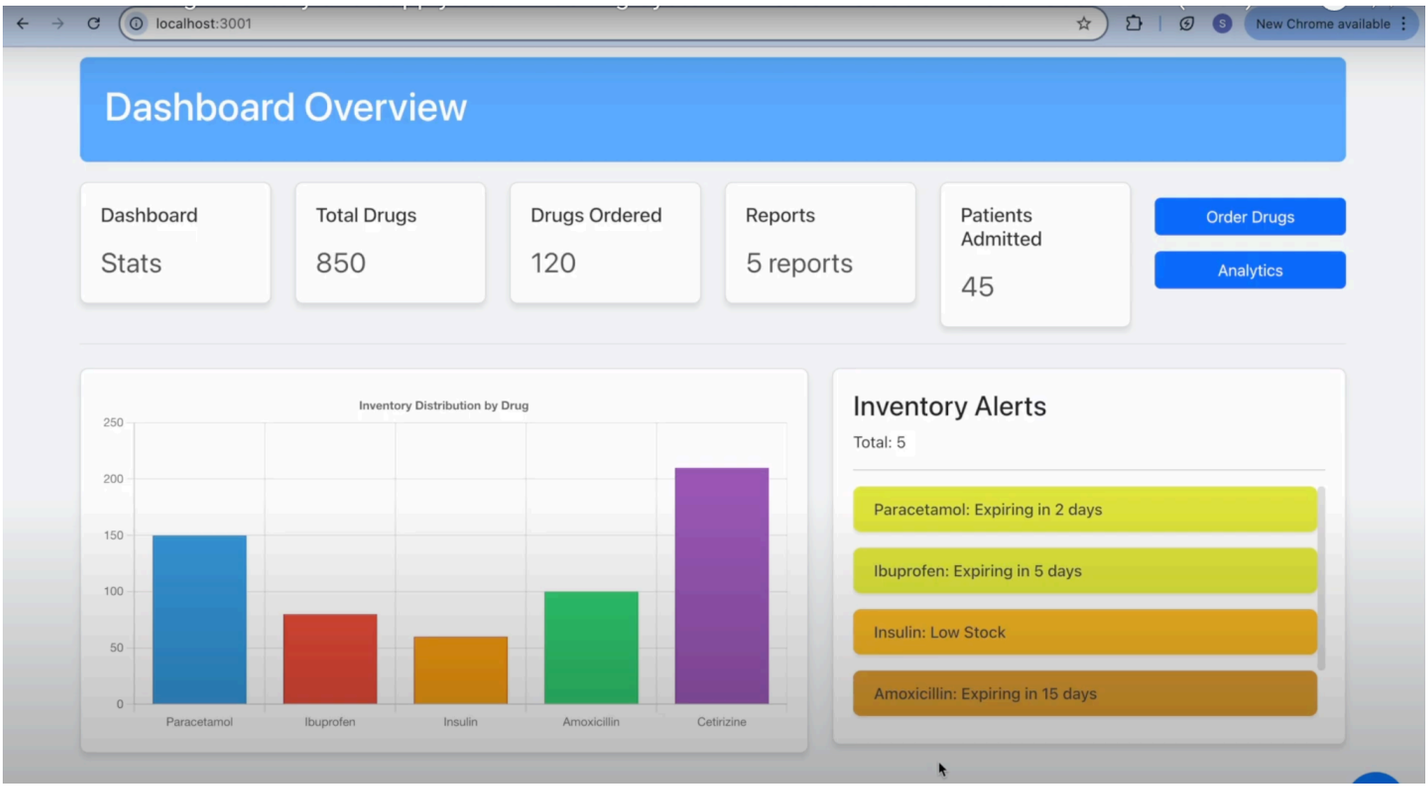
Chapter 6: Proposed Results and Discussions

6.1 Screenshots of User Interface (UI) for the respective module

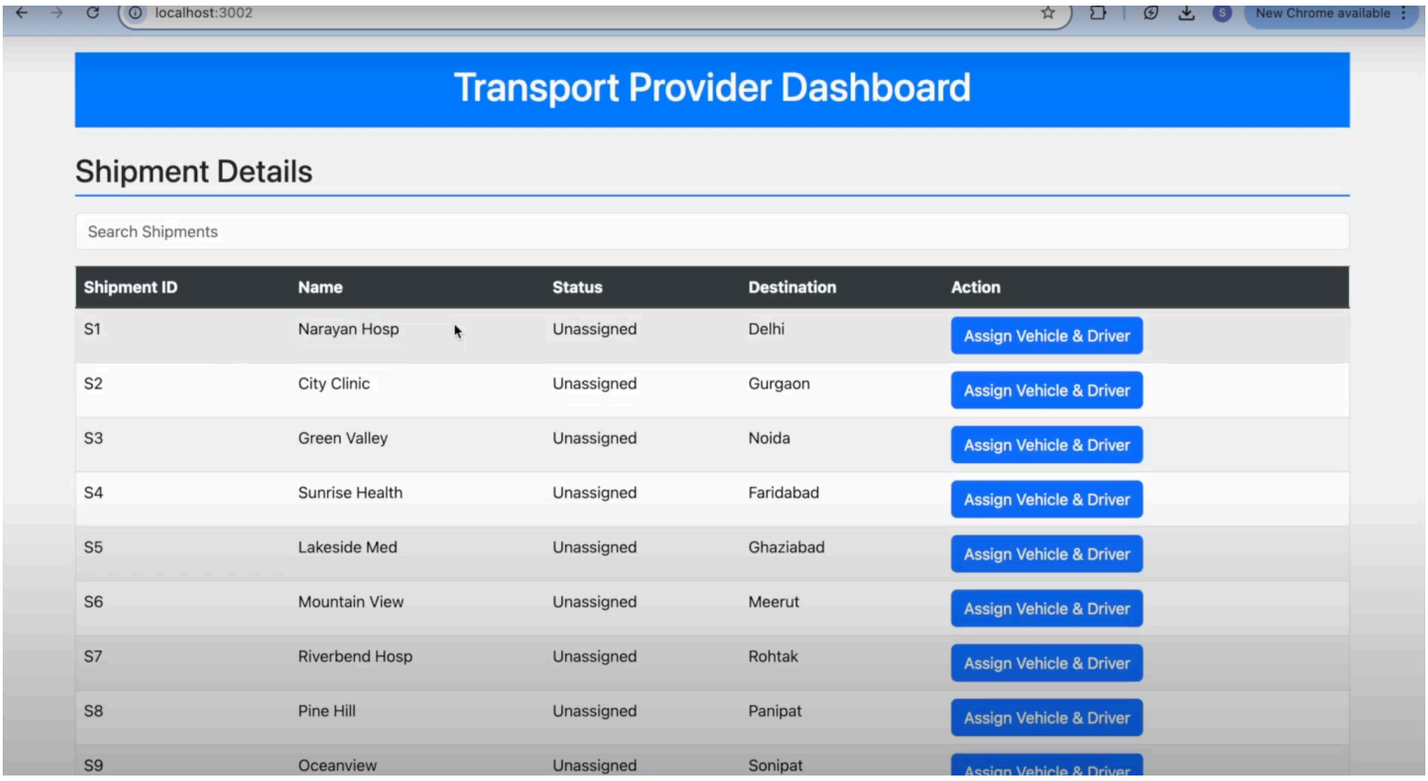
Manufacturer's Dashboard:



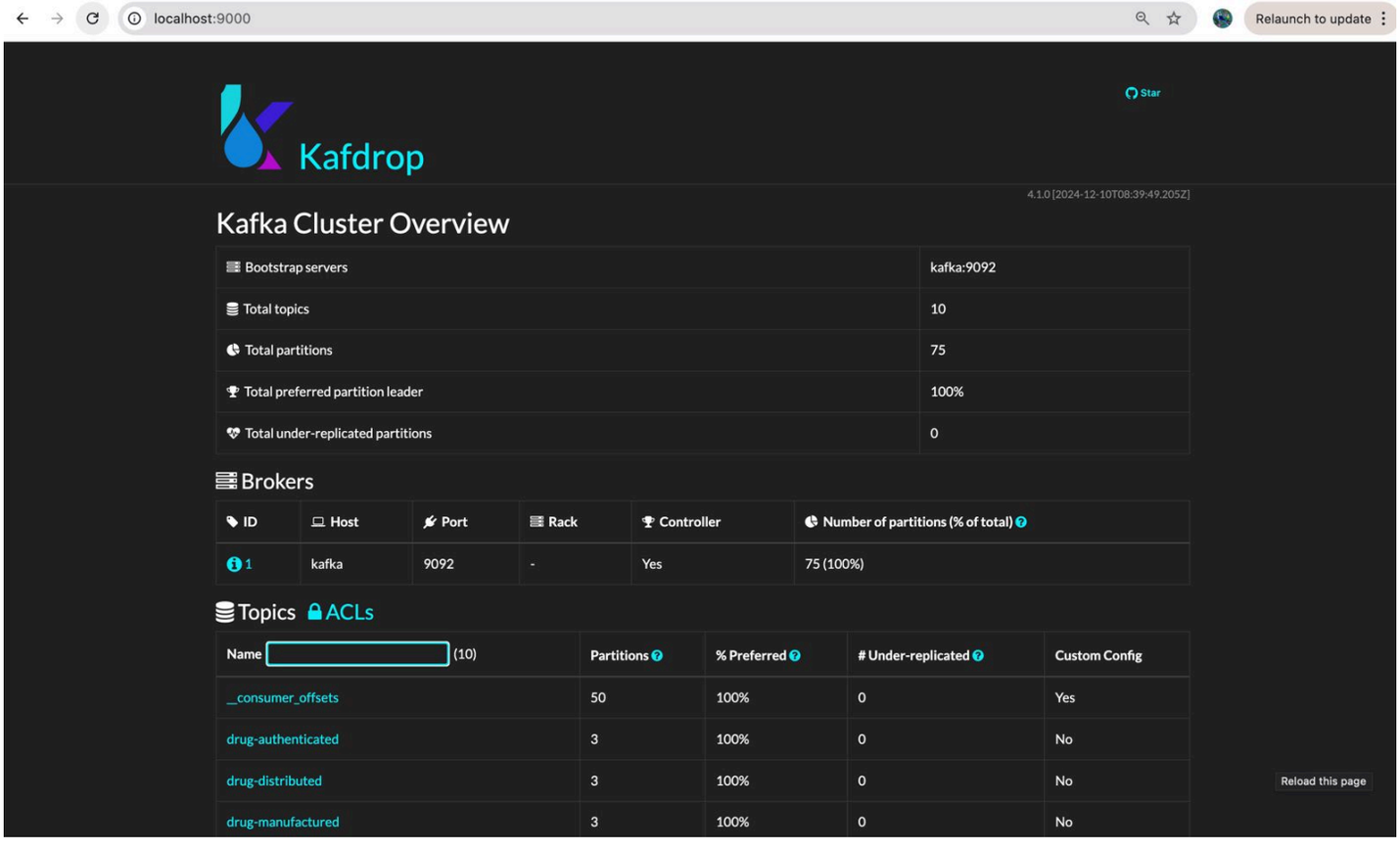
Hospital's Dashboard:



Transport Provider's Dashboard:



Kafdrop: Web UI for Kafka Topic Monitoring



6.2 Kafka Parameters

The Kafka cluster's performance was evaluated based on four key parameters, each playing a crucial role in determining the system's efficiency, scalability, and fault tolerance:

1. **Brokers**
   Kafka brokers are core components responsible for storing and managing message data, and for facilitating communication between producers and consumers. Deploying multiple brokers enhances fault tolerance, prevents performance bottlenecks, and ensures high availability—features that are essential in large-scale supply chain systems.
2. **Replication Factor**
   This parameter defines the number of copies of each message that are maintained across different brokers. A higher replication factor increases resilience to data loss in the event of broker failures but also adds storage overhead and synchronization complexity. In this study, a replication factor of three was chosen to strike an optimal balance between reliability and resource utilization.
3. **Partitions**
   Kafka topics are divided into partitions to enable parallel processing. Increasing the number of partitions allows multiple consumers to read and process messages simultaneously, thereby improving throughput. However, an excessive number of partitions can introduce synchronization challenges. The experimental setup carefully balances the partition count to maximize performance without incurring overhead.
4. **Partitioning Strategy**
   This determines how messages are distributed across partitions. Two strategies were employed in the experiment:
   - **Round-Robin (RR):** Distributes messages evenly across all partitions. This strategy promotes load balancing but does not preserve message order, which may not be ideal for all use cases.
   - **Key-Based (KB):** Routes messages with the same key (e.g., `order-id`) to the same partition. This ensures message ordering and enhances data locality, making it particularly useful for event-driven workflows and supply chain tracking scenarios.

6.3 Performance Metrics

The performance of the Kafka-based system was evaluated using four key metrics. These metrics were selected to measure Kafka's efficiency, responsiveness, and scalability under varying configurations and workloads:

1. **Average Latency (ms)**
   Latency refers to the time taken for a message to be produced, transmitted, and consumed by Kafka clients. It serves as a critical indicator of system responsiveness. In supply chain applications, low latency is essential for real-time event stream processing such as order placements and inventory updates—enabling timely and informed decision-making. Higher latency may result in delays in data availability, thereby affecting operational efficiency. Monitoring latency helps in tuning Kafka configurations to ensure optimal performance under different load conditions.

