INTERNATIONAL
HELLENIC
UNIVERSITY

Real-Time Collaboration Platform Using WebSockets and Event-Driven Arcchitecture

Christos Stylidis

SID: 205364504177

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

JANUARY 2025

THESSALONIKI – GREECE

INTERNATIONAL HELLENIC UNIVERSITY

Real-Time Collaboration Platform Using WebSockets and Event-Driven Architecture

Christos Stylidis

SID: 205364504177

Supervisor:	Prof. Leonidas Akritidis
Supervising Committee Members:	Assoc. Prof. Panayiotis Bozanis
	Assist. Prof. Dimitrios Karapiperis

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

JANUARY 2025

THESSALONIKI – GREECE

Abstract

This dissertation explores the development of a **real-time ticketing system** leveraging **WebSockets and an event-driven architecture**. The platform facilitates instant updates, allowing users, agents, and admins to seamlessly interact with ticket data. The focus of this project was to address challenges in real-time data synchronization, ensuring scalability and maintaining role-base access control.

The system was implemented using **Node.js, Express.js, MongoDB, and Kafka, Authentication Setup** (JWT) to enable real-time updates for ticket creation, assignment, and status changes. Websocket integration ensures live feedback for dashboard operation, while Kafka serves as the backbone for reliable message queueing between components. The codebase includes role-based dashboards for users, agents, and admins, supporting dynamic operations like viewing, creating and updating tickets.

Acknowledgment is extended to my supervisor, **Leonidas Akritidis**, and committee members **Panayiotis Bozanis** and **Dimitrios Karapiperis** for their valuable insights and support. Gratitude is also expressed to the **IHU faculty and peers** for their feedback encouragement throughout this project.

Christos Stylidis

23.12.2024

Table of Contents

1. Introduction

1.1. Overview

The growing reliance on real-time communication systems in various domains has emphasized the need for robust, scalable, and efficient solutions. Real-time systems have become critical in ensuring seamless interactions, particularly in industries requiring instantaneous updates, such as customer support, logistics, and collaborative platforms. These systems aim to enhance user experiences by delivering immediate feedback and reducing delays.

Despite advancements, traditional systems often face challenges in managing high-volume, bi-directional communication and ensuring consistency across different user roles. These limitations become particularly evident in scenarios involving multiple stakeholders, such as users, agents, and administrators, working collaboratively within a system. The need for a reliable solution synchronize data, ensure real-time updates, and maintain role-based access has never been more crucial.

This dissertation system designed to address these challenges. By leveraging WebSockets and an event-driven architecture, the system ensures instantaneous updates for ticket creation, status changes, and assignment operations. Built using Node.js, Express.js, MongoDB, and Kafka, the platform incorporates scalable and efficient communication mechanisms while maintaining data integrity. The proposed solution aims to offer an intuitive responsive user experience across dashboards for users, agents, and administrators.

1.2.Motivation

The increasing reliance on real-time systems across industries such as e-commerce, customer support, logistics, and collaborative platforms has highlighted the critical need for systems that an deliver instantaneous updates and seamless interactions. Traditional communication systems often struggle to meet these demands due to latency issues, lack of scalability, and difficulty in maintaining data consistency among various stakeholders. As businesses increasingly shift towards digital and interconnected operations, the need for reliable and scalable real0time communication platforms has become more pressing than ever.

Despite advancements in technology, many existing solutions fail to effectively address challenges like role-based access control, real-time data synchronization, and handling high-volume data streams efficiently. Users often face delays in receiving updates, and administrators encounter difficulties in managing operations dynamically. These limitations can lead to customer dissatisfaction, operational inefficiencies, and ultimately, business losses.

The dissertation is motivated by the opportunity to bridge these gaps by leveraging modern technologies like WebSockets, Kafka, and an event-driven architecture to develop a robust real-time ticketing system. The proposed system not only addresses the technical limitations of existing solutions but also provides a scalable and intuitive platform that ensures consistent and secure interactions for all users-be they customers, agents, or administrators. The project aims to contribute to the ongoing evolution of real-time communication systems, offering innovative approaches to tackle long0standing challenges and setting a foundation for future advancements in this field.

1.3.Objectives

The primary objective of this dissertation is to develop a robust and scalable real-time ticketing system that addresses the challenges of modern communication and operational needs in various domains. Be leveraging advanced technologies such as WebSockets, Kafka, and an event-driven architecture, the system aims to provide instantaneous updates. Seamless interactions, and enhanced role-based access for users, agents, and administrators.

Specific objectives include:

1. **Real-Time Updates:** To enable instantaneous updates for ticket creation, assignment, and status changes, ensuring a smooth and uninterrupted flow of information among stakeholders.

2. **Data Consistency and Synchronization:** To ensure that all users have access to the most up-to-date information by maintaining real-time data synchronization across the system.
3. **Role-Based Access Control:** To implement secure and efficient access controls that differentiate user privileges and ensure that each role-whether a customer, agent, or administrator-can perform its designated tasks without interference.
4. **Scalability:** To design the system to handle high volumes of concurrent users and data streams without compromising performance or reliability.
5. **User Experience:** To deliver an intuitive and responsive interface for all users, enabling effortless navigation and task completion.
6. **System Reliability and Security:** To build a system that ensures data integrity, prevents unauthorized access, and provides fault-tolerant mechanisms to handle potential systems failures.

This dissertation aims not only to address existing limitations in real-time communication systems but also to set a foundation for future enhancements, supporting scalable and innovative solutions in the field.

1.4.Structure

This dissertation is organized into several chapters, each addressing specific aspects of the development and implementation of real-time ticketing system:

1. **Chapter 1: Introduction**

   This chapter provides an overview of the project, its motivation, and objectives. It sets the foundation for understanding the problem domain and outlines the structure of the dissertation.
2. **Chapter 2: Literature Review**

This chapter surveys existing research and technologies related to real-time communication systems, event-driven architectures, and role-based access control. It highlights the strengths and limitations of current solutions and positions this project within the broader research landscape.

3. Chapter 3: System Design and Architecture

This chapter delves into the architectural design of the system, detailing the use of technologies such as WebSockets, Kafka, and MongoDB. It explains the rationale behind the chosen design and how it addresses the identified challenges.

4. Chapter 4: Implementation

This chapter provides a comprehensive discussion of the implementation process, including the development of key system features like real-time updates, fault-tolerant mechanisms, and user role management.

5. Chapter 5: Results and Evaluation

This chapter evaluates the system's performance, scalability, and reliability. It includes testing results and discusses how the system meets the objectives outlined earlier.

6. Chapter 6: Conclusions and Future Work

This chapter summarized the contributions of the dissertation and suggests potential direction for future research and development

7. Appendices

Additional materials, including code snippets, diagrams, and datasets, are provided here for reference.

2. Literature Review

2.1.Overview

The rapid advancements in technology have reshaped the landscape of communication systems, making real-time data processing and exchange an essential component of modern applications. From e-commerce platforms to collaborative tools and customer support systems, the demand for seamless and instantaneous communication has become a cornerstone for businesses and organizations seeking to enhance user experiences and operational efficiency.

Traditional communication systems, despite their widespread adoption, often fall short in addressing the complexities of real-time data exchange. These systems are typically hindered by latency issues, scalability challenges, and a lack of mechanisms to ensure data consistency and reliability. Recent research into data streaming architectures has emphasized the necessity of efficient, scalable frameworks for managing large-scale, real-time data streams, such as those encountered in modern ticketing and e-commerce systems [6]. As industries become increasingly communication has grown significantly.

This dissertation builds upon the foundation of existing research and leverages cutting-edge technologies such WebSockets, Kafka, and event-driven architectures to develop a real-time ticketing system. By addressing key challenges such as role-based access control, real-time updates, and high-volume data management, this work aims to contribute to the ongoing evolution of real0time communication systems. The subsequent sections will explore existing literature in this domain, critically analyzing related work and identifying gaps that this project seeks to address.

2.2.Related Work

The purpose of this section is to explore existing research and systems relevant to real-time communication platforms, focusing on the technologies and methodologies that address challenges such as scalability, latency, and role-based control.

By critically examining related work, this dissertation aims to identify the strengths and limitations of existing solutions, providing a foundation for understanding how the proposed system contributes to the field.

This section will focus on systems and research that utilize WebSockets, Kafka, and event-driven architectures, discussing their applications, performance, and shortcomings in real-time communication.

2.3.Critical Analysis

This section evaluates existing real-time communication systems and methodologies by analyzing their strengths and limitations. The analysis focuses on key aspects such as scalability, latency, fault tolerance, and role-based access control, which are critical to modern real-time applications. By identifying gaps in current approaches, this section sets the foundation for justifying the need for the proposed solution outlined in this dissertation.

2.3.1. Strength of Existing Systems

Scalability

Scalability is a critical aspect of real-time systems, enabling them to handle increasing loads without compromising performance. It ensures that systems can grow efficiently to meet users demands, whether through hardware upgrades or distributed architectures. Technologies like Kafka exemplify scalability through features such as horizontal scaling, where additional brokers can be added to the cluster to distribute workloads effectively [1].

Distributed architectures form the foundation of scalable systems, ensuring efficient load balancing and fault tolerance under heavy traffic conditions. Such systems are crucial for high-volume environments like e-commerce or real-time ticketing platforms [8].

For instance, Kafka's message partitioning mechanism allows data to be segmented and processed in parallel across multiple nodes, ensuring high throughput even under heavy traffic [1]. Moreover, scalable architectures, such as those designed with distributed and decoupled services, further enhance system reliability and performance by minimizing bottlenecks and resource contention [8]

This capability is particularly valuable for systems like e-commerce platforms and ticketing systems that experience significant traffic surges during peak periods. By supporting seamless scalability, these systems can maintain responsiveness and reliability, providing consistent user experiences regardless of demand fluctuations.