2. **Kafka Request Time (ms)**
   Kafka request time captures the end-to-end duration required to complete a Kafka request. This includes network overhead, broker-level processing, and replication delays. It provides insights into the internal efficiency of Kafka's operations, including message routing and acknowledgment handling. Lower request times reflect faster broker responsiveness, while elevated values may point to system bottlenecks. Optimizing this metric is crucial for sustaining consistent performance in distributed environments with dynamic workloads.
3. **Average Jitter (ms)**
   Jitter represents the variability in message delivery time between successive messages. It is typically caused by factors such as network congestion, broker load, or processing delays. In event-driven systems like supply chain management, high jitter can disrupt message sequencing and compromise data consistency. Maintaining low jitter ensures stable and predictable message delivery, which is vital for preserving data integrity. Regular monitoring and tuning can help reduce jitter, especially in high-throughput scenarios.
4. **Throughput (msgs/sec)**
   Throughput denotes the volume of messages processed by Kafka per second (msgs/sec). It reflects the system's capacity to handle high message loads efficiently. A higher throughput indicates better scalability and processing power, which is crucial for supply chain systems that demand continuous data flow even during load spikes. This metric is heavily influenced by factors such as the number of brokers, replication factor, and partition count. Evaluating throughput is key to understanding Kafka's capability to scale with growing data demands.

6.4 Evaluation

To evaluate Kafka's performance under varying conditions, three distinct configurations were designed. These configurations differ in terms of broker count, replication factor, number of partitions, and partitioning strategy. The purpose of selecting these variations is to systematically assess how scaling Kafka's architecture affects message distribution, fault tolerance, and processing efficiency.

Configuration 1 (C1) represents a minimal Kafka setup with a single broker, a replication factor of one, and a single partition. This configuration serves as a baseline for performance evaluation, showcasing Kafka's raw throughput and latency in a simple, non-redundant environment. It allows for the analysis of Kafka's behavior without the overhead of replication or inter-broker coordination.

Configuration 2 (C2) introduces a second broker, increases the replication factor to two, and utilizes three partitions. This setup enables the observation of Kafka's behavior under moderate scaling. The inclusion of replication enhances fault tolerance, while multiple partitions facilitate better message distribution and improved concurrency, allowing for more efficient parallel processing.

Configuration 3 (C3) simulates a larger-scale, production-like environment with three brokers, a replication factor of three, and six partitions. It employs key-based partitioning using the `order-id` as the key, which ensures data locality and maintains message ordering—an essential feature for consistency in supply chain operations. The increased replication factor provides robust fault tolerance by ensuring message availability even during broker failures.

This chapter presents the latency evaluation of the system under varying message loads — low, medium, and high. The three configurations compared throughout are **C1 (default), C2 (intermediate), and C3 (optimized).**

Parameter	Config 1	Config 2	Config 3
Brokers	1	2	3
Replication Factor	1	2	3
Partitions	1	3	6
Partitioning Strategy	Round-Robin (default)	Round-Robin (default)	Key-Based Partitioning
Data Distribution	Spread evenly (single broker)	Spread evenly	Key-localized distribution
Fault Tolerance	None (no redundancy)	Moderate	High (3x replication)
Performance Impact	Low fault tolerance, moderate speed	Faster than single-broker setup	More scalable but slightly higher latency due to replication

Table 6.1: Parameter Configuration Strategy

6.5 Values Obtained

Messages	Average Latency (ms)	Kafka Request Time (ms)	Average Jitter (ms)	Throughput (msgs/sec)
100	21.4	8.6	6	93.5
250	38.7	6.1	4.8	108.7
500	45.3	5.8	4.2	113.2
750	51.2	6.7	4.9	145.3
1000	57.1	7.4	5.1	180.4
1500	64.8	8.2	6.3	182.5
2500	72.3	9.5	7.1	206.6
3000	75.6	10.1	7.8	210.2
5000	84.2	11.4	8.3	212.7
7500	92.8	12.7	9.1	213.9
10000	101.5	14.5	10.6	214.3
12500	109.2	15.8	11.9	215.7
15000	115.7	17.3	13.2	215.2
17500	122.6	19.1	14.7	214.5
20000	130.8	20.8	15.9	213.1
22500	140.2	22.4	17.5	210.8
25000	150.5	24.3	18.7	208.3
30000	165.4	27.6	20.1	205.9
35000	180.9	30.2	22.5	200.7
40000	190.2	34.1	25.3	195.4
45000	205.7	37.5	28.1	180.2
50000	218.9	40.9	30.4	165.8

Table 6.2: Values Obtained from Configuration 1

Messages	Average Latency (ms)	Kafka Request Time (ms)	Average Jitter (ms)	Throughput (msgs/sec)
100	15.8	5.2	4.1	125.7
250	28.4	4.5	3.8	140.3
500	34.7	4	3.2	162.8
750	39.9	4.3	3.5	185.4
1000	42.1	4.7	3.6	235.1
1500	48.9	5.5	4.1	247.3
2500	56.2	6.1	4.9	280.6
3000	59.4	6.8	5.3	295.4
5000	67.1	7.9	6.1	312.9
7500	75.6	9.4	7.3	317.4
10000	82.3	10.8	8.2	320.1
12500	89.4	12.1	9.5	325.7
15000	95.1	13.7	10.9	324.9
17500	102.8	15.4	12.4	322.7
20000	110.3	16.8	13.6	320.4
22500	118.6	18.2	14.9	318.1
25000	127.5	19.6	15.7	316.4
30000	139.3	22.1	18.4	312.8
35000	152.6	24.3	20.9	308.7
40000	160.9	25.8	22.7	298.1
45000	174.3	28.9	25.4	282.6
50000	185.7	31.4	28.6	265.2

Table 6.3: Values Obtained from Configuration 2

Messages	Average Latency (ms)	Kafka Request Time (ms)	Average Jitter (ms)	Throughput (msgs/sec)
100	13.2	4.1	3.5	140.8
250	24.8	3.7	3.2	158.3
500	30.5	3.4	2.8	182.6
750	34.2	3.6	3	210.4
1000	36.9	4	3.1	260.4
1500	43.7	4.7	3.8	275.8
2500	50.1	5.4	4.3	310.9
3000	54.3	6.1	4.7	328.5
5000	62.8	7.2	5.5	345.1
7500	70.2	8.8	6.7	355.7
10000	78.5	10.1	7.5	360.8
12500	85.7	11.4	8.9	366.4
15000	93.6	12.9	10.2	365.1

17500	101.4	14.6	11.7	363.2
20000	109.8	16.3	13.1	360.9
22500	118.1	18	14.6	358.5
25000	127.4	19.3	15.4	355.2
30000	139.9	21.7	18.1	350.6
35000	152.5	24.2	20.6	345.3
40000	165.8	27.1	23.4	340.1
45000	178.3	29.5	25.8	325.4
50000	192.7	32.4	28.5	310.2

Table 6.4: Values Obtained from Configuration 3

6.6 Graphical Representation of Performance Metrics

6.1 a. Latency Under Low Load

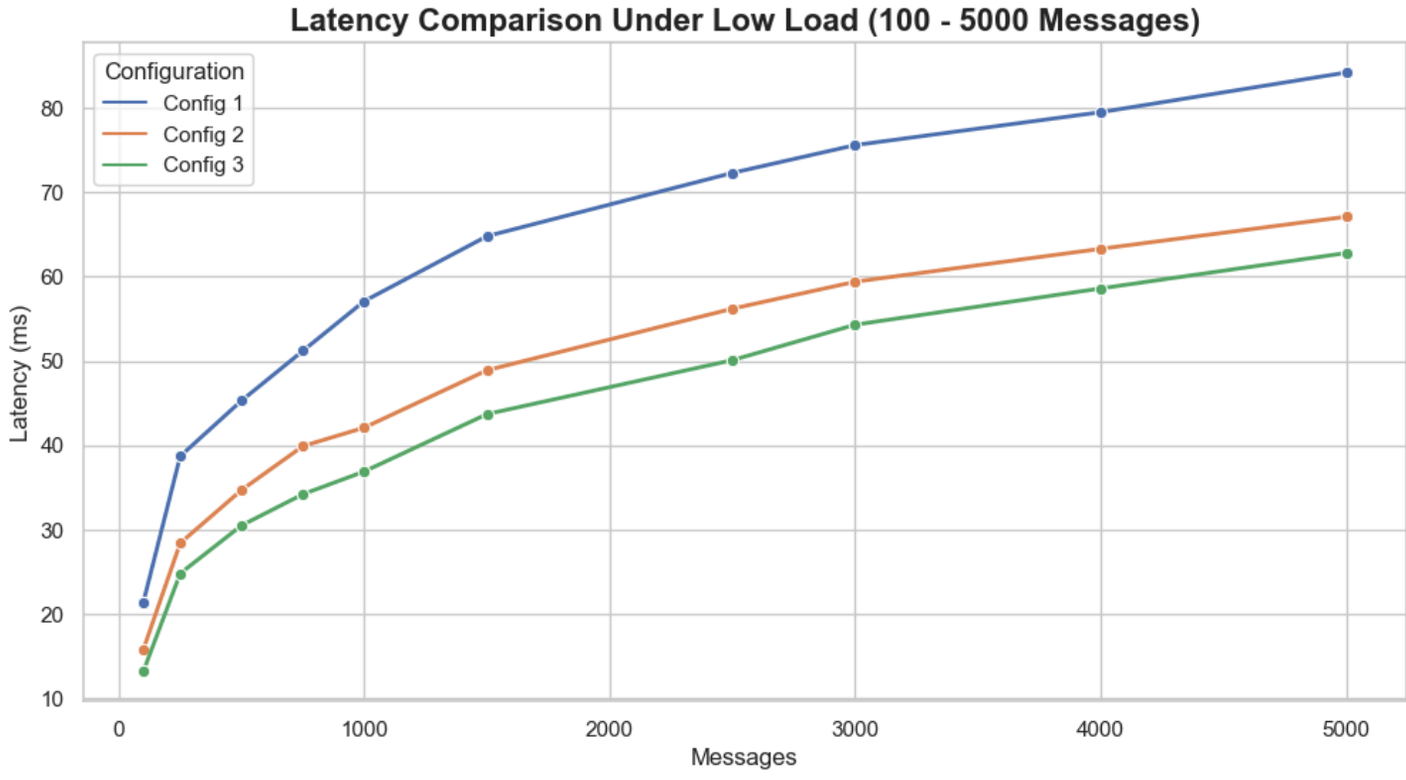


Figure 6.1 a.: Latency Under Low Load

Under low message volumes, all configurations exhibit efficient performance. However, Configuration C3 consistently delivers the lowest latency across the board.

Key Observations:

- At **100 messages**:
  - **C3**: 13.2 ms
  - **C2**: 15.8 ms (**16.5% higher** than C3)
  - **C1**: 21.4 ms (**38.3% higher** than C3)
- At **250 messages**:
  - **C3**: 24.8 ms

- **C2**: 28.4 ms
- **C1**: 38.7 ms (**C3 is 35.9% better than C1**)

Reasons for C3's superior low-load performance:

- Higher broker count and partitioning increases parallelism.
- Key-based partitioning improves data locality.
- Reduced linger time and smaller batch sizes minimize broker synchronization delays.
- Balanced load reduces disk I/O and context switching overhead.
- Optimized network paths and fewer GC pauses enhance responsiveness.

Conclusion: C3 is highly efficient for low-load, real-time use cases such as lightweight supply chain event tracking.

6.1 b. Latency Under Medium Load

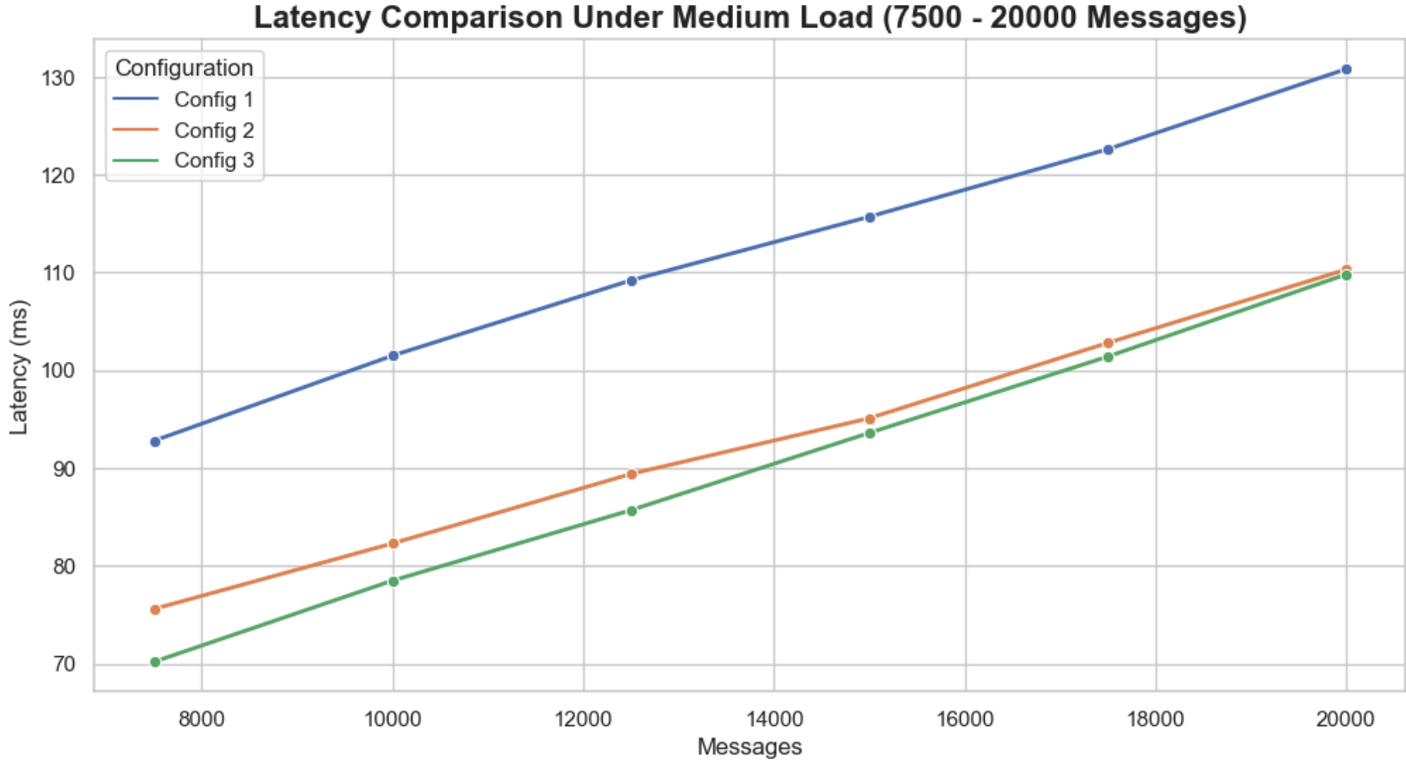


Figure 6.1 b. : Latency Under Medium Load

As the message volume increases to a medium load, overall latency rises for all configurations due to higher processing demands. Despite this, C3 maintains the lowest latency for most of the medium-load range.