Latency

Latency is another key strength of modern real-time systems, ensuring minimal delays in data transmission and processing. Technologies like WebSockets are designed to reduce latency by establishing persistent, bi-directional communication channels between clients and servers [2]. Additionally, databases such as MongoDB are optimized for real-time analytics, enabling rapid data ingestion and processing to support low-latency applications [7]. Unlike traditional HTTP request-response mechanisms, WebSockets allow for instant data exchange, eliminating the overhead of repeatedly establishing connections [2]. This efficiency in communication aligns with the principles of distributed real-time systems, ensuring minimal delays in high-throughput environments [1].

For example, in collaborative platforms or ticketing systems, low latency enables near-instantaneous updates, such as notifying agents about new tickets or updating users on ticket status changes. This responsiveness is essential for maintaining user satisfaction and operational efficiency, especially in scenarios requiring immediate feedback and decision-making.

Fault Tolerance

Fault tolerance is a critical aspect of robust real-time systems, ensuring reliability and uninterrupted operations even in the event of component failures. Technologies like Kafka inherently support fault tolerance by utilizing distributed architecture and replication [1]. Additionally, microservices architectures bring fault isolation by separating services into independent components, reducing the scope of failures and ensuring system resilience [9].

This design ensures that data remains accessible and consistent even if one or more brokers fail, maintaining the integrity of real-time data streams.

For example, in high-stakes applications like real-time ticketing systems, fault tolerance ensures that tickets are not lost during transmission or processing due to system failures. By leveraging Kafka's replication capabilities and failover mechanisms, real-time systems can handle unexpected disruptions, providing a seamless experience for users and administrators alike.

Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) plays a crucial role in ensuring security and operational efficiency in real-time systems. By assigning permissions and access levels based on user roles, such as administrators, agents, and end-users, RBAC ensures that sensitive operations are only accessible to authorized individuals [2]. In addition, modern RBAC mechanisms often integrate microservices architectures, which enhance scalability and flexibility by decoupling roles and permissions into independent services [9]. Such approaches allow for dynamic updates and faster propagation of access control changes, making them particularly effective in distributed systems.

This minimizes the risk of unauthorized actions and protects the integrity of the system.

Existing systems effectively utilize RBAC to streamline workflows and enforce security policies [3]. For instance, microservices-based RBAC implementations allow for fine-grained control and flexible updates across distributed systems [9]. In ticketing platforms, administrators can manage and monitor tickets, agents can handle assigned tasks, and users can create and track tickets within their permissions. This structured access control enhances user accountability and prevents accidental or malicious misuse of the system, fostering a secure and well-organized environment [3].

2.3.2. Weaknesses of Existing Systems

Scalability Issues

Although significant progress has been made in developing real-time communication systems, scalability continues to be a pressing challenge. As the number of users and the volume of data increase, many traditional systems struggle to maintain efficiency, leading to performance bottlenecks, slower response times, and operational disruptions during peak usage.

Furthermore, the inability of some systems to dynamically allocate resources in response to sudden spikes in activity exacerbates scalability challenges. This limitation often results in degraded user experiences, particularly in industries like e-commerce, logistics, and collaborative platforms, where real-time communication is critical.

Latency and Performance Bottlenecks

While modern communication systems such as those leveraging WebSockets and Kafka offer reduced latency compared to traditional architectures, performance bottlenecks can still emerge in specific scenarios. Systems relying on real-time data processing must ensure minimal delay to maintain user satisfaction, yet challenges like high server loads, inefficient resource allocation, or suboptimal network conditions can introduce significant latencies.

For example, WebSocket-based systems depend heavily on maintaining active connections, which can degrade in performance in proper scaling mechanisms, such as distributed servers or load balancers, are not in place. Similarly, Kafka, while efficient in handling large data streams, may encounter delays if producers or consumers are not optimized for high-throughput environments or if disk I/O becomes a bottleneck.

Addressing latency issues requires a multi-faceted approach, including the optimization of network protocols, the introduction of caching mechanisms, and the effective distribution of computational resources. Such measures are critical in ensuring that performance bottlenecks do not undermine the reliability of real-time systems.

Fault Tolerance Limitations

Fault tolerance is a critical aspect of real-time communication systems, ensuring uninterrupted service and data integrity even in the presence of failures. Systems like Kafka and WebSockets are inherently designed to handle faults to some extent, but challenges still arise when failures occur at scale or across multiple components.

For instance, Kafka's replication mechanism allows data to persist even if one or more brokers fail. However, if replication factors are not adequately configured or if the leader election process becomes delayed, it can impact the overall system availability. Similarly, in WebSocket-based systems, a dropped connection due to server failure can disrupt real-time communication unless reconnection mechanism and failback strategies are implemented effectively.

Modern fault-tolerant systems also need to address challenges like:

- **Partition loss** in distributed setups.
- Ensuring **state consistency** across nodes.
- Implementing robust **retry mechanisms** to recover from transient failures.

By identifying and addressing these issues, the proposed solution in this dissertation aims to ensure resilience, minimizing downtime and maintaining consistent service levels in real-time environments.

Role-Based Access Control Constraints

Role-Based Access Control (RBAC) mechanisms are widely employed in modern systems to ensure secure and restricted access to resources based on user roles and permissions. While RBAC systems provide a structured framework for managing user access, they are not without limitations, especially in the context of real-time applications.

One of the key challenges is RBAC implementations lies in **scalability**. As the number of users and roles increases, the complexity of managing permissions and ensuring consistent enforcement grows exponentially. Systems with frequent role or permission updates may face delays in propagating changes, leading to potential security gaps or operational inefficiencies.

Another limitation is the **lack of flexibility** in adapting to dynamic access needs. Real-time communication systems often involve ad hoc collaboration or temporary access requirements, which traditional RBAC frameworks struggle to accommodate efficiently.

Moreover, **integration issues** arise when combining RBAC with real-time communication technologies like WebSockets. Maintaining synchronized role-based access in distributed architectures, especially during network interruptions or system failures, presents significant challenges. These constraints can hinder the system's ability to provide uninterrupted service and maintain data security [4].

By addressing these constraints, the proposed solution in this dissertation aims to enhance the effectiveness of RBAC mechanisms in real-time environments. It seeks to integrate dynamic role updates, minimize propagation delays and ensure seamless operation across distributed nodes.

2.3.3. Proposed Improvements

Enhancing Scalability

To address the scalability challenges identified in existing systems, the proposed solution employs a distributed architecture leveraging Kafka and WebSockets. By distributing workloads across multiple nodes, the system ensures efficient resource allocation and minimizes bottlenecks during peak usage.

Dynamic resource allocation mechanisms are implemented to adapt to sudden spikers in user activity. This involves leveraging Kafka's partitioning capabilities to distribute messages evenly brokers and scaling WebSocket serves to handle concurrent connections effectively.

Additionally, the system incorporates real-time monitoring and load-balancing strategies to identify and mitigate performance bottlenecks proactively [5]. By continuously analyzing system performance and reallocating resources, the proposed solution achieves greater scalability and reliability in handling high-volume data streams.

Improved Fault Tolerance

To ensure resilience and data integrity, the proposed solution utilizes Kafka's replication mechanism to minimize the impact of broker failures. By distributing data across multiple brokers and implementing a suitable replication factor, the system ensures data availability and fault tolerance in case of node failures.

Advanced fault-tolerance strategies, such as leader election optimization and dynamic reallocation of resources, can further enhance system reliability. For instance, improving the responsiveness of leader election mechanisms during broker failures reduces downtime and prevents disruptions to real-time communication.

Additionally, WebSocket-based systems can incorporate retry and reconnection mechanisms to handle transient network failures effectively. By proactively addressing potential points of failure, these improvements ensure seamless real-time interactions, even under adverse conditions.

The proposed solution aims to address critical challenges such as:

- **Partition recovery** in distributed environments.
- **Leader election stability** to minimize failover latency.
- **Optimized resource utilization** to handle high-volume workloads during failures.

By implementing these enhancements, the system strengthens its ability to deliver consistent and reliable service across real-time environments.

Refining Role-Based Access Control

To address the limitations identified in the existing RBAC implementations, the proposed solution introduces dynamic role management and real-time permission updates. By leveraging a distributed architecture, the system ensures that role-based access policies remain consistent across multiple nodes, even during high-volume operations.

Key improvements include:

- **Dynamic Role Updates:** Incorporating mechanisms to propagate role and permission changes in real time, reducing delays and ensuring seamless enforcement of access policies.
- **Adapting to Temporary Roles:** Enabling the creation of ad hoc or temporary roles for specific tasks or collaborations without compromising system security.
- **Integration with Real-Time Systems:** Ensuring synchronization between RBAC policies and real-time communication technologies such as WebSockets to maintain uninterrupted and secure interactions/

By implementing these enhancements, the proposed solution fosters a secure and adaptable environment, capable of meeting the demands of modern real-time communication systems.

2.4. Summary and Conclusion

The literature review highlighted the rapid advancements in real-time communication systems and their increasing importance across various industries. Technologies such WebSockets, Kafka and event-driven architectures have proven to be effective in addressing many challenges, including scalability, latency, fault tolerance, and secure access control through RBAC.

Despite these advancements, several limitations persist:

- Scalability issues during peak loads sudden spikes in user activity.
- Latency and performance bottlenecks in high-throughput environments.
- Fault tolerance challenges in handling broker failures and ensuring data consistency.
- Constrains in RBAC systems related to scalability, flexibility, and integration with real-time technologies.

The proposed solution addresses these gaps by leveraging dynamic resource allocation, ad vanced fault-tolerance mechanism, and refined role-based access control strategies. By incorporating these improvements, the system aims to provide a robust, scalable, and secure real-time communication platform.

This chapter forms the foundation for the subsequent chapters, which will detail the design, architecture, implementation, and evaluation of the proposed system, showcasing how it addresses the identified gaps and contributes to advancing real-time communication systems.

3. System Design and Architecture

3.1. Overview

The system design of the real-time ticketing platform is centered around a robust architecture that integrates multiple components to ensure scalability, reliability, and seamless real-time communication. By leveraging modern technologies like WebSockets, Kafka, MongoDB, and

Node.js, the system is built to handle high user concurrency while maintaining data integrity and responsiveness.

At the core of the architecture, WebSockets provide a persistent, bi-directional communication channel between clients and the server, enabling instantaneous updates for ticket creation, assignment, and status changes. Kafka serves as the backbone for reliable message queuing and data streaming, ensuring scalability and fault tolerance in handling high-volume transactions. MongoDB is employed as the primary database, offering flexibility in managing dynamic ticketing data and ensuring high performance during read and write operations.

This chapter provides a detailed explanation of the system's architecture outlining how its components interact to achieve the objectives of real-time data synchronization, role-based access control, and fault tolerance. The following sections will delve into the architectural design principles, the choice of technologies, and their integration within the system to address the challenges identified in earlier chapters.

3.2.Architectural Design

The architectural design of the real-time ticketing system is based on a modular and distributed approach, ensuring scalability, fault tolerance, and efficient real-time communication. The system architecture consists of the following core components:

1. **Client-Side Interface:**
    - The frontend is built using React, providing users, agents, and administrators with intuitive dashboards for ticket creation, management, and monitoring.
    - The client communicates with the backend through REST APIs certain operations and WebSocket connections for real-time updates.
2. **Backend Server:**

- Built using Node.js and Express.js the backend handles core business logic, authentication, and communication between the frontend, database, and message broker.
- WebSocket integration ensures live updates for ticket operations, such as creation, assignment, and status changes.

3. **Message Broker (Kafka):**
    - Kafka serves as the event-streaming platform, facilitating reliable message queuing between system components.
    - Although Kafka supports advanced features such as message partitioning and replication for fault tolerance, this implementation focuses on its core capability of ensuring reliable communication and event delivery.

4. **Database (MongoDB):**
    - MongoDB stores ticket data, user roles, and systems logs. Its NoSQL structure allows for flexible data modeling and efficient handling of dynamic ticketing operations.
    - The database ensures data consistency and support high-speed read/write operations.
5. **Role-Based Access Control (RBAC):**
    - A structured RBAC system ensures secure access to the platform. Roles include users, agents, and administrators, each with specific permissions.
    - Token-based authentication using JSON Web Tokens (JWT) provides secure and scalable session management.
6. **Real-Time Communication:**

- WebSocket connections facilitate bi-directional communication between the server and clients, enabling instantaneous updates for ticket activities.

7. **Fault Tolerance Mechanisms:**
    - The system leverage Kafka's inherent reliability for event delivery. Advanced features like replication failover mechanisms, while supported by Kafka, are not configured in this implementation.
    - Basic retry and reconnection mechanism ensure resilience during transient network disruptions.

The architectural design emphasizes modularity, making it easier to extend or modify components without disrupting the overall system. The next section will delve into the detailed workflows and interactions between these components.

3.3.Detailed Workflow and Interactions

Overview of Workflows:

The real-time ticketing system ensures seamless interaction among users, agents, and administrators by employing a modular workflow that facilitates ticket creation, assignment, and status updates. Each workflow is designed to leverage the capabilities of WebSockets for real-time communication and Kafka for reliable message delivery.

For instance, when a user creates ticket, the request flows from the client-side interface to the backend server, where it is validated, processed, and stored in the MongoDB database, Simultaneously, Kafka's event-streaming mechanism ensures that updates are broadcast to all relevant stakeholders, enabling instantaneous visibility across the system.

Interaction Between Components

- **Client-Side Interaction:**

i. Users initiate actions such as ticket creation, updates, or queries through the React-based dashboard.

ii. WebSocket connections maintain a persistent, bi-directional link between the client and server, enabling real-time updates for all subscribed clients without the need for repeated HTTP requests.

- **Backend Server Operations:**

iii. The backend server, built with Node.js and Express.js, processes incoming requests by interacting with MongoDB for data storage and the WebSocket server for real-time communication.

iv. It handles role-based permissions, ensuring that each user can only perform actions within their designated access level.

- **Kafka as a Message Broker:**

v. Kafka is integrated into the architecture as a basic event-streaming platform for message queuing.

vi. The current implementation focuses on sending messages to a single topic (tickets) and consuming those messages for processing. Advanced features like partitioning and multi-consumer setups are not utilized.

- **Database Operations**

vii. MongoDB servers as the system's data repository, storing ticket details, user roles, and log information.

viii. Queries are optimized to ensure high-speed data retrieval for real-time operations, particularly during search and filtering tasks.

Role-Based Interactions

- **Users:**
  - **i.** Users can create new tickets, view existing tickets, and track the progress of their requests in real time.
  - **ii.** Upon ticket creation, the system assigns a unique identifier to the ticket and broadcasts updates to relevant agents and administrators using WebSocket events.
- **Agents:**
  - **iii.** Agents receive ticket assignments through real-time WebSocket updates and can update ticket statuses.
  - **iv.** The system ensures that agents only have access to tickets assigned to their categories or expertise, as determined by ticket properties and agent roles.
- **Administrators:**
  - **v.** Administrators can create and delete tickets, update ticket statuses, and monitor ticket operations across all categories.
  - **vi.** They have global access to all tickets and are responsible for overseeing the overall system's functionality.

Illustrative Diagrams

- **User Dashboard Workflow**

  This workflow focuses on the process of creating a ticket within the User Dashboard. It outlines how users interact with the system to submit a new ticket, starting from filling out the ticket form to its submission. The process demonstrates the seamless integration of the backend, including Kafka for message queuing and MongoDB for ticket storage. Additionally, it highlights how the system ensures real-time updates using WebSocket to notify the User, Agent, and Admin Dashboards about the newly created ticket.

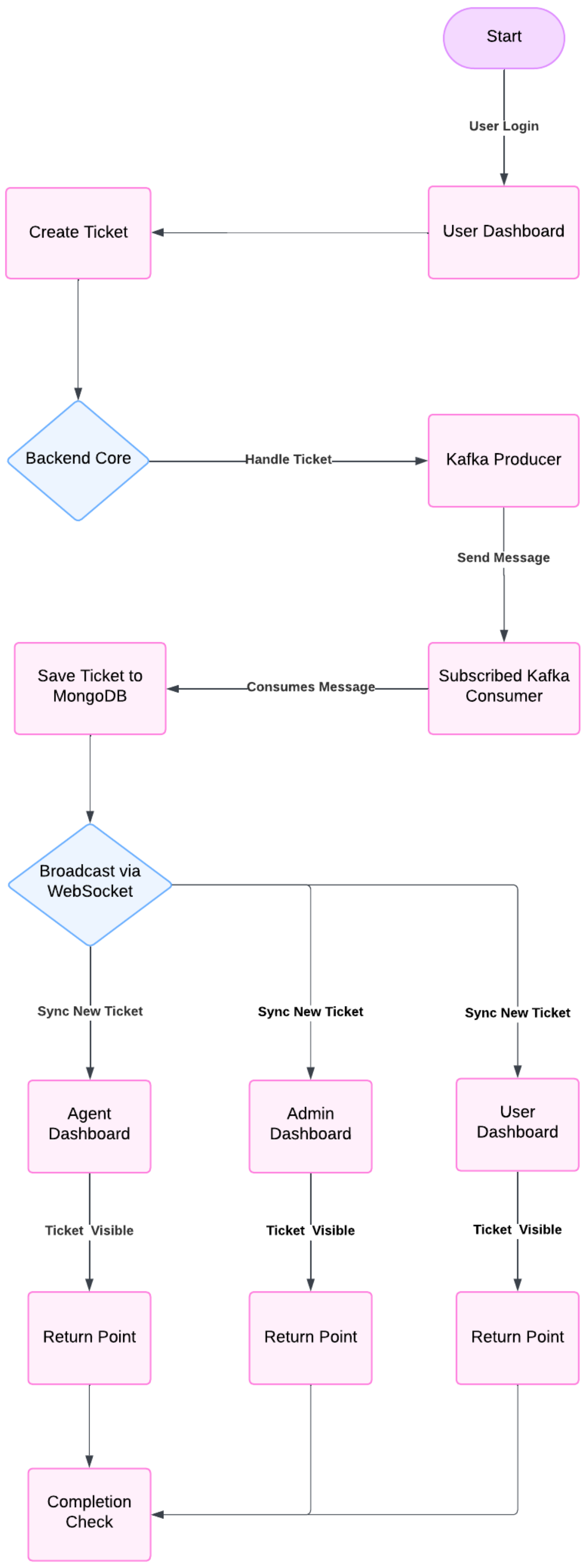
- **Agent Dashboard Workflow**

  This workflow demonstrates the process of updating a ticket within the Agent Dashboard. It begins with an agent logging in and accessing the dashboard, where they can update ticket details such as status. The updated information is processed by the backend core, which handles the updates and broadcasts the changes via WebSocket. This ensures that all connected dashboards, including the User, Admin, and other Agent Dashboards, are synchronized in real time to reflect the updated ticket details.