However, near **20,000 messages, C2 starts matching C3's performance.** This is due to replication **overhead in C3**, which becomes more prominent at scale.

Comparative Notes:

- C3 continues to benefit from parallelism.
- C2 avoids replication-induced delays, making it more stable under mid-level workloads.

Conclusion: C3 remains effective, but **C2 is more consistent** under medium loads and may be more efficient when fault tolerance is less critical.

6.1 c. Latency Under High Load

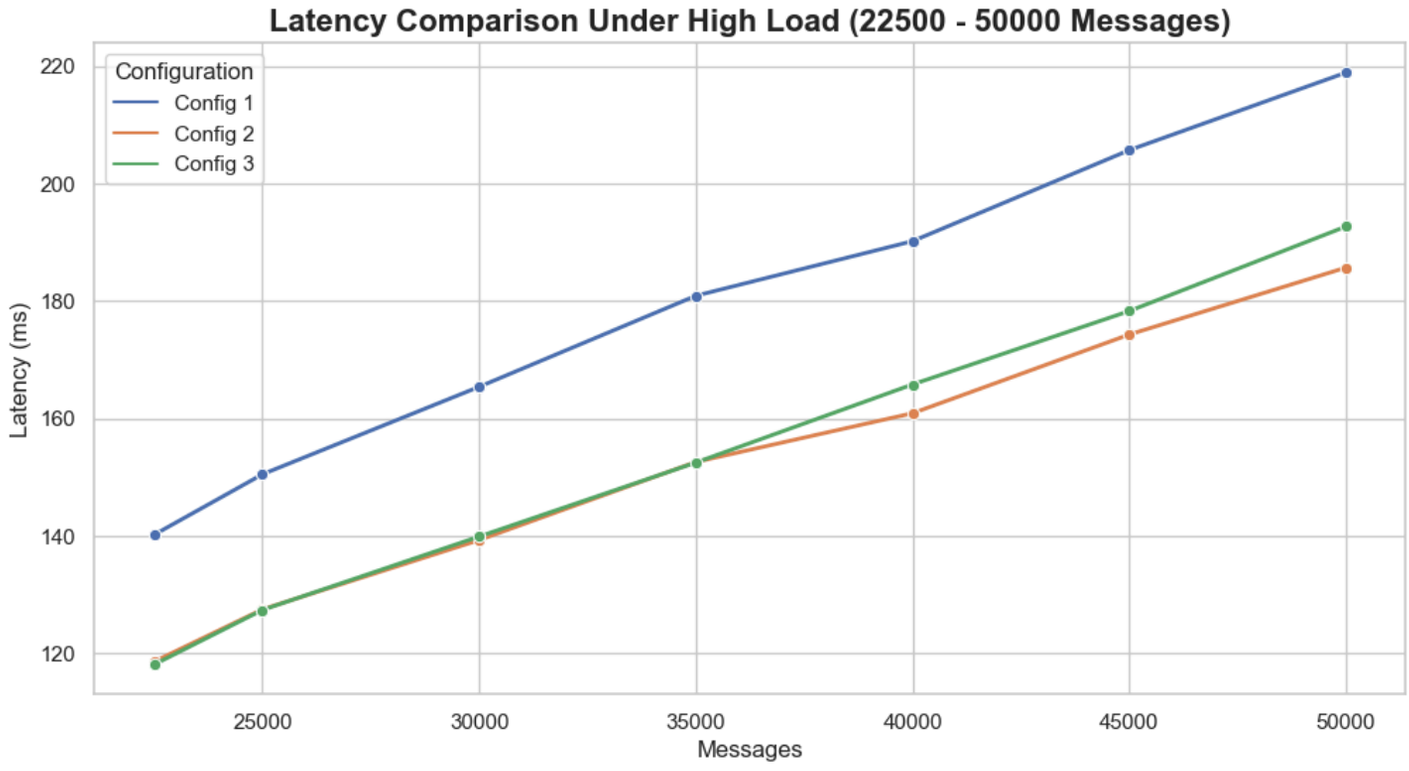


Figure 6.1 c.: Latency Under High Load

In the **high-load scenario (≥ 35,000 messages)**, a shift is observed. **C3 initially performs better**, but as the load grows, its **latency increases** due to the cost of replication operations.

Meanwhile, **C2 achieves better latency** at peak traffic because it carries less replication overhead.

Key Takeaways:

- **C3**: Optimized for fault tolerance but suffers under very high throughput.
- **C2**: Offers **better latency performance at scale** with a trade-off in reliability.

Conclusion: There is a **clear trade-off**:

- Use **C3** when fault tolerance and data locality are critical.
- Use **C2** when minimizing latency at high volumes is the primary goal.

6.2 Kafka Request Time

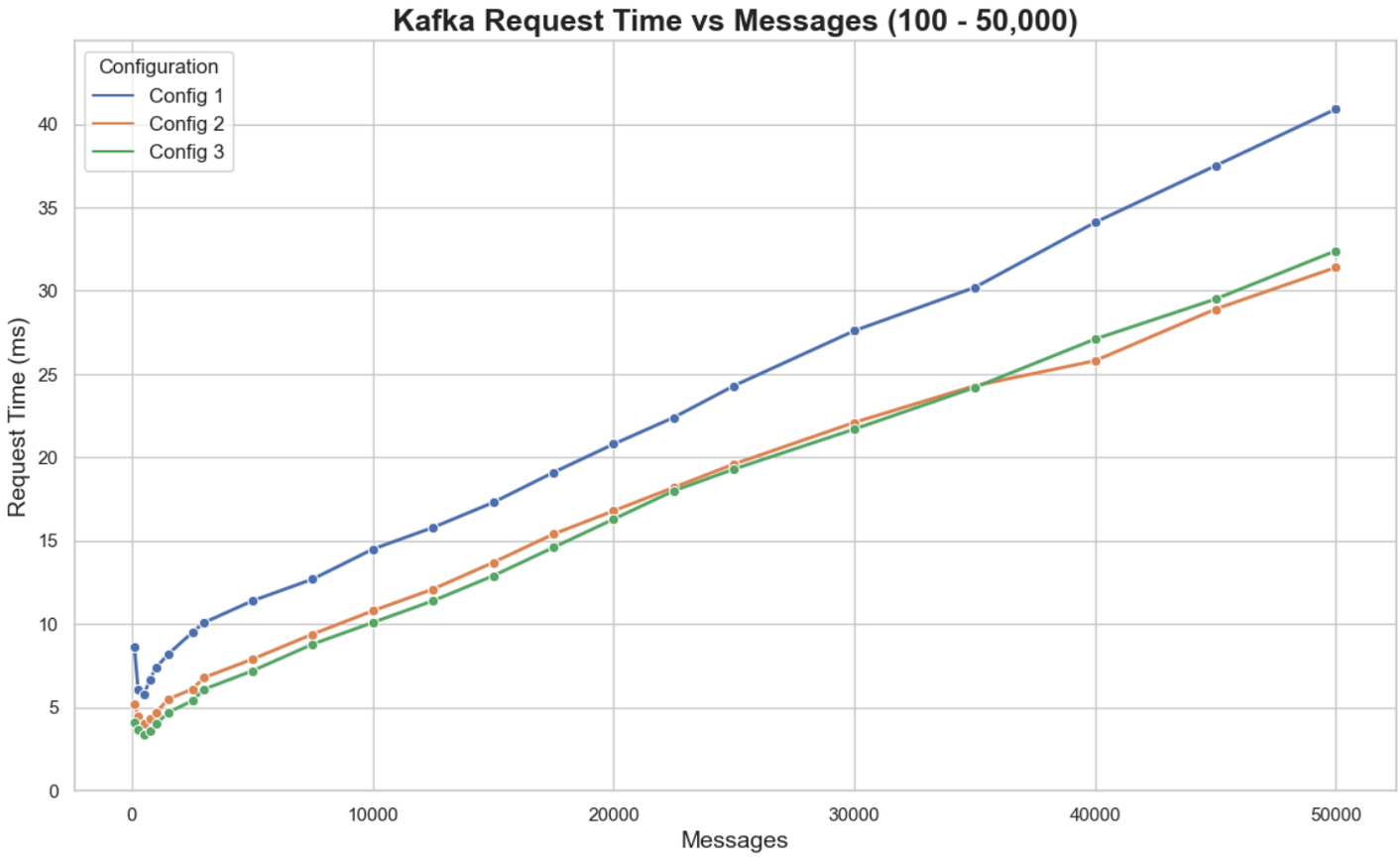


Figure 6.2: Kafka Request Time vs Messages

The analysis of Kafka request time highlights how different configurations perform under increasing message loads. The trend, as depicted in the figure above, shows a steady rise in request time across all configurations — **C1 (default)**, **C2 (intermediate)**, and **C3 (optimized)** — as the number of messages increases. However, the **rate of increase and performance efficiency** varies notably.

Low to Moderate Load Observations:

At lower message volumes, **C3 demonstrates the most efficient request times**, owing to its higher broker count, increased partitioning, and key-based partitioning strategy. These factors collectively enhance load distribution and reduce processing overhead.

- At **100 messages**:
  - **C3**: 4.1 ms
  - **C2**: 5.2 ms (C3 is **21.2% faster**)
  - **C1**: 8.6 ms (C3 is **52.3% faster**)
- At **250 messages**:
  - **C3**: 3.7 ms
  - **C2**: 4.5 ms
  - **C1**: 6.1 ms (**C3 improves by 39.3% over C1**)

High Load Observations

As message volume increases further, **C3 continues to outperform**, especially under moderate to high loads:

- At **10,000 messages**:
  - **C3**: 10.1 ms
  - **C2**: 10.8 ms
  - **C1**: 14.5 ms (C3 shows a **30.3% improvement** over C1)

However, at **extremely high message volumes** (≥ 40,000 messages), **C3's performance slightly degrades** compared to C2:

- At **40,000 messages**:
  - **C3**: 27.1 ms
  - **C2**: 25.8 ms

This slight decline is due to the **routing complexity introduced by key-based partitioning** in C3, which ensures message locality but at the cost of increased processing overhead. In contrast, **C2's round-robin strategy**, although less fault-tolerant, handles large-scale message distribution with minimal routing overhead.

Conclusion:

- **C3** excels in low to moderate loads, offering the **lowest request times** for real-time, low-latency applications.
- **C2 performs slightly better at very high volumes**, making it more suited for **massive-scale batch processing** where routing simplicity and throughput are prioritized.

6.3 Jitter

Figure 6.3: Jitter vs. Messages

The jitter analysis illustrates how message delivery consistency changes with increasing message volume across the three configurations — **C1 (default), C2 (intermediate), and C3 (optimized)**. Jitter, defined as the variation in message delivery time, serves as a critical metric for evaluating real-time system stability. Lower jitter values reflect more predictable and uniform message propagation.

Low to Moderate Load Observations:
At smaller message volumes, **C3 demonstrates superior consistency**, delivering the lowest jitter. This is attributed to its **higher broker count, increased partitions, and key-based routing**, which together reduce queuing delays and improve data locality.

- At **100 messages**:
  - **C3**: 3.5 ms
  - **C2**: 4.1 ms (C3 is **14.6% lower**)
  - **C1**: 6.0 ms (C3 is **41.7% lower**)
- At **250 messages**:
  - **C3**: 3.2 ms
  - **C2**: 3.8 ms
  - **C1**: 4.8 ms (**C3 improves by 33.3% over C1**)

Mid-range Load Observations:
As message volume rises, **C3 maintains more stable jitter** compared to the other configurations.

- At **5,000 messages**:
  - **C3**: 5.5 ms
  - **C2**: 6.1 ms
  - **C1**: 8.3 ms (**C3 outperforms C1 by 33.7%**)

This performance can be credited to **C3's key-based partitioning**, which keeps related messages within the same partition, minimizing rebalancing delays and ensuring consistent delivery.

High Load Observations:

At extremely high volumes (≥ 40,000 messages), **the jitter values for C2 and C3 converge**, reducing the gap between them due to network congestion and processing saturation. These bottlenecks limit the benefits of C3's routing advantages.