- **Admin Dashboard Workflow**

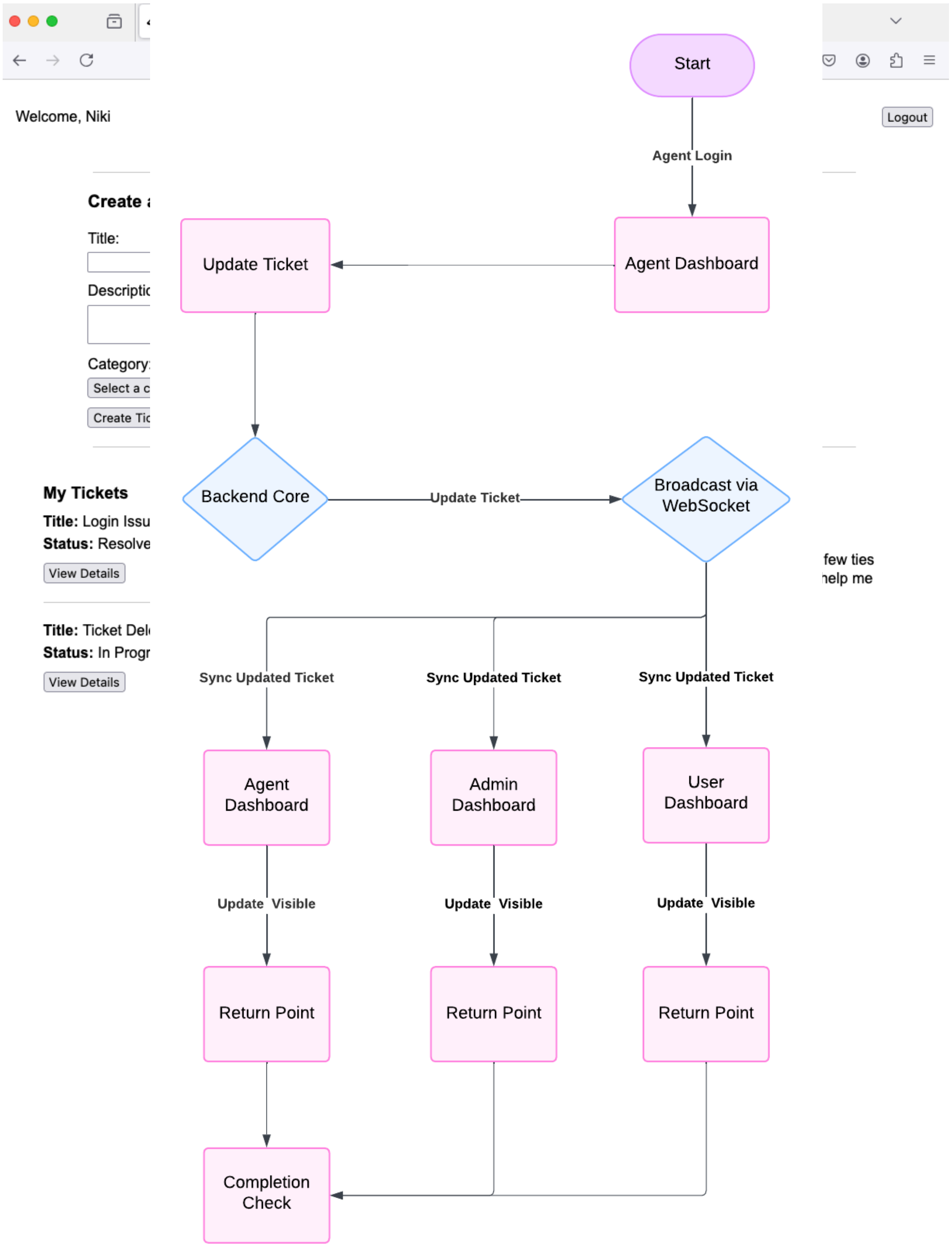
  This diagram illustrates the workflow for the Admin Dashboard in the ticketing system. It highlights ability to view all tickets, create new tickets, update ticket statuses, and delete tickets. The diagram also showcases the integration with the backend for processing updates and broadcasting changes via WebSocket. This ensures all connected dashboards (User, Agent, and Admin) are synchronized in real-time with the latest ticket information.

User Workflow



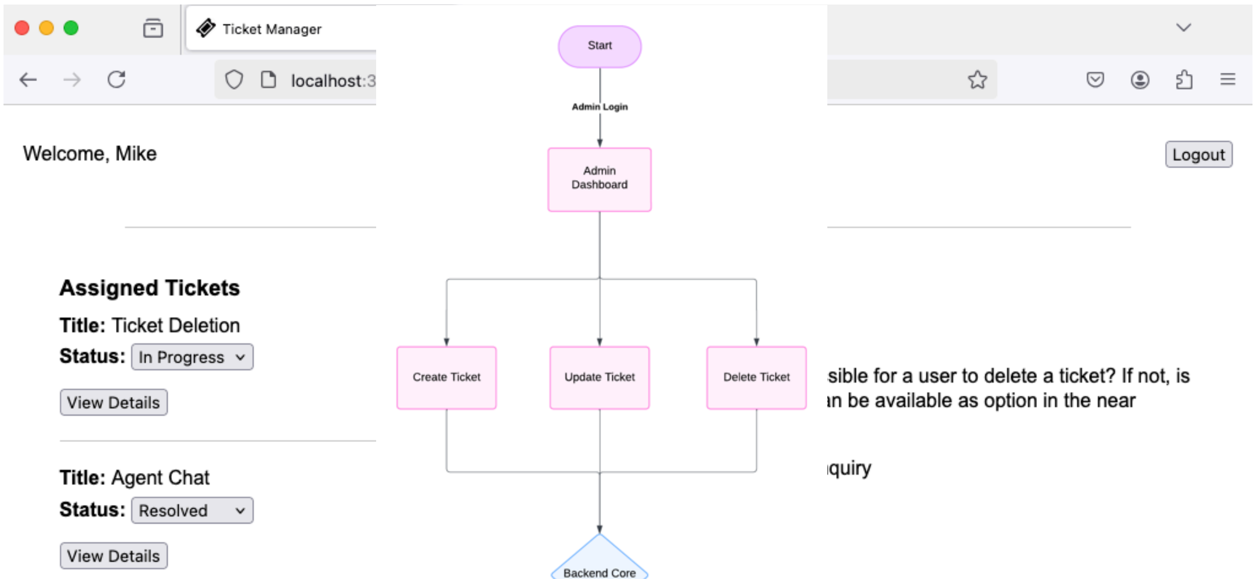
User Dashboard

Agent Workflow


Welcome, Niki
Logout
Create
Title:
Descriptio
Category:
Select a c
Create Tic
My Tickets
Title: Login Issu
Status: Resolve
View Details
Title: Ticket Del
Status: In Progr
View Details
few ties
help me
Start
Agent Login
Agent Dashboard
Update Ticket
Backend Core
Update Ticket
Broadcast via WebSocket
Sync Updated Ticket
Sync Updated Ticket
Sync Updated Ticket
Agent Dashboard
Admin Dashboard
User Dashboard
Update Visible
Update Visible
Update Visible
Return Point
Return Point
Return Point
Completion Check

Agent Dashboard

Admin Workflow



Welcome, Mike

Logout

Assigned Tickets

Title: Ticket Deletion

Status: In Progress

View Details

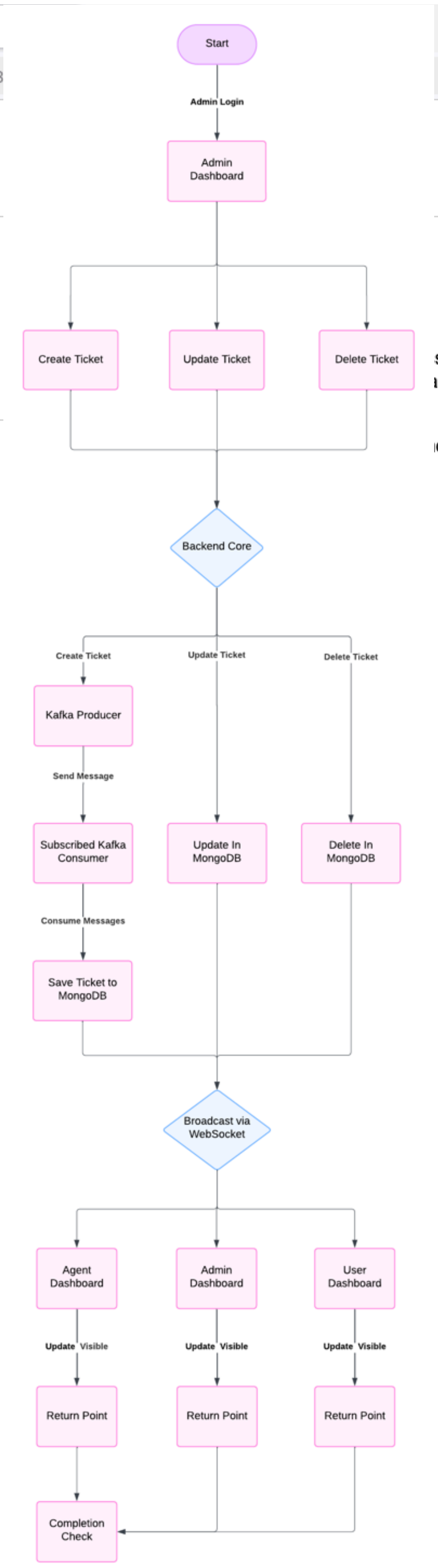
Title: Agent Chat

Status: Resolved

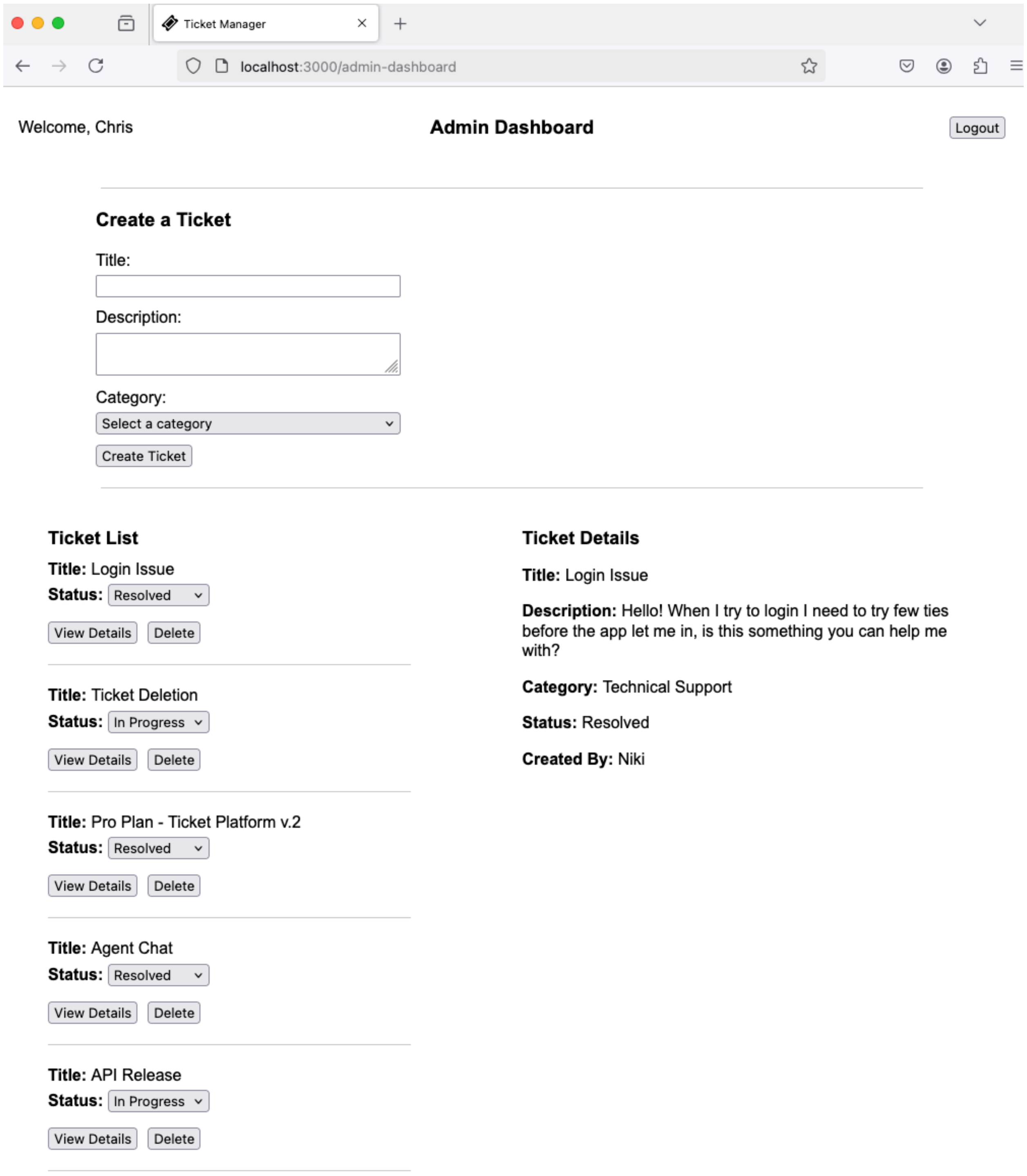
View Details

sible for a user to delete a ticket? If not, is
ın be available as option in the near

ıquiry



Admin Dashboard


Ticket Manager

localhost:3000/admin-dashboard

Welcome, Chris

Admin Dashboard

Logout

Create a Ticket

Title:

Description:

Category:

Select a category

Create Ticket

Ticket List

Title: Login Issue
Status: Resolved
View Details Delete

Title: Ticket Deletion
Status: In Progress
View Details Delete

Title: Pro Plan - Ticket Platform v.2
Status: Resolved
View Details Delete

Title: Agent Chat
Status: Resolved
View Details Delete

Title: API Release
Status: In Progress
View Details Delete

Ticket Details

Title: Login Issue

Description: Hello! When I try to login I need to try few ties before the app let me in, is this something you can help me with?

Category: Technical Support

Status: Resolved

Created By: Niki

Ticket Creation Process

This subsection describes the detailed process for creating tickets in the system. It covers how user requests are handled from the frontend to the backend and the database, followed by broadcasting real-time updates using Kafka and WebSocket.

- **Frontend Validation and Data Submission**

  When a user creates a ticket, the process begins on the User Dashboard. The frontend provides a form with fields for the ticket title, description, and category. Before submission, client-side validation ensures all required fields are completed, adhere to the expected formats, and meet predefined criteria. Once the validation succeeds, the frontend sends the ticket data to the backend via a POST request to the /create-ticket endpoint. This request includes the user's session token (JWT) in the headers, which the backend uses to authenticate the user and verify their authorization to create tickets. This ensures only legitime users can perform this operation, maintaining the integrity and security of the system.

- **Backend Processing**

  Upon receiving the request, the backend performs additional validation to verify the data's integrity and ensure the user is authorized to create tickets. Using Kafka's producer functionality, the backend sends a "new-ticket" message containing the ticket details (e.g., title, description category, and initial status: Waiting) to the designated topic. A Kafka consumer, subscribed to this topic, processes the event and saves the ticket data to the tickets collection in MongoDB. This stored document includes key details such as the ticket's creator, category, and status.

- **Real-Time Updates**

  After a ticket creation request is processed, the backend integrates Kafka and WebSocket to ensure real-time communication across all dashboards. First, the system uses Kafka's producer functionality to send a "new-ticket" message to the designated topic. The message includes all necessary ticket details, such as the little, description, category, and initial status (waiting). A Kafka consumer, subscribed to this topic, processes the event and saves the ticket data to the MongoDB database, ensuring asynchronous and reliable data handling.

Once the Kafka consumer confirms that the ticket has been stored successfully, the backend emits a “ticket-created” event via WebSocket. This event is broadcast to all connected clients, including the User, Agent, and Admin Dashboards, ensuring that all relevant stakeholders are updated in real time. This two-step process of Kafka message handling and WebSocket broadcasting ensures that the system remains both scalable and responsive, with room for future enhancements such as multi-consumer setups or external integrations.

Ticket Management Workflows

- **User Interaction with Created Tickets**

  Once a ticket is created, users can view its details and monitor its progress in real time through the User Dashboard. The dashboard fetches ticket data directly from the tickets collection in MongoDB, using the user’s unique identifier to ensure that only tickets associated with their account are displayed. This is achieved via a secure API call to the /get-tickets endpoint, which validates the user’s session token (JWT) to authorize access.

  Real-time updates are facilitated through WebSocket events such as “status-updated”. For example. When an agent modifies the status of a ticket (e.g., from waiting to in progress), the User Dashboard receives as instant notification and updates the displayed ticket details without requiring a page refresh. This seamless integration between the backend, database, and WebSocket ensures that users are always informed about the latest developments in their tickets.

- **Agent Interaction with Tickets**

  Agents play a key role in managing tickets within the system. Once a ticket is created and assigned to an agent based on its category, the Agent Dashboard provides real-time access to all assigned tickets. The dashboard retrieves ticket data via a secure API call to the /

get-tickets endpoint, using the agent's session token (JWT) to validate the identify and authorization.

Agents can view ticket details, including the ticket's status (waiting, in progress, or resolved), title, and description. If required, agents can update a ticket's status by interacting with the dashboard. When an agent modifies a ticket (e.g., updating its status to in progress), a PUT request is sent to the /update-ticket endpoint. The backend processes the request, validates the agent's authorization, and updates the ticket's status in MongoDB.

Real-time notifications are then triggered via a "status-updated" WebSocket event, which is broadcast to all relevant dashboards. This ensures that users are instantly notified of updates to their tickets, and other agents or admins have the latest information available. The seamless in integration of WebSocket and database updates enhances coordination among agents and improves response times.

- **Admin Role in Ticket Management**

  Admins have full control over ticket operations and management within the system, granting them the highest level of access across all dashboards. Using the Admin Dashboard, they can monitor, create, update, and delete tickets, ensuring efficient oversight of the platform's operations.

  - **View and Monitor Tickets**

    Admins can view and monitor tickets across all categories using the Admin Dashboard. The dashboard retrieves tickets via the common /get-tickets endpoint, where the backend dynamically filters tickets based on the user's role. For admins, all tickets are retrieved, while agents and users only see tickets relevant to their assigned categories or those they created, respectively. The ticket details displayed include the title, description, category, status, and creator. This unified approach ensures that all roles can access their relevant data while maintaining security and efficiency.

  - **Create Tickets**

Like user functionality. Admins can create tickets by submitting a form with details such as title, description, and category. The frontend sends the data to the backends' / create-ticket endpoint, where it is processed and stored in the database. Real-time updates ensure the newly created ticket is visible to all relevant dashboards (e.g., Agent and User Dashboards).

- **Update Ticket Status**

  Admins can update ticket statuses, like agents, through the Admin Dashboard. For example, they can mark a ticket as resolved or in progress depending on the current state of operations. These updates are processed by the /update-ticket endpoint, and corresponding WebSocket events "status-updated" notify all connected clients of the change.

- **Delete Tickets**

  Admins have the exclusive ability to delete tickets. When ad admin deletes a ticket through the dashboard, a DELETE request is sent to the /delete-ticket endpoint. The backend processes the request by verifying the admin's authorization, removing the ticket from MongoDB, and broadcasting a "ticket-deleted" WebSocket event to ensure that all dashboards are updated in real time. This capability allows admins to manage obsolete or duplicate tickets efficiently.

System-Level Workflows and Communication

- **Kafka Producer / Consumer Workflow**

  Kafka servers as a reliable event-streaming platform in the system, facilitating asynchronous communication between components. Its producer / consumer model ensures scalability and fault tolerance by decoupling the ticket creation process into distinct steps.

  The **Kafka Producer**, initialized in the backend, plays a key role during ticket creation. Once the user submits ticket details, the backend's /create-ticket endpoint sends a “new-ticket” event to the Kafka topic named tickets using the sendMessage function. This event includes essential ticket data, such as the title, description, category, and initial status.

Simultaneously, the **Kafka Consumer**, also initialized in the backend, listens to the tickets topic. When a “new-ticket” event is detected, the consumer processes the message by saving the ticket details to the MongoDB database. This separation ensures that data is reliably stored even if downstream services experience temporary issues.

Both the producer and consumer are connected to the Kafka cluster through a shared configuration. For example, the clientId is set to “ticketing-system”, and the Kafka broker is hosted locally at localhost:9092. This lightweight setup effectively integrates Kafka into the system, ensuring seamless communication and data processing.

- **WebSocket Event Flow**

Introduction

WebSockets enable real-time communication between the server and connected clients. Unlike traditional HTTP requests, WebSockets maintain a persistent, bi-directional connection, allowing instant event-based communication. This functionality is critical for synchronizing the User, Agent, and Admin Dashboards in the real-time ticketing system.