- At **50,000 messages**:
  - **C3**: 28.5 ms
  - **C2**: 28.6 ms
  - Difference: **Just 0.1 ms**

Here, **C2's round-robin distribution strategy** becomes equally effective, as it handles bulk traffic with less computational overhead.

Conclusion:
- **C3 is optimal** for environments requiring **low-latency and consistent delivery**, making it suitable for **real-time streaming and monitoring** applications.
- For **very high throughput workloads**, **C2 becomes equally competitive**, offering **similar jitter** performance with a **simpler load-balancing mechanism**, which is ideal for **batch processing or high-volume distributed systems**.

6.4 Throughput

Figures illustrate the throughput performance across the three Kafka configurations — **C1 (baseline)**, **C2 (intermediate)**, and **C3 (optimized)** — under **Low**, **Medium**, and **High** load conditions, respectively. Throughput, measured in messages per second (msgs/sec), reflects the system's capacity to handle message volume efficiently.

a. Low Load

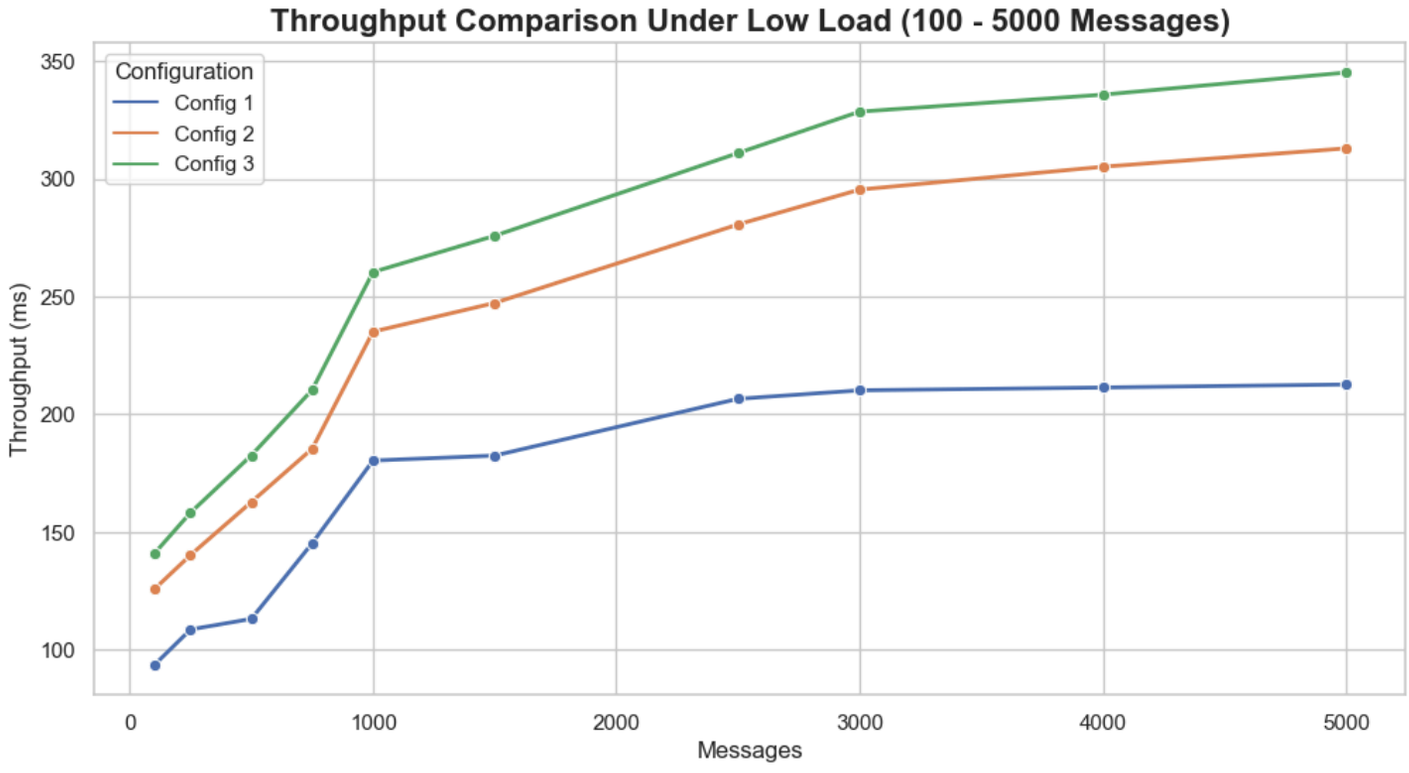


Figure 6.4 a.: Throughput Under Low Load

At lower message volumes, **C3 demonstrates the highest throughput**, followed by C2, while **C1 consistently lags behind**. This behavior showcases C3's architectural advantage: a higher number of brokers, greater partitioning, and efficient key-based message routing. These improvements enable **parallel message consumption**, lowering contention and boosting throughput.

- At **100 messages**:
  - **C3**: 140.8 msgs/sec
  - **C2**: 125.7 msgs/sec
  - **C1**: 93.5 msgs/sec
- At **5,000 messages**:
  - **C3**: 345.1 msgs/sec
  - **C2**: 312.9 msgs/sec
  - **C1**: 212.7 msgs/sec

This widening gap with load increase highlights how C3's superior parallelism and resource allocation handle larger loads more efficiently. On the other hand, C1's single-broker architecture quickly becomes a bottleneck, showing poor scalability and making it unsuitable even for moderate message traffic.

b. Medium Load

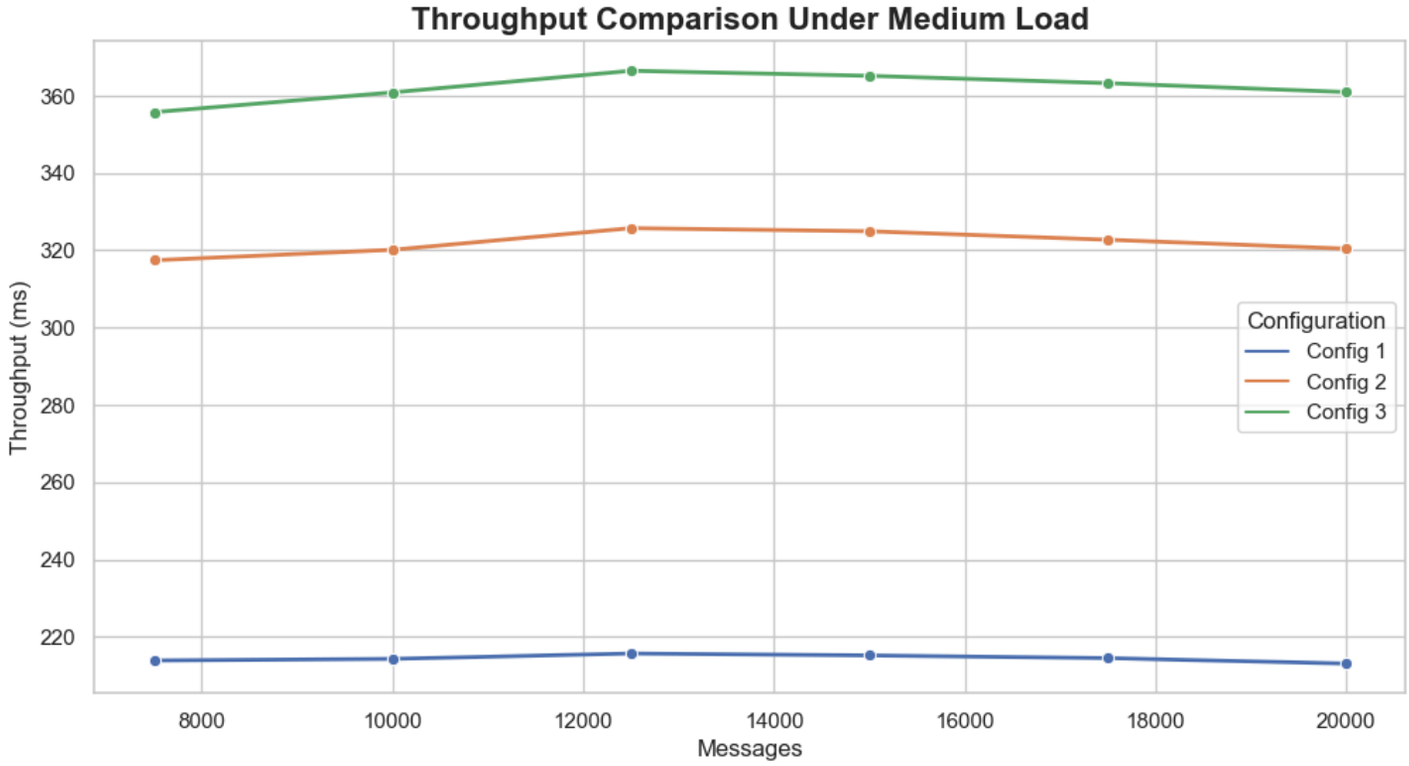


Figure 6.4 b.: Throughput Under Medium Load

As message volume increases to a **moderate load**, all configurations begin to show a **plateauing throughput trend**. Although **C3 continues to outperform C2**, the difference between them narrows. This is due to Kafka's **internal overhead**, such as replication synchronization, disk I/O latency, and inter-broker communication costs, which limit the throughput benefits of scaling.

- Kafka's **resource utilization becomes less efficient** at this stage.
- The performance gains from additional brokers and partitions in **C3 begin to level off**.
- **C1** continues to fall behind, struggling with scalability limitations and increased bottlenecks.

c. High Load

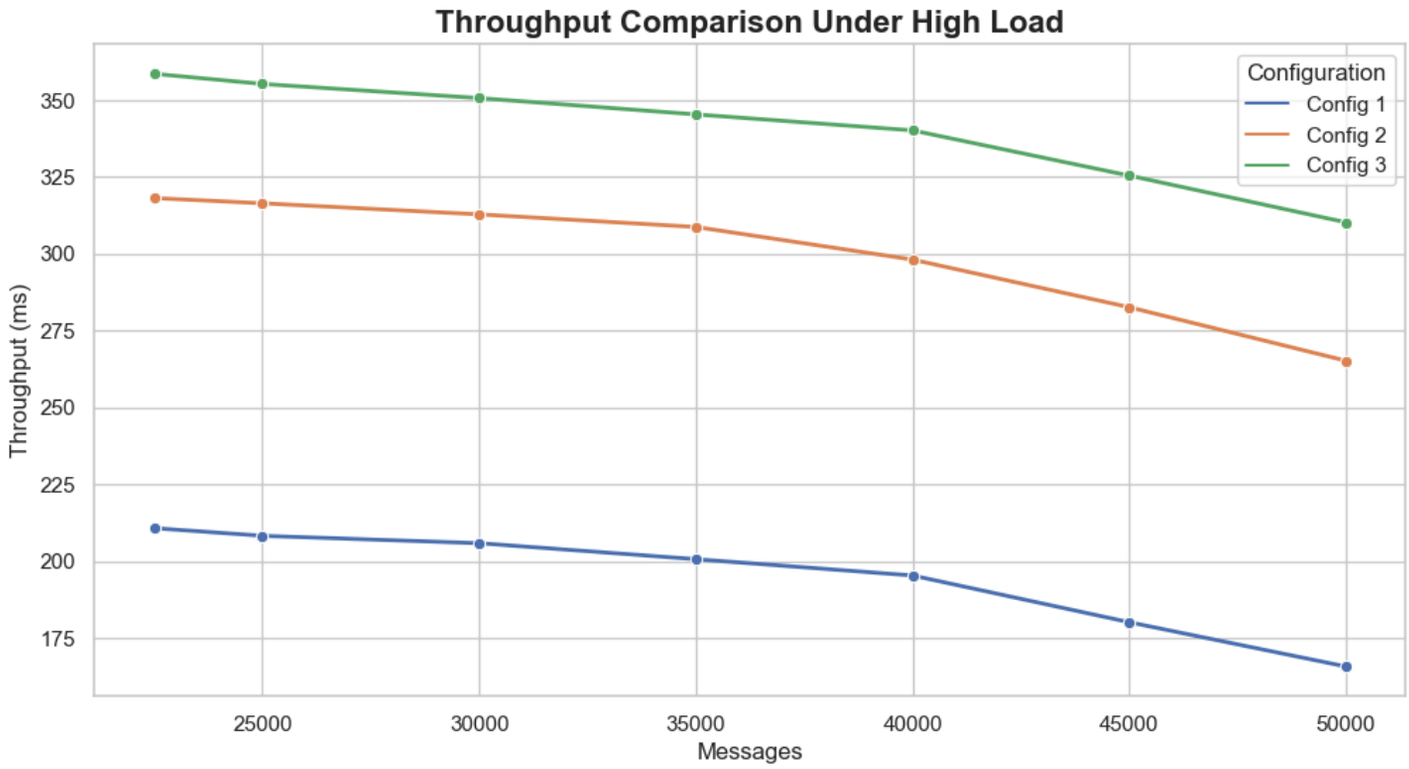


Figure 6.4 c.: Throughput Under High Load

At high message volumes, **throughput begins to decline across all configurations**, revealing the system's saturation limits. While **C3 still leads**, its performance starts to **degrade more rapidly** due to **higher replication and coordination overhead**.

- **C3** begins to show diminishing returns from its optimization strategies.
- **C2**, with a **lighter replication load**, sustains performance more steadily and starts **closing the gap** with C3.
- **C1**, which lacks parallelism and has limited capacity, experiences a **steep decline** in throughput.