Step 1: Connection Establishment

The WebSocket connection begins with the initialization of the server. The backend uses the socket.io library to setup up a WebSocket server, which listens for incoming client connections. Each client (e.g., User Dashboard, Agent Dashboard) establishes a connection to the WebSocket server upon logging in. The server assigns a unique socket ID to every connected client, ensuring individual sessions for real-time communication.

Step 2: Event Emission and Listening

After the WebSocket connection is established, the server and clients communicate via specific event channels. These events allow for real-time updates across all connected dashboards.

1. Backend Event Emission:

The backend emits events when critical actions occur, such as:

- **“ticket-created”:** Sent when a new ticket is created. This event carries ticket details like title, description, and category.
- **“status-updated”:** Sent when a ticket’s status changes (e.g., from “waiting” to “in-progress”).
- **“ticket-deleted”:** Sent when an admin deletes a ticket. The event contains the ticket ID for client synchronization.

2. Client Event Listeners:

Each dashboard (User, Agent, Admin) listens to relevant events using the socket.on() method:

- **User Dashboard**: Listens for “ticket-created,” “status-updated,” and “ticket-deleted” to display real-time changes.
- **Agent Dashboard**: Similar to the User Dashboard but focuses on tickets assigned to the agent’s category.

- **Admin Dashboard**: Primarily listens for general updates and handles admin-specific functionalities like deletion locally.

Step 3: Synchronization and Updates:

Synchronization ensures that any updates made to tickets are reflected in real-time across all connected dashboards (User, Agent, Admin). This step highlisghts how the system achieves consistency and responsiveness:

1. Broadcasting Events:

When a ticket event occurs (e.g., creation, status update, or deletion), the backend broadcasts the corresponding event to all connected clients using the io.emit() method. This ensures that every dashboard receives the update without the need for refreshing or polling.

2. Dashboard Updates:

Each dashboard listens to specific events and updates its interface accordingly:

- **User Dashboard:** Automatically updates the ticket list and status details when events like "status-updated" or "ticket-deleted" are received.
- **Agent Dashboard:** Refreshes the agent's assigned ticket list relevant events occur.
- **Admin Dashboard:** Handles updates efficiently, ensuring administrative actions like ticket deletion reflect in the system instantly.

3. Ensuring Consistency:

To prevent discrepancies, the system validates incoming events and updates only the relevant tickets. For example:

- If a “**ticket-deleted**” event is received, only the corresponding ticket is removed from the interface.
- If a “**status updated**” event is received, the specific ticket’s status is updated while preserving other details.

Step 4: Error Handling and Reconnection:

The WebSocket implementation in the real-time ticketing system ensures stable connections between the server and clients. While error handling is basic, it covers essential scenarios to maintain connectivity.

1. Socket Connection Initialization:

- Each client (e.g., User, Agent, or Admin Dashboard) establishes a WebSocket connection upon loading the dashboard.
- A unique socket.id is generated and logged on the backend to track active connections

2. Automatic Reconnection:

- When a WebSocket connection is interrupted (e.g., due to a network issue or browser refresh), the socket.io library automatically attempts to reconnect to the server.
- This ensures that clients rejoin the real-time communication seamlessly without manual intervention.

3. Frontend Error Handling:

- Error events such as connection loss or failed reconnection are handled using basic console logging for debugging purposes.
- While the application doesn't display user-facing error messages (e.g., "Connection lost"), these errors are monitored in the browser developer tools for troubleshooting.

4. Backend Logging:

- The backend logs disconnection events, including the socket.id of disconnected clients, to track and monitor connection statuses.

5. Room for Improvement:

- Currently, no visual indicators or notifications are implemented for users experiencing connection issues. However, this could be a potential enhancement to improve user experience.

Step 5: Scalability and Performance Optimization

The WebSocket implementation in the ticketing system is designed to handle multiple connections and real-time updates efficiently. While it is a lightweight setup, it incorporates several elements that contribute to scalability and performance:

1. Efficient Use of socket.io:

- The socket.io library supports real-time communication for multiple users without compromising performance.
- Its capability to manage event-driven architecture ensures the system is lightweight and adaptable to growing demands.

2. Backend Resource Management:

- WebSocket connections are dynamically handled by the backend.
- Disconnection cleanup ensures server resources are freed immediately.

3. Horizontal Scaling with Load Balancers:

- The architecture allows the use of multiple WebSocket servers for scalability.
- Load Balancers can ensure users stay connected to the correct server for real0time updates.

4. Integration with Kafka:

- Kafka acts as the backbone for message processing, ensuring large-sclae data streams are managed efficiently.
- The Kafka producer / consumer workflow supports high volumes of real-time events without overloading the system.

5. Database Optimization:

- MongoDB's indexing and schema design improve query performance for real-time updates.
- Optimized write operations ensure tickets are updated and retrieved with minimal delay.

6. Potential Future Enhancements:

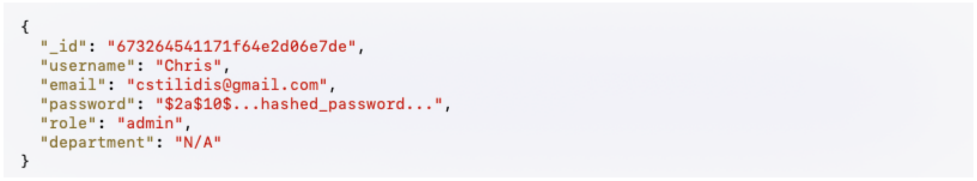
- **Rate Limiting:** Prevent overloading the WebSocket server with excessive updates.
- **Monitoring Tools:** Implement tools to track WebSocket and Kafka performance metrics.

Database Design and Operations

- **Database Schema and Relationships:**

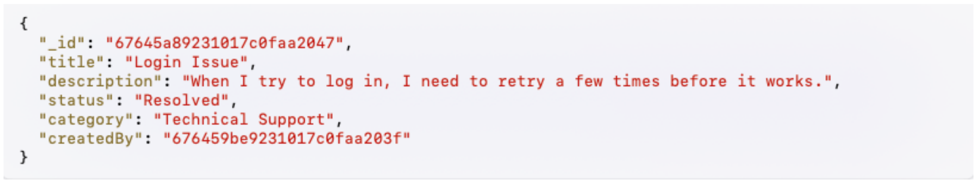
  The Real-Time Ticketing System leverages MongoDB for efficient storage and management of data. The database consists of two main collections: Account and Tickets. These collections are structured to ensure scalability and easy access to information.

- **Accounts Collection:**
    - Fields:
        - _id (ObjectId): A unique identifier for each account.
        - username (String): The display name of the user.
        - email (String): The user’s email address
        - password (String): A hashed version of the user’s password for secure authentication.
        - role (String): Specifies the user’s role (e.g., admin, agent, or user).
        - department (String): Applicable only for agents, specifying their department (e.g., Technical Support, Sales).
    - Example Document:



```
{
  "_id": "673264541171f64e2d06e7de",
  "username": "Chris",
  "email": "cstilidis@gmail.com",
  "password": "$2a$10$...hashed_password...",
  "role": "admin",
  "department": "N/A"
}
```

- **Tickets Collection:**
    - Fields:
        - _id (ObjectId): A unique identifier for each ticket.
        - title (String): A short title describing the ticket.
        - description (String): Detailed information about the issue.
        - status (String): Tracks the ticket’s progress (e.g., Open, Resolved).
        - category (String): Specifies the type of issue e.g., Technical Support, General Inquiry)
        - createdBy (ObjectId): Reference the _id of the user who created the ticket
    - Example Document:



```
{
  "_id": "67645a89231017c0faa2047",
  "title": "Login Issue",
  "description": "When I try to log in, I need to retry a few times before it works.",
  "status": "Resolved",
  "category": "Technical Support",
  "createdBy": "676459be9231017c0faa203f"
}
```

- **Relationships:**

  The createdBy field in the Tickets collection references the _id field in the Accounts collection. This relationship allows the system to associate each ticket with the user who created it, enabling role-based interaction and efficient data querying.

- **Database Operations and Query Optimization**

Efficient database operations are crucial for ensuring the Real-Time Ticketing System performs as expected. Below, the current CRUD operations implemented in the system are described, followed by suggestions for potential enhancements in query optimization.

CRUD Operations

The Ticketing System supports the full range of CRUD (Create, Read, Update, Delete) operations for managing tickets. These operations are defined in the ticketRoute.js file and interact with the MongoDB collections via Mongoose.

1. Create:

- **Endpoint**: POST /tickets
- Inserts a new ticket into the database using the data provided in the request body.
- Field include:
  - title
  - description

    - status
    - category
    - createdBy (a reference to the Account collection)

2. Read:

- **Endpoint**: GET /tickets.
- Retrieves all tickets from the database.

3. Read Ticket by ID:

- **Endpoint:** GET /tickets/:id
- Fetches a specific ticket by its unique identifier.

4. Update Ticket by ID:

- **Endpoint:** PUT /tickets/:id
- Updates the status field of a ticket by its ID.

5. Delete:

- **Endpoint:** DELETE /tickets/:id
- Removes a ticket from the database

Potential Query Optimization Techniques

Currently, the application does not include advanced query optimization techniques like indexing, pagination, or lean queries. The following optimizations are suggested for future implementation to improve database efficiency:

1. Indexing:

- Create custom indexes for frequently queried fields, such as status and createdBy, to speed up query performance.

2. Pagination:

- Implement pagination using limit and skip parameters for the GET /tickets endpoint to handle large datasets efficiently.

3. Lean Queries:

- Use Mongoose's .lean() method for read operations to return plain JavaScript objects instead of full Mongoose documents, reducing memory overhead.

4. Cashing:

- Add caching mechanisms for repetitive queries, such as retrieving all tickets, to reduce database load.

5. Batch Updates:

- Optimize write operations by implementing batch updates for scenarios where multiple tickets need updates simultaneously.

System Security

- **JWT Authentication**

  The Real-Time Ticketing System implements JSON Web Tokens (JWT) to secure user authentication and control access to the system's features. This approach ensures that sensitive data, such as user credentials, remains protected while facilitating seamless communication between the frontend and backend.