This trend highlights the importance of **balancing replication, broker count, and partition strategy**. While **C3 excels under low to moderate load**, **C2 offers a more stable throughput at very high volumes** due to its simpler distribution model and lower coordination cost.

Conclusion:

- **C3** is ideal for high-throughput, real-time workloads, especially under low to medium traffic.
- **C2** becomes increasingly competitive at scale, providing **more consistent performance** under heavy load conditions.
- **C1** is suitable only for basic, low-traffic Kafka deployments due to its limited scalability.

Chapter 7 : Conclusion

7.1 Limitations

Despite the system's improved performance and scalability through Kafka tuning, several limitations exist in the current implementation:

- **Synthetic Workload:** The experimental environment used simulated data instead of real-world supply chain datasets. While this enabled controlled testing, it may not fully capture the complexities of real production systems.
- **Limited Scaling Scope:** Kafka configurations were tested on a single-machine Docker environment. Real-world distributed deployments across multiple physical or cloud nodes might exhibit different performance behavior due to network latency, hardware variations, and orchestration complexity.
- **Zookeeper Bottleneck:** The architecture relies on a single Zookeeper instance, which may act as a single point of failure and limit fault tolerance in larger deployments.
- **Static Partitioning Strategy:** The system uses static partitioning strategies (round-robin and key-based). Dynamic partition rebalancing based on load or topic traffic was not explored.
- **Security Aspects Ignored:** The study did not address security mechanisms such as message encryption, authentication, or authorization, which are crucial in a production-grade supply chain system.

7.2 Conclusion

This study conducted a comprehensive performance evaluation of a Kafka-based microservices architecture for supply chain management under three different configurations—C1, C2, and C3—across varying workload levels (low, medium, and high). The experimental analysis focused on critical performance metrics including **latency, throughput, Kafka request time, and jitter**, revealing insightful trends in system behavior and optimization strategies.

1. Superior Performance of C3 at Low and Medium Loads

- Configuration C3 consistently delivered the lowest latency and jitter under light to moderate workloads, owing to its higher broker count, expanded partitioning, and key-based message distribution.
- At message volumes between 1,000 and 10,000:
    - C3 exhibited **35.3% lower latency** and **41.7% lower jitter** than C1.
    - It achieved **17% higher throughput** compared to C2, making it the most effective for **real-time event streaming**.
- The reduced Kafka request time and consistent message delivery in C3 confirmed its suitability for latency-sensitive applications in supply chain environments.

2. Convergence of C2 and C3 at Higher Loads

- As message volumes increased to the range of **30,000–50,000**, the performance gap between C2 and C3 narrowed.
- **C2 outperformed C3** slightly in terms of latency and jitter under high load conditions due to lower replication overhead and a simpler round-robin distribution strategy.
- This made C2 more efficient for batch processing and high-volume throughput-centric workloads.

3. Kafka Saturation at Extremely High Loads

- Beyond **40,000 messages**, all configurations exhibited increased latency and declining throughput, indicating **Kafka's saturation point** due to factors such as:
    - Disk I/O contention
    - Inter-broker synchronization delays
    - Replication overhead
- These findings highlight the need for further optimization, including dynamic broker scaling, adaptive partitioning, and advanced consumer rebalancing techniques.

4. Real-World Implications

- **C3** is ideal for **low-latency, real-time supply chain event processing**, where responsiveness and ordering consistency are crucial.
- **C2** is better suited for **large-scale, high-volume environments**, offering improved stability with less replication overhead.
- **C1**, with its minimal setup, lacks scalability and fault tolerance, making it suitable only for basic or development-level workloads.

7.3 Future Scope

- **Real-World Deployment & Dataset Integration:** Integrating the architecture with actual pharmaceutical supply chain data and deploying on cloud-based infrastructure (e.g., AWS, Azure) will allow for more comprehensive benchmarking and validation.
- **Security and Compliance:** Future iterations can incorporate secure communication protocols (SSL/TLS), authentication mechanisms (SASL), and compliance with standards like HIPAA or FDA CFR Part 11 for pharmaceutical data.
- **Dynamic Load Balancing & Elastic Scaling:** Implementing smart load balancers and autoscaling Kafka clusters based on message volume and processing lag would improve resilience in fluctuating traffic conditions.
- **Advanced Monitoring and Anomaly Detection:** Integration with observability tools like Prometheus and Grafana, alongside ML-based anomaly detection on Kafka streams, would enhance fault prediction and proactive maintenance.
- **Edge-to-Cloud Integration:** As IoT devices become more common in supply chains, extending Kafka to support edge ingestion and cloud-based analytics could improve real-time decision-making and end-to-end visibility.

Chapter 8: References

[1] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. ‘01-2017.

[2] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybylek. Mono lithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access, 10:1–1, 01 2022.

[3] Chris Richardson. Microservices patterns: with examples in Java. Simon and Schuster, 2018.

[4] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. 2015.

[5] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux J., 2014(239), March 2014.

[6] Jay Kreps, Neha Narkhede, and Jun Rao. Benchmarking apache kafka: 2 million writes per second on three cheap machines, 2016. LinkedIn Engineering Blog.

[7] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.

[8] V Ganesh Tejas and Dr. Hemavathy R. Microservices and its intercommunication using kafka. International Research Journal of Engineering and Technology (IRJET), 07(04), April 2020.

[9] Kiran Peddireddy. Enhancing inventory management systems through kafka data stream integration. International Advanced Research Journal in Science, Engineering and Technology, 8(9), September 2021.

[10] S¨oren Henning and Wilhelm Hasselbring. Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. Journal of Systems and Software, 208:111879, 2024.

[11] Suwarna Shukla and Prabhneet Singh. Revolutionizing supply chain management: Real-time data processing and concurrency management using kafka and akka. International Journal of Innovative Science and Research Technology, 9(5), May 2024.

[12] Chandrakanth Lekkala. Designing high-performance, scalable kafka clusters for real-time data streaming. 01 2021.

[13] J. Rao. How to choose the number of topics/partitions in a kafka cluster?, March 2015. [Posted 12-March-2015, Accessed 06-April- 2022].

[14] Satish Krishnamurthy, Ashvini Byri, Ashish Kumar, Dr. Satendra Pal Singh, Om Goel, and Prof. Dr. Punit Goel. Utilizing kafka and real-time messaging frameworks for high-volume data processing. (IJPREMS), 02(02), February 2022.

[15] Theofanis P. Raptis and Andrea Passarella. On efficiently partitioning a topic in apache kafka. May 2022.

[16] Sameer Shukla. Exploring the power of apache kafka: A comprehensive study of use cases. International Journal of Latest Engineering and Management Research (IJLEMR), 08(03), March 2023.

[17] Tejas V and Dr V. Development of kafka messaging system and its performance test framework using prometheus. International Journal of Recent Technology and Engineering (IJRTE), 9:1622–1626, 05 2020.

[18] Guenter Hesse, Christoph Matthies, and Matthias Uflacker. How fast can we insert? an empirical performance evaluation of apache kafka. In 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), page 641–648. IEEE, December 2020.

[19] Roman Wiatr, Renata Słota, and Jacek Kitowski. Optimising kafka for stream processing in latency sensitive systems. In the 7th International Young Scientist Conference on Computational Science, 2018.

[20] Cloudera Inc. Tuning Apache Kafka Performance, 2024. Accessed: 2025-02-27.

Optimizing Kafka-Based Supply Chain Architectures: A Comprehensive Performance Analysis

Nupur Giri
HOD, Computer Engineering
V.E.S.I.T.
nupur.giri@ves.ac.in

Sejal Datir
Computer Engineering
V.E.S.I.T.
2021.sejal.datir@ves.ac.in

Simran Ahuja
Computer Engineering
V.E.S.I.T.
2021.simran.ahuja@ves.ac.in

Sania Khan
Computer Engineering
V.E.S.I.T.
2021.sania.khan@ves.ac.in

Jesica Bijju
Computer Engineering
V.E.S.I.T.
2021.jesica.bijju@ves.ac.in

***Abstract*— This paper examines the impact of Apache Kafka parameter tuning on the performance and scalability of a microservices-based supply chain management system. The study evaluates the influence of key parameters including batch size, compression type, partition count, and replication factor—on critical performance metrics such as throughput, latency, and fault tolerance. Experimental results demonstrate that fine-tuning Kafka configurations significantly reduces processing delays, improves throughput, and enhances system resilience under varying workloads. Notably, the trade-off between replication overhead and performance efficiency becomes evident at high message volumes, where lower replication factors yield better processing efficiency. The findings provide actionable insights and practical guidelines for optimizing Kafka performance in large-scale, event-driven supply chain systems.**

I. Introduction

Supply Chain Management (SCM) is a critical component of modern business operations, ensuring the seamless flow of goods from manufacturers to consumers. Traditionally, SCM systems relied on monolithic architectures, where all core functionalities such as order processing, inventory management, logistics, and tracking were tightly coupled into a single application. While effective in earlier systems, this architecture is increasingly inefficient in handling the complexities and scale of modern supply chains. Monolithic SCM systems suffer from performance bottlenecks when processing large transaction volumes, often resulting in degraded efficiency [1]. They are also prone to system-wide failures, where a single fault can disrupt the entire supply chain workflow [2]. Furthermore, they lack scalability, as the entire application must be scaled even if only one module experiences high demand, leading to inefficient resource utilization [3]. Additionally, they are inflexible to business changes, making it difficult to introduce new features or integrate third-party services without significant rework and redeployment.

To overcome these limitations, modern SCM systems are shifting towards microservices architecture, deployed using Docker containers. The approach presented in this paper consists of five independent, loosely coupled services - Manufacturer, Hospital, Transport, Monitor, and Notification. These services operate independently but communicate through Apache Kafka, a real-time distributed event streaming platform that facilitates fast, reliable, and scalable message exchange between microservices [4]. Docker containerization ensures consistent deployment environments across different infrastructure setups, simplifies service orchestration, and enhances portability. By running each service in an isolated container, the architecture achieves improved fault isolation, independent scaling, and simplified maintenance[5]. While Kafka significantly enhances microservices efficiency, its performance heavily depends on parameter optimization. Improper configurations can lead to processing delays, inefficient resource usage, and lower fault tolerance. This paper investigates the impact of tuning key Kafka parameters, including partition count, batch size, and replication factor, on throughput, latency, and overall system performance. Partitioning enhances scalability by enabling parallel processing across multiple brokers. Replication improves fault tolerance by maintaining redundant copies of messages across brokers, ensuring data durability in case of node failures. Batch size optimizations improve throughput by reducing the overhead of individual message transmission [6].

By fine-tuning these Kafka parameters and deploying services using Docker, the proposed system achieves faster event processing, improved reliability, and better scalability. The results demonstrate how parameter optimization and containerized deployment significantly enhance the resilience and performance of microservices-based supply chain systems [7].

II. Literature Survey

Several studies have explored Apache Kafka's role in real-time messaging, data processing, and system optimization. In [8], Kafka's application in microservices architecture is examined, highlighting its asynchronous messaging capabilities and advantages over traditional RESTful APIs in terms of scalability and fault tolerance. Similarly, [9] investigates

Kafka's integration into inventory management systems, demonstrating how real-time data streaming enhances inventory tracking, demand forecasting, and overall supply chain efficiency. Expanding on Kafka's role in microservices, [10] evaluates the scalability of five modern stream processing frameworks on Kubernetes clusters. The benchmarking reveals that all frameworks exhibit near-linear scalability with sufficient resources but differ in efficiency, concluding that no single framework is universally superior, as performance varies by use case.

Focusing on Kafka in supply chains, [11] proposes an integrated Kafka-Akka architecture for concurrent and real-time processing in supply chain management. The study demonstrates how this architecture enhances inventory optimization, order fulfillment, and operational agility. Meanwhile, [12] covers best practices for building and optimizing scalable Kafka clusters, including architecture design, data partitioning, replication strategies, and configuration tuning. The authors emphasize that efficient cluster architecture and fault tolerance mechanisms are essential for achieving high availability, scalability, and low-latency data streaming at large scales.

Partitioning strategies for performance optimization are analyzed in [13]. The study explains how Kafka partitions enable parallelism and higher throughput but introduces trade-offs. It provides guidelines for determining the optimal number of partitions based on throughput requirements, considering factors such as producer and consumer performance, file handle limits, and memory usage. In a comparative study, [14] evaluates Kafka against other messaging frameworks, such as RabbitMQ and ActiveMQ, concluding that Kafka offers superior performance for high-throughput, low-latency applications. Further refining Kafka's scalability, [15] explores heuristic algorithms for optimizing topic partitioning, improving load balancing, fault tolerance, and replication efficiency.

Kafka's real-world applications across industries are presented in [16], where the authors highlight its adaptability in finance, healthcare, IoT, and fraud detection. The study showcases Kafka's role in predictive analytics and event-driven architectures. Similarly, [17] evaluates Kafka as a distributed messaging system based on the publish-subscribe model, emphasizing its high throughput, fault tolerance, and widespread adoption by Fortune 500 companies. Through multiple test scenarios, the system's efficiency is assessed based on QoS parameters.

Performance benchmarking studies further emphasize the importance of configuration tuning. [18] presents a performance analysis of message brokers, including Kafka, under various data ingestion scenarios. The benchmarking demonstrates Kafka's peak ingestion rate of approximately 420,000 messages per second, reinforcing the need for thorough testing and tuning of Kafka configurations for optimal performance. Similarly, [19] addresses Kafka's latency-sensitive processing, identifying high CPU consumption in KafkaProducer as a bottleneck. The study proposes buffering techniques and API modifications, reducing KafkaProducer's CPU load by 75% while maintaining low latency. Additionally, [20] offers a comprehensive performance tuning guide, covering partitioning strategies, broker configurations, garbage collection, and ZooKeeper optimization, ensuring high-throughput and low-latency Kafka deployments.