Token Generation

- JWTs are generated during the login process within the authController.js file.

- When a user logs in, the system creates a token by combining the user's email, account ID, and randomly generated JWT secret stored in the /env file.
- The token is set to expire after one hour, ensuring that expired tokens cannot be reused.

Access Control

- The token is sent to the client after successfully authentication and is included in subsequent requests as part of the headers.
- The system verifies the token's validity for every incoming request to protect against unauthorized access.
- Role-based access control (RBAC) is enforced at the route level. Only users with appropriate roles can access specific dashboards (e.g., Admin, User, or Agent dashboards).

Implementation Highlights

- **Randomized Security**: The use of a unique secret key for each session ensures tokens are unpredictable and secure.
- **Expiration Mechanism:** Token expiration minimizes the risk of misuse in case of credential compromise.
- **Role Verification:** Routes include middleware checks to verify both the token and user role, ensuring that access permissions are strictly enforced.

- **Password Hashing**

  Password security is a critical component of the real-time ticketing system, ensuring user data is protected from unauthorized access. The application employs bcrypt, a widely-used library for hashing passwords.

Hashing Implementation

1. During User Registration

- In the authController.js file, when a user registers, the registerAccount function receives the raw password from the request body along with other user details.
- A new instance of the Account schema is created using these details, but the password is not yet hashed at this stage.

2. Pre-Save Middleware:

- The accountModel.js file contains a Mongoose schema for accounts.
- Before saving the account to the database, the pre-save middleware (accountSchema.pre("save")) is triggered. This middleware:
    - Generate a unique salt script bcrypt.genSalt().
    - Hashes the password using bcrypt.hash() by combining the raw password with the generated salt.
- The hashed password replaces the original raw password in the scehma before it is stored in the database.

Security Benefits

- **Protection Against Brute Force Attacks:** By salting passwords, the application ensures that even if two users have the same password, their hashed values will differ.
- **Enhanced Security:** The use of bcrypt ensures the system is resilient to modern cracking techniques, as it automatically handles complexities like salting and has strengthening.

This Implementation ensures that user passwords are securely stored, mitigating risks of credential leakage.

- **Role-Based Access Control (RBAC)**

  Role-Based Access Control (RBAC) is implemented in the Real-Time Ticketing System to manage user access and define permissions based on their roles. The application currently supports three distinct roles: **User**, **Agent**, and **Admin**, each with specific access rights. This system ensures that users can only access resources and perform actions permitted by their roles.

Implementation Details

1. **Role Assignment:**
   - Roles are assigned during account creation. By default:
     - Users are assigned to the role user.
     - Admins and agents must be manually configured during or after account creation.
2. **Middleware Validation:**
   - The authMiddleware.js file includes the authenticateJWT middleware, which:
     - Validates the incoming JWT token to ensure the user is authenticated.
     - Extracts the user's details (e.g. role, id) and attaches them to req.user for subsequent handling.
   - This middleware is applied selectively in the ticketRoute.js file:
     - GET / **(Get All Tickets):** Middleware ensures authenticated access and retrieves tickets based on the user's role.

    - POST / **(Create Ticket):** Validates that the user is authenticated before creating a new ticket.
    - PUT /:id **(Update Ticket):** Ensures the requester is authenticated before updating a ticket.

3. Endpoint Behavior Based on Roles:

- The role field, extracted from req.user, determines how each role interacts with endpoints:
    - **User**: Can retrieve tickets they created (createdBy).
    - **Agent**: Accesses tickets within their assigned department (Category).
    - **Admin**: Can retrieve all tickets, with details of the creator populated for additional context.

4. Inconsistent Middleware Application:

- The authenticateJWT middleware is not currently applied to
    - GET /id **(Get Ticket by ID):** This endpoint retrieves ticket details based on the provided ID but does not validate the requester.
    - DELETE /:id **(Delete Ticket):** Admins can delete tickets without middleware validation.
- These cases may require review to ensure consistent access control.

5. Role-Based Routing in the Frontend:

- The App.js file ensures users are redirected to the appropriate dashboard after login:
    - User Dashboard: Displays user-specific tickets and actions
    - Agent Dashboard: Displays department-specific tickets.
    - Admin Dashboard: Provides access to all tickets and administrative actions

- **Testing and Validation**

Manual Testing Procedures

During the development process, the application was thoroughly tested using manual testing methods to ensure the correctness of each feature and endpoint. The following steps were taken:

1. Postman API Testing:

Before integrating the front-end interface, Postman was used to test all backend endpoints, including:

- POST /tickets: Validating ticket creation with correct and incorrect data inputs
- GET /tickets: Ensuring tickets are retrieved based on user roles and filters
- PUT /tickets/:id: Verifying status updates for tickets.
- DELETE /tickets/:id Confirming the successful deletion of tickets

2. End-to-End Testing After Front-End Integration:

Once the React-based front-end was connected to the backend, the application was tested by performing every possible user action. This included:

- User Role Testing: Logging in as a user, agent, and admin to validate access control and role-base interactions.
- WebSocket Event Validation: Testing real-time updates, including ticket creation, status updates, and deletion, across multiple dashboards (User, Agent, Admin) simultaneously.
- Edge Cases: Checking scenarios like invalid credentials, unauthorized actions, and sending incomplete form data.

Observation and Results

- Endpoints Validation: All endpoints responded as expected, with appropriate error messages and data handling for incorrect requests.
- WebSocket Reliability: Real-time updates worked across connected clients after refreshing the initial dashboard connection.
- User Experience: The interface behaved consistently with expected workflows, with no major functional issues observed.

Potential Enhancements for Testing

- **Automated Testing:** Tools like Jest and Mocha could be introduced for automated testing of both the front-end and backend.
- **Load Testing:** Simulating high traffic using like Apache JMeter to evaluate performance under stress.
- **Integration Testing:** Verifying interactions between different modules, such as Kafka and MongoDB, in a continuous integration environment.

3.4.Chapter Conclusion

This chapter detailed the design, architecture, and workflows of the Real-Time Ticketing System, emphasizing the modularity, scalability, and reliability of its compnents. The key points covered include:

1. System Design and Architecture:

- A high-level overview of the system's components, including the role of WebSockets, Kafka, and MongoDB, in ensuring real-time communication and efficient data management.

2. Detailed Workflow and Interactions:

- An explanation of user, agent, and admin workflows, supported by illustrative diagram showcasing the ticket creation, assignment, and management processes.

3. Database Design and Operations:

- The schema relationships between the Account and Tickets collection in MongoDB.
- CRUD operations and potential query optimization techniques to enhance system performance.

4. System Security:

- Implementation of robust authentication, password hashing, and role-based access control mechanisms.

5. Testing and Validation:

- Thorough manual testing of all system features using Postman and React dashboards to validate endpoints and real-time updates.

The chapter provided a comprehensive breakdown of the system's core functionalities, security, and operational aspects. This foundation sets the stage for the next chapter, which will focus on deployment strategies, system evaluation, and potential future improvements.

4. System Implementation and Development

4.1. Overview of the Development Process

The development of the Real-Time Ticketing System was guided by the need for a robust, scalable, and user-friendly application. The system leverages modern web technologies to provide real-time functionality, efficient data handling, and secure access control for users, agents, and administrators.

The system was developed in **four main phases**:

1. **Backend Setup**: Establishing the foundational backend with Node.js, Express, and MongoDB for API development and database integration.
2. **Frontend Integration**: Building the user interface using React, integrating role-based routing, and connecting the frontend to backend APIs.
3. **Real-Time Communication**: Implementing WebSocket (socket.io) for instant updates across dashboards and ensuring seamless synchronization.
4. **Testing and Validation**: Manual testing was performed to validate all functionalities and identify bugs during the development process.

The following technologies played a key role in the development process:

- **Backend**: Node.js, Express, MongoDB, Mongoose, socket.io, Kafka (message streaming), bcrypt (security), and JSON Web Token (JWT) for authentication.
- **Frontend**: React, React Router, and socket.io-client for real-time interactions.
- **Development Tools:** Postman (API Testing), Visual Studio Code (IDE), and Docker for containerization.

The next sections provide a deep dive into the implementation details of each component, starting with the backend.

4.2.Backend Implementation

4.2.1. Setting Up the Backend

The backend environment was initialized using Node.js, a lightweight and efficient runtime environment for building server-side applications. The following key dependencies were installed to enable core functionalities:

- **Express.js:** A web application framework for building API endpoints and routing
- **Mongoose:** A library for MongoDB object modeling to manage schemas and database operations.
- **socket.io:** For enabling WebSocket communication to achieve real-time updates.
- **Kafka-node:** To facilitate message streaming between producer and consumer in the Kafka ecosystem.
- **bcrypt:** To securely hash passwords for user authentication.
- **Jason Web Token (JWT):** For implementing secure user authentication and session management.

The project directory was organized using a modular architecture to improve maintainability and scalability. Major folders included:

- Controllers: Handlers for route-specific logic, such as authController.js and ticketController.js.
- Routes: Definitions of API endpoints like authRoute.js and ticketRoute.js.
- Models: MongoDB schema definitions for Account and Ticket.
- Middleware: Functions for authentication and error handling, such as authMiddleware.js.
- Services: Additional functionalities like kafkaProducer.js, kafkaConsumer.js, and socket.js.

4.2.2. Implementing API Endpoints

The backend API endpoints were developed to support CRUD operations for managing tickets and user accounts. The following endpoints were implemented:

1. **Authentication Routes** (authRoute.js):
   - **POST /auth/register**: Allows new users to register by creating an account
   - **POST /auth/login**: Authenticates users and generates a JWT for secure session handling.
2. **Ticket Routes** (ticketRoute.js):
   - **POST /tickets/create**: Allows users to create a new ticket. The backend validates the user's session and data integrity before saving the ticket to MongoDB.
   - **GET /tickets:** Retrieve a list of tickets. Filters are applied based on the user's role (e.g., admin sees all tickets; agents see category-specific tickets).
   - **GET /ticket/:id**: Retrieves a specific ticket by its ID.