III. System Performance and Design Considerations

Optimizing Kafka for a microservices-based supply chain management system requires careful consideration of scalability, fault tolerance, and latency. While Kafka provides an event-driven architecture, its default configurations may not always be optimal for large-scale, real-time supply chains. Factors such as inefficient partitioning, improper replication strategies, and inadequate broker scaling can lead to bottlenecks that impact message throughput and system reliability. To address these challenges, this study explores different Kafka configurations to analyze their effects on system performance.

A key design consideration is partitioning strategy, which directly influences message distribution and processing efficiency. Default round-robin partitioning spreads messages evenly but does not account for data locality, potentially increasing retrieval times. In contrast, key-based partitioning groups related events together, improving access efficiency at the cost of potential load imbalance. Similarly, replication enhances fault tolerance by maintaining redundant copies of data, but higher replication factors introduce additional overhead, affecting overall performance. The trade-off between these factors necessitates a thorough evaluation to determine the most effective configuration for supply chain applications.

This section establishes the foundation for the methodology by outlining the rationale behind the Kafka parameter optimizations explored in this study. The following section presents an empirical evaluation of these configurations, detailing their impact on system performance through experimental analysis and benchmarking.

IV. Methodology

A. System Architecture and Experimental Setup

The Kafka-based drug supply chain system follows a microservices architecture, as illustrated in Fig. 1. It consists of five loosely coupled services - Hospital, Manufacturer, Transport, Monitor, and Notification - communicating through Kafka topics for real-time event streaming. This architecture enables independent scaling, fault isolation, and efficient parallel processing, making the system highly resilient and adaptable to fluctuating workloads. To evaluate the system's performance, a Dockerized Kafka cluster was deployed as the core messaging backbone. The cluster was configured with varying broker counts across three configurations to simulate different levels of scalability and fault tolerance. A single Zookeeper instance was used to manage broker metadata, track broker health, and coordinate leader election. All components were containerized using Docker, running

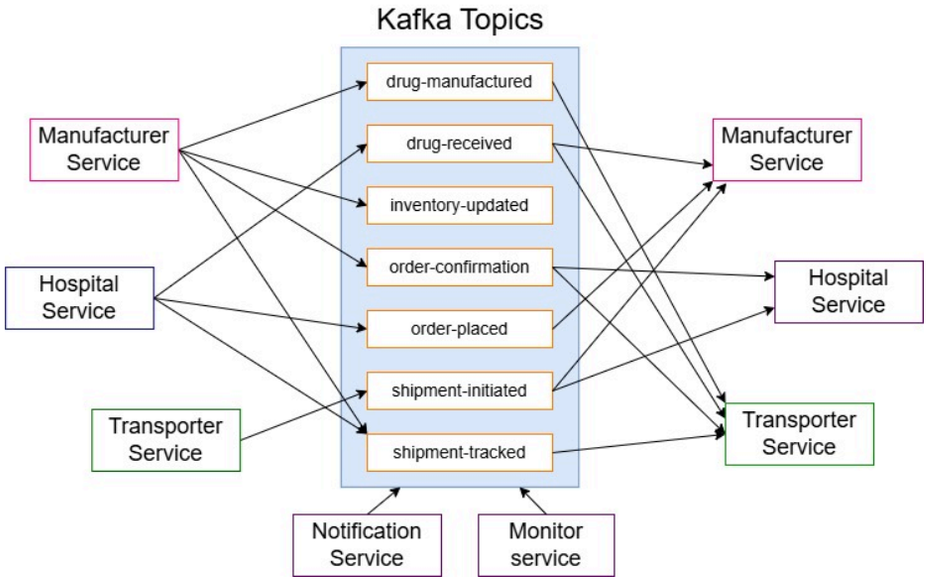


Fig. 1. Kafka architecture diagram

within an isolated bridge network to enable efficient and secure inter-service communication.

The experiments were conducted on a host machine equipped with a MacBook Pro powered by an M1 chip, 16 GB of RAM, and 256 GB of storage. This setup provided sufficient processing power and memory capacity to simulate large-scale workloads without hardware-induced performance bottlenecks.

B. Kafka-Based Microservices Architecture and Communication Flow

The system comprises five microservices, each handling distinct operations within the drug supply chain, as illustrated in Fig. 1. This event-driven architecture enables seamless communication through Kafka topics, ensuring efficient order processing, inventory management, and delivery tracking.

The Hospital Service initiates the workflow by publishing messages to the `order-placed` topic when drug orders are created. It also consumes messages from the `order-confirmation` topic to track the status of placed orders, ensuring hospitals receive real-time updates. The Manufacturer Service consumes messages from the `order-placed` topic to initiate drug production. After verifying inventory availability, it produces updates to the `drug-manufactured`, `inventory-updated`, and `order-confirmation` topics to reflect production progress, ensuring real-time manufacturing and inventory tracking.

The Transport Service handles delivery operations by consuming messages from the `drug-manufactured` topic and producing updates to the `shipment-status` topic, reflecting the shipment's progress. This ensures real-time order tracking and improves transparency. The Monitor Service tracks system performance and Kafka metrics, such as throughput, latency, and broker health. It publishes anomaly alerts to the monitor-alerts topic when irregularities are detected, enhancing observability and fault detection. The Notification Service consumes messages from multiple topics, such as `order-confirmation` and `shipment-status`, broadcasting real-time updates to connected clients to improve supply chain visibility.

Kafka topics serve as communication channels between the microservices, enabling asynchronous, event-driven processing. The `order-placed` topic triggers drug production, while the drug-manufactured topic initiates the delivery workflow. The `shipment-status` topic ensures `real-time` shipment tracking, and the `order-confirmation` topic keeps hospitals informed of order status. The `monitor-alerts` topic enhances system reliability by broadcasting anomalies. This structured topic-based communication ensures reliable event propagation and consistent data flow across the microservices.

C. Test Workload

To simulate real-world usage patterns, the system was subjected to varying message loads, categorized into three distinct levels: light, medium, and heavy. These load levels were designed to assess Kafka's performance under different traffic conditions, evaluating metrics such as throughput, latency, and fault tolerance.

A Python-based simulation script was used to generate the workload, dynamically producing Kafka messages to the `order-placed` topic. Each message represented a drug order, encapsulated in a JSON payload containing details such as drug names, quantities, and timestamps. The timestamps were appended at the time of message production, serving as the basis for calculating end-to-end latency as the message traversed through the microservices. As the system workflow progressed, the subscriber services consumed these messages, triggering subsequent events, including inventory updates, shipment initiation, and notifications.

The workload was classified into the following categories:

- **Light Load**: This level includes message volumes ranging from 100 to 5,000 messages, simulating low-demand scenarios with minimal strain on the system.
- **Medium Load**: This level encompasses message volumes between 7,500 and 20,000 messages, representing stable and consistent order traffic.
- **Heavy Load**: This level simulated peak demand conditions with message volumes ranging from 25,000 to 50,000 messages. It evaluates the system's performance under high-stress scenarios, testing Kafka's capacity to handle large-scale message loads efficiently.

D. Parameter Configuration Strategy

To evaluate Kafka's performance under varying conditions, three distinct configurations were designed, differing in broker count, replication factor, number of partitions, and partitioning strategy. The selection of these configurations aims to capture the impact of scaling Kafka's architecture on message distribution, fault tolerance, and processing efficiency.

- **Configuration 1** (C_1) represents a minimal setup with a single broker, a replication factor of one, and a single partition. This configuration was chosen to establish a performance baseline, reflecting how Kafka behaves in a basic, non-redundant environment. It highlights the raw throughput and latency characteristics without the overhead of replication or multi-broker coordination.
- **Configuration 2** (C_2) introduces an additional broker, increases the replication factor to two, and expands the number of partitions to three. This setup was selected to observe the effect of moderate scaling, specifically how Kafka handles fault tolerance and parallel processing. The replication factor improves reliability, while multiple partitions enable better message distribution and concurrency.
- **Configuration 3** (C_3) employs three brokers, a replication factor of three, and six partitions, using key-based partitioning. This configuration was designed to simulate a larger-scale, production-like environment. The higher replication factor ensures message availability during broker failures, while key-based partitioning—using `order-id` as the key—optimizes data locality and preserves message ordering, which is critical for consistent supply chain operations.

Parameter	C_1	C_2	C_3
Brokers	1	2	3
Replication Factor	1	2	3
Partitions	1	3	6
Partitioning Strategy	Round-Robin	Round-Robin	Key-Based

TABLE I
KAFKA CLUSTER CONFIGURATIONS

Table I summarizes the three Kafka configurations used in the experiment, highlighting the key parameters. By comparing these configurations, the experiment aims to analyze how scaling Kafka's architecture impacts key performance metrics, providing insights into its efficiency under varying workloads.

E. Kafka Parameters

The Kafka cluster's performance was evaluated based on four key parameters, each influencing the system's efficiency, scalability, and fault tolerance.

1) **Brokers:** Kafka brokers are responsible for storing and managing message data, facilitating communication between producers and consumers. Deploying multiple brokers enhances fault tolerance, prevents bottlenecks, and ensures high availability, which is critical in large-scale supply chain systems.
2) **Replication Factor:** This parameter determines how many copies of a message exist across brokers, preventing data loss in case of failures. A higher replication factor improves resilience but increases storage overhead and synchronization complexity. The optimal configuration of three ensures redundancy and minimizes downtime.
3) **Partitions:** Partitions divide Kafka topics into smaller segments, enabling parallelism. More partitions allow multiple consumers to process messages concurrently, enhancing throughput. However, excessive partitions can lead to synchronization overhead. The experiment balances partition count to optimize performance.
4) **Partitioning Strategy:** This defines how messages are distributed across partitions. Two strategies were used:
   - **Round-Robin (RR):** Distributes messages evenly across partitions, optimizing load balancing but sacrificing message order consistency.
   - **Key-Based (KB):** Ensures messages with the same key are routed to the same partition, preserving ordering and improving data locality, which is beneficial for event-driven workflows.

F. Performance Metrics

The system's performance was evaluated using four key metrics, measuring Kafka's efficiency under different configurations and workloads.

1) **Average Latency (ms):** Latency measures the time taken for a message to be produced, transmitted, and consumed by Kafka clients. It reflects the overall responsiveness of the system. In supply chain applications, low latency ensures faster event stream processing (e.g., order placements, inventory updates), enabling real-time decision-making. Higher latency indicates delays in data availability, which can impact operational efficiency. Monitoring latency helps optimize Kafka configurations to maintain efficient message processing under varying loads.
2) **Kafka Request Time (ms):** Kafka request time represents the end-to-end processing time of a Kafka request, including network overhead, broker processing, and replication delays. It offers insights into the efficiency of Kafka's internal operations, such as message routing and acknowledgment. Lower request times

indicate faster broker responsiveness, while higher values reveal potential bottlenecks. Reducing Kafka request time is essential for maintaining consistent performance in distributed systems under fluctuating workloads.

3) **Average Jitter (ms):** Jitter quantifies the variation in message delivery time between subsequent messages, caused by network congestion, broker load, or processing delays. In event-driven systems like supply chains, high jitter can lead to out-of-order message delivery, impacting data consistency. Lower jitter ensures stable and predictable message delivery, which is critical for maintaining data integrity. Monitoring jitter helps fine-tune Kafka configurations, reducing fluctuations, especially under high-load conditions.
4) **Throughput (msgs/sec):** Throughput measures the rate at which Kafka processes messages, represented in messages per second (msgs/sec). Higher throughput indicates better system efficiency and the ability to handle larger message volumes. In supply chain systems, maintaining high throughput ensures continuous event stream processing, even during load spikes. Throughput is directly influenced by broker count, partitioning, and replication factor, making it a vital metric for assessing Kafka's scalability and overall performance.

V. Inferences

A detailed analysis of the experimental results is obtained from evaluating Kafka's performance under varying message loads and architectural configurations and presented in Table II.

The inferences are drawn based on key performance metrics such as jitter, Kafka request time, latency, and throughput. Each metric is analyzed across three distinct Kafka setups to assess their scalability and efficiency. The following subsections provide graphical representations and corresponding observations based on the values shown in Table II, highlighting how Kafka's behavior evolves with increasing message volume. These insights offer a comprehensive understanding of Kafka's performance characteristics and help identify potential bottlenecks under different load conditions.

A. Latency

In this study, we evaluated the average latency under three configurations across three load scenarios: Low, Medium and High Load.

1) **Low Load**

Fig. 2 shows the latency trend under low message volumes reveals that all configurations handle the workload efficiently, but C_3 consistently outperforms the others, exhibiting the lowest latency. This is attributed to its higher broker count, increased partitioning, and key-based partitioning strategy, which collectively enhance parallelism and reduce processing time. At 100 messages, C_3 achieves a latency of 13.2 ms, which is 16.5% lower than C_2 (15.8 ms) and 38.3% lower

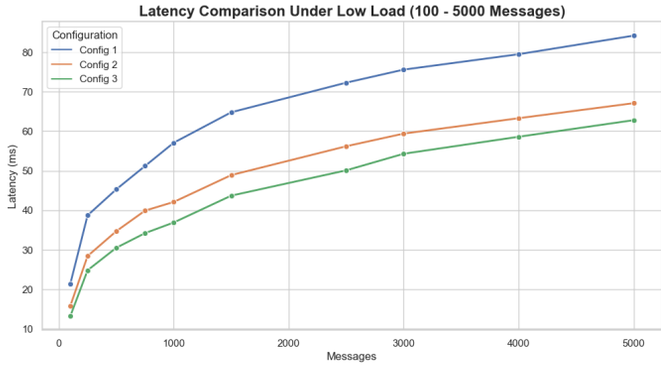


Fig. 2. Latency under Low Load

than C_1 (21.4 ms). This trend persists at 250 messages, where C_3 maintains a latency of 24.8 ms, compared to 28.4 ms for C_2 and 38.7 ms for C_1, highlighting a 35.9% improvement over the default configuration. The superior low-load performance of C_3 stems from its efficient partition parallelism and key-based partitioning, which ensures that related messages are routed to the same partition. This improves data locality and reduces retrieval time. Additionally, lower batch sizes and reduced linger time in C_3 minimize broker-to-broker synchronization delays, enhancing responsiveness.

The improved load distribution across multiple brokers reduces context switching and disk I/O overhead, resulting in faster message retrieval. Furthermore, optimized network round-trip times and reduced Garbage Collection (GC) pauses in C_3 further enhance its efficiency. As a result, C_3 proves highly effective for low-load, real-time streaming scenarios, delivering faster message processing and lower latencies, making it ideal for lightweight supply chain event processing.

2) **Medium Load**

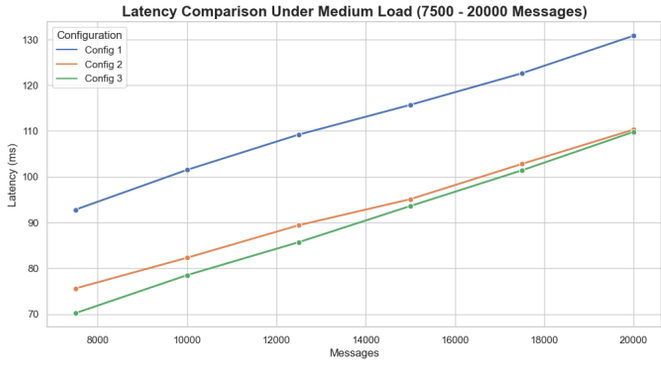


Fig. 3. Latency under Medium Load

Based on Fig. 3, we can derive that as the message volume increases, latency grows for all configurations due to higher processing demands. C_3 maintains the lowest latency across most of the medium load range,

showcasing its superior parallelism and distribution efficiency. However, towards the upper end of the range (near 20,000 messages), C_2 starts to overlap with C_3. The overlapping performance between C_2 and C_3 at higher message counts can be attributed to replication overhead in C_3. While C_3's higher replication factor offers better fault tolerance, it introduces replication-related delays under increased load. Thus, C_2 demonstrates more stable latency performance at medium loads due to fewer replication operations.

3) High Load

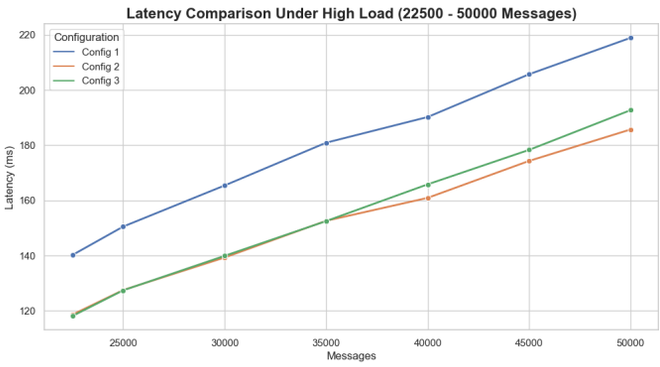


Fig. 4. Latency under High Load

At high message volumes, as depicted in Fig. 4, the latency difference between C_2 and C_3 becomes more noticeable. Although C_3 still offers better latency initially, the performance gap narrows and eventually reverses as the message load increases. C_2 achieves lower latency at the highest message volumes ($\geq$ 35,000 messages), while C_3's performance deteriorates slightly due to the replication overhead. Under high load, the benefits of increased replication in C_3 diminish due to replication-induced latency overhead. The lower replication factor in C_2 makes it more efficient for handling large-scale message loads, resulting in lower latency. This highlights a trade-off between fault tolerance and performance C_3 offers better fault tolerance but slightly higher latency at scale.

B. Kafka Request Time

The Kafka request time trend depicted in Fig. 5 reveals a consistent increase across all configurations as the message volume rises, but the magnitude of increase varies significantly across the three setups.

At low message volumes, C_3 exhibits the lowest request time due to its higher broker count, increased partitioning, and key-based partitioning strategy, which improve load distribution and reduce message processing overhead. For 100 messages, C_3 achieves a request time of 4.1 ms, which is 21.2% lower than C_2 (5.2 ms) and 52.3% lower than C_1 (8.6 ms). This efficiency persists at 250 messages, where C_3 records 3.7 ms, compared to 4.5 ms in C_2 and 6.1 ms in

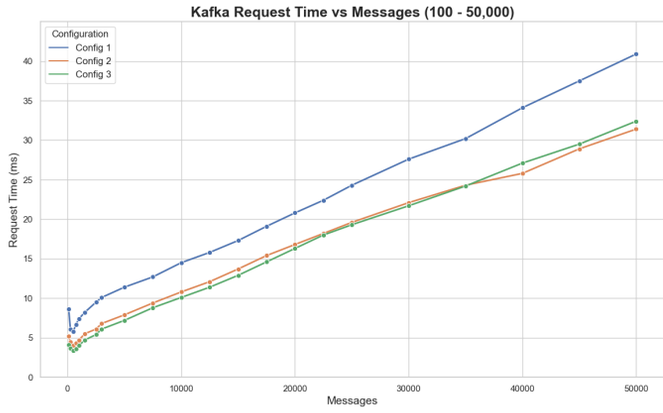


Fig. 5. Kafka Request Time vs Messages

C_1, demonstrating a 39.3% improvement over the baseline configuration.

As the message volume increases, the gap between the configurations widens, highlighting C_3's superior handling of higher loads. At 10,000 messages, C_3 maintains a request time of 10.1 ms, while C_2 records 10.8 ms and C_1 reaches 14.5 ms, resulting in a 30.3% improvement over the default configuration.

At low to moderate loads, C_3 consistently achieves the lowest request times, making it highly efficient for low-latency real-time operations. However, at extremely high volumes ($\geq$ 40,000 messages), its performance slightly deteriorates relative to C_2 due to increased overhead from key-based partitioning, which introduces routing complexity. For instance, at 40,000 messages, C_3 records 27.1 ms, slightly higher than C_2 at 25.8 ms. In contrast, C_2 benefits from its simpler round-robin strategy, which efficiently distributes large-scale workloads with lower partitioning overhead, making it more suitable for massive-scale batch processing.

C. Jitter

Fig. 6. Jitter vs Messages

The jitter trend shown in Fig. 6 presents a steady increase across all configurations as message volume rises, but with significant performance differences between the setups. Jitter measures the variability in message delivery times, and lower

Messages	Latency (ms)			Kafka Request Time (ms)			Jitter (ms)			Throughput (msgs/sec)		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
100	21.4	15.8	**13.2**	8.6	5.2	**4.1**	6.0	4.1	**3.5**	93.5	125.7	**140.8**
250	38.7	28.4	**24.8**	6.1	4.5	**3.7**	4.8	3.8	**3.2**	108.7	140.3	**158.3**
500	45.3	34.7	**30.5**	5.8	4.0	**3.4**	4.2	3.2	**2.8**	113.2	162.8	**182.6**
750	51.2	39.9	**34.2**	6.7	4.3	**3.6**	4.9	3.5	**3.0**	145.3	185.4	**210.4**
1000	57.1	42.1	**36.9**	7.4	4.7	**4.0**	5.1	3.6	**3.1**	180.4	235.1	**260.4**
1500	64.8	48.9	**43.7**	8.2	5.5	**4.7**	6.3	4.1	**3.8**	182.5	247.3	**275.8**
2500	72.3	56.2	**50.1**	9.5	6.1	**5.4**	7.1	4.9	**4.3**	206.6	280.6	**310.9**
3000	75.6	59.4	**54.3**	10.1	6.8	**6.1**	7.8	5.3	**4.7**	210.2	295.4	**328.5**
5000	84.2	67.1	**62.8**	11.4	7.9	**7.2**	8.3	6.1	**5.5**	212.7	312.9	**345.1**
7500	92.8	75.6	**70.2**	12.7	9.4	**8.8**	9.1	7.3	**6.7**	213.9	317.4	**355.7**
10000	101.5	82.3	**78.5**	14.5	10.8	**10.1**	10.6	8.2	**7.5**	214.3	320.1	**360.8**
12500	109.2	89.4	**85.7**	15.8	12.1	**11.4**	11.9	9.5	**8.9**	215.7	325.7	**366.4**
15000	115.7	95.1	**93.6**	17.3	13.7	**12.9**	13.2	10.9	**10.2**	215.2	324.9	**365.1**
17500	122.6	102.8	**101.4**	19.1	15.4	**14.6**	14.7	12.4	**11.7**	214.5	322.7	**363.2**
20000	130.8	110.3	**109.8**	20.8	16.8	**16.3**	15.9	13.6	**13.1**	213.1	320.4	**360.9**
22500	140.2	118.6	**118.1**	22.4	18.2	**18.0**	17.5	14.9	**14.6**	210.8	318.1	**358.5**
25000	150.5	127.5	**127.4**	24.3	19.6	**19.3**	18.7	15.7	**15.4**	208.3	316.4	**355.2**
30000	165.4	**139.3**	139.9	27.6	22.1	**21.7**	20.1	18.4	**18.1**	205.9	312.8	**350.6**
35000	180.9	152.6	**152.5**	30.2	24.3	**24.2**	22.5	20.9	**20.6**	200.7	308.7	**345.3**
40000	190.2	**160.9**	165.8	34.1	**25.8**	27.1	25.3	**22.7**	23.4	195.4	298.1	**340.1**
45000	205.7	**174.3**	178.3	37.5	**28.9**	29.5	28.1	**25.4**	25.8	180.2	282.6	**325.4**
50000	218.9	**185.7**	192.7	40.9	**31.4**	32.4	30.4	28.6	**28.5**	165.8	265.2	**310.2**

TABLE II

KAFKA PERFORMANCE METRICS ACROSS CONFIGURATIONS

jitter indicates more consistent and predictable message propagation.

At low message volumes, C_3 achieves the lowest jitter, highlighting its superior consistency due to its higher broker count, increased partitioning, and key-based strategy, which enhance data locality and reduce message queuing delays. For 100 messages, C_3 records a jitter of 3.5 ms, which is 14.6% lower than C_2 (4.1 ms) and 41.7% lower than C_1 (6.0 ms). Similarly, at 250 messages, C_3 achieves 3.2 ms, while C_2 reaches 3.8 ms and C_1 shows 4.8 ms, indicating a 33.3% improvement over the baseline.

As the message volume increases, C_3 continues to demonstrate more stable jitter compared to the other configurations. At 5,000 messages, C_3 records 5.5 ms, while C_2 shows 6.1 ms and C_1 spikes to 8.3 ms, resulting in a 33.7% improvement over the baseline. This is due to C_3's key-based partitioning, which ensures related messages are routed to the same partition, reducing rebalancing overhead and improving consistency.

At higher loads ($\geq$ 40,000 messages), the performance gap narrows, with C_2 and C_3 exhibiting almost identical jitter. For instance, at 50,000 messages, C_3 records 28.5 ms, just 0.1 ms lower than C_2 (28.6 ms). This convergence is due to the network saturation and processing bottlenecks that arise at massive volumes, limiting the advantage of key-based routing in C_3. Meanwhile, C_2 benefits from its simpler round-robin distribution, which reduces complexity during large-scale parallel processing, resulting in almost equivalent jitter values.

Overall, C_3 consistently maintains the lowest jitter at low and moderate loads, making it ideal for time-sensitive, real-time applications where consistent message delivery is essential. However, at extremely high volumes, C_2 achieves near-parity with C_3, making it equally viable for large-scale batch processing scenarios.

D. Throughput

Figs. 7, 8, & 9 illustrate the throughput performance of three Kafka configurations (C_1, C_2, C_3) under Low, Medium and High conditions respectively.

1) **Low Load**

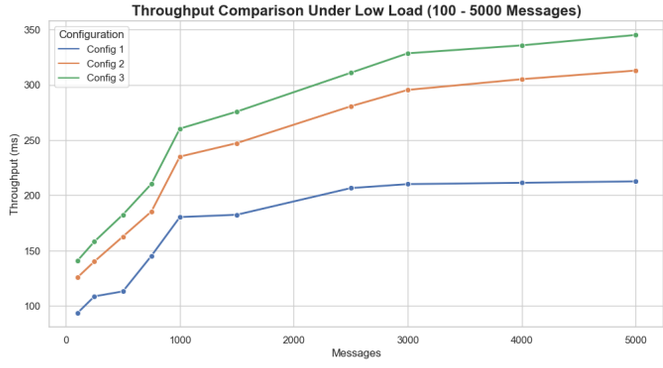


Fig. 7. Throughput under Low Load

At low message volumes, C_3 consistently achieves the

highest throughput, followed by C_2, with C_1 trailing behind, as depicted in Fig. 7. This performance trend underscores the effectiveness of C_3's increased parallelism, which is driven by a higher broker count, expanded partitioning, and efficient key-based message distribution. These architectural optimizations enable parallel consumers to process messages concurrently, reducing contention and significantly enhancing overall throughput. For instance, at 100 messages, C_3 achieves 140.8 msgs/sec, while C_2 processes 125.7 msgs/sec, and C_1 manages only 93.5 msgs/sec. As the message volume increases, the throughput gap widens. By the time the system reaches 5000 messages, C_3's throughput rises to 345.1 msgs/sec, whereas C_2 achieves 312.9 msgs/sec, and C_1 lags behind at 212.7 msgs/sec. This significant performance gap highlights how the enhanced partitioning and broker distribution in C_3 allows it to handle larger message loads more efficiently.

The throughput advantage of C_3 at low loads underscores the importance of parallelism and optimized resource allocation in Kafka. The use of multiple brokers and increased partitioning reduces the burden on individual nodes, enabling superior message distribution. Conversely, C_1 struggles due to its single broker bottleneck and limited partitioning capabilities, resulting in lower throughput, causing its performance to degrade quickly as the load increases, making it unsuitable for even moderately scaled Kafka operations.

2) Medium Load

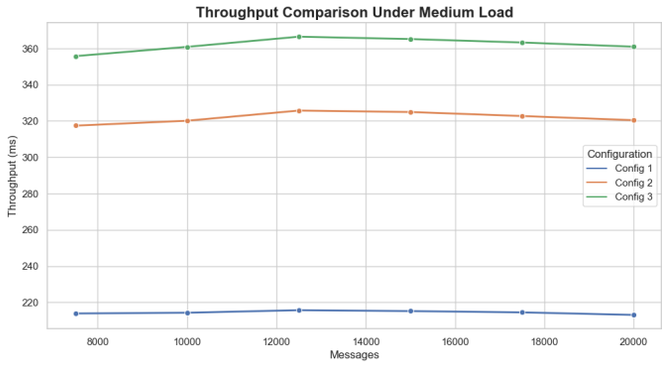


Fig. 8. Throughput under Medium Load

At medium message volumes, the throughput performance across the configurations stabilizes and exhibits a plateauing trend, highlighting the diminishing returns of scaling Kafka infrastructure. C_3 consistently outperforms C_2, although the gap between them narrows as the load increases. This convergence is driven by Kafka's internal overhead, including replication synchronization, disk I/O, and inter-broker coordination, which cap the performance gains from additional brokers and partitions. As the graph in Fig. 8 illustrates, both C_3 and C_2 reach a near-saturation point, where throughput improvements become marginal despite the rising message volume. This plateauing behavior reveals that Kafka's resource utilization becomes less efficient at medium loads, as the added replication and coordination costs counteract the benefits of parallelism. C_1, with its single-broker architecture, falls significantly behind, reflecting its limited scalability and increased bottleneck effect under moderate traffic conditions.

3) High Load

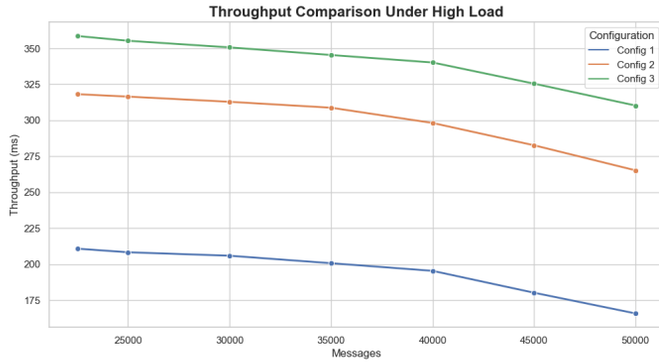


Fig. 9. Throughput under High Load

At high message volumes, the throughput across all configurations begins to decline, indicating the onset of resource saturation and Kafka's diminishing processing efficiency. While C_3 continues to lead, the performance gap between C_3 and C_2 narrows even further. Both configurations plateau and eventually exhibit a downward trend as the system reaches its capacity limits. The throughput dip is caused by intensified replication overhead, increased disk I/O contention, and inter-broker communication delays, which collectively reduce Kafka's ability to sustain higher message rates. The graph in Fig. 9 reveals that C_3 initially maintains the highest throughput, but its performance deteriorates faster due to its higher replication factor, which amplifies coordination overhead under heavy load. In contrast, C_2, with its lower replication burden, sustains throughput more effectively, closing the gap with C_3. This convergence highlights how reducing replication overhead enhances throughput efficiency during large-scale message processing. C_1, which is already constrained by its single-broker architecture, experiences a steeper throughput decline, as its limited capacity and lack of parallelism make it highly susceptible to saturation. The bottlenecked architecture struggles to manage the growing message load, resulting in a sharper drop in throughput.

VI. Conclusion and Key Findings

This study comprehensively evaluated the performance of a Kafka-based supply chain architecture under three configurations (C_1, C_2, and C_3) across varying load levels (low,

medium, and high). The results highlight significant performance differences in terms of latency, throughput, request time, and jitter, revealing key insights into the efficiency and scalability of Kafka-based event streaming systems.

1) **Superior Performance of C_3 at Low and Medium Loads**
   - Throughout the low and medium load scenarios, C_3 consistently achieved the lowest latency and jitter due to its higher broker count, increased partitioning, and key-based message distribution.
   - At low loads (1,000 to 10,000 messages), C_3 exhibited:
     - 35.3% lower latency and 41.7% lower jitter compared to C_1.
     - 17.0% higher throughput than C_2, making it the most efficient configuration for real-time event streaming.
   - The reduced request times and stable jitter values indicate superior consistency and reliability in C_3 at small-to-medium message volumes.
2) **Convergence of C_2 and C_3 at Higher Loads**
   - At higher message volumes (30,000 to 50,000), the performance gap between C_2 and C_3 narrows due to replication-induced overhead in C_3.
   - C_2 outperforms C_3 slightly in terms of latency and jitter at extremely high loads, making it better suited for batch processing scenarios.
   - C_2's simpler round-robin distribution and lower replication factor reduce overhead, enhancing its efficiency for large-scale workloads.
3) **Kafka Saturation at Extremely High Loads**
   - As the message volume increases beyond 40,000, all configurations experience diminishing throughput and rising latency due to Kafka's disk I/O contention, inter-broker coordination, and replication overhead.
   - The throughput decline highlights Kafka's processing saturation point at massive loads, requiring further optimizations such as broker scaling, dynamic partitioning, and consumer rebalancing to sustain efficiency.
4) **Real-World Implications:**
   - For low-latency, real-time supply chain event processing, C_3 is the most effective configuration, offering superior responsiveness and stability.
   - For large-scale, high-volume batch processing, C_2 offers better stability with lower replication overhead, making it more suitable for throughput-centric workloads.
   - C_1, with its single-broker setup, is inefficient for all but the smallest workloads, making it unsuitable for production-scale Kafka operations.

Overall, the study demonstrates that C_3 offers the most consistent and scalable performance for real-time streaming at small-to-medium loads, while C_2 becomes more efficient for large-scale batch processing. These findings provide practical insights for designing and optimizing Kafka-based supply chain architectures, enabling high throughput, low-latency, and fault-tolerant event streaming.

REFERENCES

[1] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: yesterday, today, and tomorrow*. 01 2017.

[2] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybylek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:1–1, 01 2022.

[3] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[4] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. 2015.

[5] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[6] Jay Kreps, Neha Narkhede, and Jun Rao. Benchmarking apache kafka: 2 million writes per second on three cheap machines, 2016. LinkedIn Engineering Blog.

[7] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.

[8] V Ganesh Tejas and Dr. Hemavathy R. Microservices and its intercommunication using kafka. *International Research Journal of Engineering and Technology (IRJET)*, 07(04), April 2020.

[9] Kiran Peddireddy. Enhancing inventory management systems through kafka data stream integration. *International Advanced Research Journal in Science, Engineering and Technology*, 8(9), September 2021.

[10] Sören Henning and Wilhelm Hasselbring. Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. *Journal of Systems and Software*, 208:111879, 2024.

[11] Suwarna Shukla and Prabhneet Singh. Revolutionizing supply chain management: Real-time data processing and concurrency management using kafka and akka. *International Journal of Innovative Science and Research Technology*, 9(5), May 2024.

[12] Chandrakanth Lekkala. Designing high-performance, scalable kafka clusters for real-time data streaming. 01 2021.

[13] J. Rao. How to choose the number of topics/partitions in a kafka cluster?, March 2015. [Posted 12-March-2015, Accessed 06-April-2022].

[14] Satish Krishnamurthy, Ashvini Byri, Ashish Kumar, Dr. Satendra Pal Singh, Om Goel, and Prof. Dr. Punit Goel. Utilizing kafka and real-time messaging frameworks for high-volume data processing. *(IJPREMS)*, 02(02), February 2022.

[15] Theofanis P. Raptis and Andrea Passarella. On efficiently partitioning a topic in apache kafka. May 2022.

[16] Sameer Shukla. Exploring the power of apache kafka: A comprehensive study of use cases. *International Journal of Latest Engineering and Management Research (IJLEMR)*, 08(03), March 2023.

[17] Tejas V and Dr V. Development of kafka messaging system and its performance test framework using prometheus. *International Journal of Recent Technology and Engineering (IJRTE)*, 9:1622–1626, 05 2020.

[18] Guenter Hesse, Christoph Matthies, and Matthias Uflacker. How fast can we insert? an empirical performance evaluation of apache kafka. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, page 641–648. IEEE, December 2020.

[19] Roman Wiatr, Renata Słota, and Jacek Kitowski. Optimising kafka for stream processing in latency sensitive systems. In *7th International Young Scientist Conference on Computational Science*, 2018.

[20] Cloudera Inc. *Tuning Apache Kafka Performance*, 2024. Accessed: 2025-02-27.

Optimizing Kafka-Based Supply Chain Architectures:

by Simran Ahuja

Submission date: 22-Apr-2025 11:29AM (UTC+0530)
Submission ID: 2653236763
File name: Optimizing_Kafka.pdf (661.89K)
Word count: 6114
Character count: 35708

Optimizing Kafka-Based Supply Chain Architectures:

ORIGINALITY REPORT

0%	0%	0%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

Exclude quotes	On	Exclude matches	< 1%
Exclude bibliography	On		

Industry / Inhouse:
Research / Innovation:

Project Evaluation Sheet 2024-25(Sem 8)

Class: D17A/B/C

Title of Project (Group no): Drug Inventory and Supply Chain Management Using Microservices (Group 2)

Mentor Name & Group Members: Dr. Nupur Giri, Simran Ahuja (2), Jesica Bijju (10), Rejal Datir (14), Sania Khan (36)

	Engineering Concepts & Knowledge (5)	Interpretation of Problem & Analysis (5)	Design / Prototype (5)	Interpretation of Data & Dataset (3)	Modern Tool Usage (5)	Societal Benefit, Safety Consideration (2)	Environment Friendly (2)	Ethics (2)	Team work (2)	Presentation Skills (3)	Applied Engg & Mgmt principles (3)	Life - long learning (3)	Professional Skills (5)	Innovative Approach (5)	**Total Marks (50)**
Review of Project Stage 1	5	4	3	2	5	2	2	2	2	3	2	3	4	5	44

Comments: Smart contract deployment- Parameters dashboard.

Name & Signature Reviewer1

	Engineering Concepts & Knowledge (5)	Interpretation of Problem & Analysis (5)	Design / Prototype (5)	Interpretation of Data & Dataset (3)	Modern Tool Usage (5)	Societal Benefit, Safety Consideration (2)	Environment Friendly (2)	Ethics (2)	Team work (2)	Presentation Skills (3)	Applied Engg & Mgmt principles (3)	Life - long learning (3)	Professional Skills (5)	Innovative Approach (5)	**Total Marks (50)**
Review of Project Stage 1	5	4	3	2	5	2	2	2	2	3	2	3	4	5	44

Comments: Traffic Simulation

Date: 01/03/2025

Name & Signature Reviewer2

(7)

Project Evaluation Sheet 2024 - 25

Title of Project: Optimizing Kafka - based Supply Chain architectures : A comprehensive performance analysis

Group Members: Suriya Ahuja (02), Sejal Datir (14), Jesica Biju (10), Sania Khan (36)

Engineering Concepts & Knowledge (5)	Interpretation of Problem & Analysis (5)	Design / Prototype (5)	Interpretation of Data & Dataset (3)	Modern Tool Usage (5)	Societal Benefit, Safety Consideration (2)	Environment Friendly (2)	Ethics (2)	Team work (2)	Presentation Skills (2)	Applied Engg&Mgmt principles (3)	Life - long learning (3)	Professional Skills (3)	Innovative Approach (3)	**Research Paper** (5)	**Total Marks** (50)
5	4	4	3	5	2	2	2	2	2	3	3	3	3	3	46

Comments: _____

Name & Signature Reviewer1

Inhouse/ Industry _ Innovation/Research:

Engineering Concepts & Knowledge (5)	Interpretation of Problem & Analysis (5)	Design / Prototype (5)	Interpretation of Data & Dataset (3)	Modern Tool Usage (5)	Societal Benefit, Safety Consideration (2)	Environment Friendly (2)	Ethics (2)	Team work (2)	Presentation Skills (2)	Applied Engg&Mgmt principles (3)	Life - long learning (3)	Professional Skills (3)	Innovative Approach (3)	**Research Paper** (5)	**Total Marks** (50)
	4	4	3	5	2	2	2	2	2	3	3	3	3	3	46

Comments: _____

Date: 1st April,2025

Name & Signature Reviewer 2