- **PUT /tickets/:id**: Allows user to update ticket status or details. Validation ensures authorized access.
- **DELETE /ticket/:id**: Enables admins to delete tickets.

4.2.3. Database Integration and Models

The backend leverages MongoDB for efficient data storage and management. Using the Mongoose library, it defines schemas to ensure consistency and proper validation.

Account Schema (accountModel.js):

- **Fields**:
  - username: The user’s display name.
  - email: A unique identifier for accounts.
  - password: Securely hashed using bcrypt.
  - role: Access level (e.g., user, agent, admin).
  - department: Optional; used for agents to categorize tickets.
- **Features**:
  - Passwords are hashed using bcrypt saving.
  - Role-based validation ensures the correct role is assigned.

Ticket Schema (ticketModel.js)

- **Fields:**
  - title: A brief description of the issue.
  - description: Detailed information.

    - status: Current state of the ticket (E.g., waiting, resolved).
    - category: The ticket's type of classification (e.g., Technical Support, Sales).
    - createdBy: A reference to the Account model, linking the ticket to its creator.
- **Features**:
    - createdBy is populated to include user details when retrieving tickets.
    - Enum validation ensures status and category have predefined values.
    - Schema validation guarantees that required fields are present and correctly formatted.

4.2.4. Integration with Kafka

The integration of **Apache Kafka** into the Real-Time Ticketing System plays a critical role in ensuring efficient message streaming and event-driven communication. By using Kafka alongside **Docker**, the system achieves high scalability, fault tolerance, and minimal latency.

Kafka facilitates communication between different components of the system through the following key elements:

1. **Producer-Consumer Workflow:**
    - Producers (e.g., backend services) generate and publish events, such as ticket creation, updates, or notifications, to Kafka topics.
    - Consumers (e.g., WebSocket services or other backend services) subscribe to these topics to receive and process the events in real time.

2. **Kafka Usage in Ticket Creation:**
   - Kafka is used specifically for handling ticket creation events in the system.
   - Producers in the backend emit ticket creation events to dedicated topic, which is processed by consumers responsible for storing and updating ticket related data.
   - Thes decouples the ticket creation process for other system components, ensuring smooth processing even under high loads.
3. **Zookeeper:**
   - Zookeeper facilitates the management of Kafka, ensuring its proper operation.
   - It tracks the Kafka broker and maintains synchronization across components.
   - Even with a single broker, Zookeeper ensures configurations are consistent and operations are reliable.
4. **Dockerized Setup:**
   - Kafka and Zookeeper are containerized using **Docker** to simplify deployment and ensure portability.
   - The containerized setup guarantees that Kafka runs consistently across different environments, avoiding dependency conflicts.
   - This setup is particularly useful if the application needs to scale or be deployed to a cloud service in the future.
5. **Benefits of Kafka Integration in System:**
   - **Real-Time Ticket Handling**: Kafka ensures ticket creation events are processed efficiently.

- **System Reliability**: Decoupling components via Kafka improves the reliability and scalability of the ticketing system.
- **Simplified Deployment:** Dockerized Kafka integration facilitates smooth deployment without dependency issues.

4.2.5. Real -Time Updates with WebSocket Integration

Real-time updates enable instantaneous communication between clients and servers, ensuring seamless synchronization of data across multiple users. By utilizing WebSockets, a persistent connection is maintained, allowing the server to send updates to clients as events occur. This approach eliminates the need to repeated requests, providing a more efficient and user-friendly experience.

WebSocket Server Setup

The WebSocket server is configured to handle bi-directional communication between the server and clients. It is initialized during the backend setup and listens for events triggered by changes in the system. Clients connect to the WebSocket server and subscribe to relevant updates, ensuring the receive notifications in real time.

Event-Driven Communication

WebSockets adopt an event-driven approach, where updates are sent immediately to subscribed clients. These updates include data changes, status modifications, which are broadcasted as soon as they are triggered, ensuring minimal latency in data delivery.

Connection Management

Managing connections efficiently is crucial for maintaining a reliable system. The WebSocket server supports multiple simultaneous connections, provides mechanisms for connection in case of distributions, and ensures that user sessions remain secure and active throughout their interactions.

Benefits of WebSocket Integration

- **Immediate Updates**: Ensures that users receive data changes as they occur, enhancing responsiveness.
- **Reduced Network Load**: Eliminates the need for frequent HTTP request by using persistent connections.
- **Improved Scalability**: Handles multiple connections concurrently, enabling smooth operation for many users.

4.2.6. Backend Error Handling and Logging

Error handling and logging are essential for maintaining a robust backend system. Proper error management ensures the application can gracefully handle unexpected issues, while logging provides valuable insights into system performance and use interactions. Together, these components enhance reliability, simplify debugging, and contribute to a seamless user experience.

Error Handling Strategies

- **Middleware-Based Error Handling**: Centralized middleware is used to catch errors from different routes and return appropriate HTTP responses to the client.
    - Example: 500 Internal Server Error for unexpected issues or 400 Bad Request for validation errors.
- **Validation Errors:** Before processing requests, validation middleware ensures the incoming data meets the required criteria, reducing the likelihood of application-level errors.
- **Graceful Degradation:** If a specific feature fails, fallback mechanisms ensure that the system continue to function without crashing.

Benefits of Proper Error Handling and Logging

- **Improved Debugging**: Logs provide developers with detailed insights to quickly identify and resolve issues.
- **Enhanced Security:** By logging unauthorized access attempts or suspicious behavior, the application can detect and respond to potential threats.
- **User Experience:** Graceful error handling ensures minimal disruption to users, maintaining confidence in the application.

4.2.7. Security Features in Backend

General Description:

Security in backend development is critical to protect sensitive user data, ensure the integrity of the system, and maintain user trust. This section explores the mechanisms implemented to secure the application's backend, including authentication, encryption, and role-based access control.

Authentication and Authorization

- **JWT Authentication:**
  - The system uses JSON Web Tokens (JWTs) for user authentication.
  - JWTs are issued upon login and are required for accessing protected routes.
  - Tokens include encoded user information and are validated with a secret key.
- **Role-Based Access Control (RBAC):**
  - Different roles (e.g., user, agent, admin) determine access to various features.
  - Middleware verifies user roles before granting access to specific routes or dashboards.

Password Security

- **Hashing with Bcrypt:**
  - User passwords are securely hashed using bcrypt before being stored in the database.
  - Salts are used to enhance password security and defend against brute-force attacks.

Role-Based Access Control (RBAC)

- **Implementation in Middleware:**
    - Middleware checks the role property stored in the token.
    - Specific endpoints validate whether a suer has sufficient privileges (user, agent, or admin) before granting access.
    - Ensures that only authorized users can perform sensitive actions like ticket deletion (restricted to admins).

Token Validation

- **JWT Verification**:
    - Tokens are validated on each request to protected routes using middleware.
    - Ensures the token is intact, unexpired, and matches the server's secret key.
    - Decoded information from the token (e.g., id, role, email) s attached to the req.user object for downstream use.

Error Message Handling

- **Custom Error Messages:**
    - Provides meaningful yet secure error messages for invalid tokens (e.g., "Unauthorized access") without revealing implementation details.
    - Ensures consistent communication with the client.

4.3.Front-End Implementation

4.3.1. Overview of the Front-End Development

The front-end of the Real-Time Ticketing System was developed using React, a moder JavaScript library known for its efficiency and flexibility in building user interfaces. The front-end servers as the primary interface for users, agents, and administrators, enabling seamless interaction with the backend APIs and real-time communication through WebSocket integration.

Key Technologies and tools used in the front-end development include:

- **React**: For building component-based user interfaces.
- **React Router:** To handle role-based navigation and routing
- **Axios**: For making HTTP requests to the backend APIs
- **socket.io-client**: To enable real-time udpates and event-driven communication between the server and clients
- **CSS**: For styling and maintaining a clean and user-friendly interface.

The front-end is designed with a modular structure to ensure scalability and maintainability, focusing on role-based access to dashboards and real-time updates. The system accommodates three distinct roles: **User**, **Agent**, and **Admin**, with tailored functionalities for each.

4.3.2. User Dashboard Implementation

The User Dashboard provides a simple and intuitive interface for users to create tickets, monitor their progress, and receive real-time updates.

Key Features:

1. **Ticket Creation:**
   - Users can create tickets by filling out a form with fields for title, description and category.
   - The form data is validated on the client side to ensure required fields are completed.
   - Upon submission, **an API call** is made to the /create-ticket endpoint with the user's session token included for authentication.

2. **Viewing Tickets:**
   - The dashboard displays a list of tickets created by the logged-in user.
   - The system uses the /get-tickets endpoint to fetch the tickets associated with the user's account.
   - Real-time updates are handled through WebSocket events such as ticket-created and status-updated, ensuring the ticket list reflects the latest changes without requiring a page refresh.

3. **Real-Time Notifications**
   - The socket.io-client library listens for relevant events, updating the UI dynamically as tickets are created or their statuses change
   - For example, when an agent updates a ticket’s status, the user immediately sees the updated status on their dashboard.

4.3.3. Agent Dashboard Implementation

The **Agent Dashboard** is tailored to allow agents to manage tickets assigned to their categories and update their statuses.

Key Features:

1. **Assigned Tickets:**
   - Agents can view tickets assigned to their department (e.g., Technical Support, General Inquiry).
   - The system uses the /get-tickets endpoint with role-based filters to ensure only relevant tickets are displayed.
2. **Updating Ticket Status:**
   - Agents can change the status of tickets (e.g., from “waiting” to “in-progress” or “resolved”).
   - Updates are made via the /update-ticket endpoint, with role-based validation ensuring only authorized agents can perform this action.

- Once updated, the status change is broadcast to all relevant dashboards through the status-updated WebSocket event.

3. **Real-Time Updates:**
   - The dashboard dynamically listens for updates on tickets within the agent's category using WebSocket events such as ticket-created and status-updated.

4.3.4. Admin Dashboard Implementation

The **Admin Dashboard** provides complete oversight and control of the system, allowing administrators to monitor, create, update, and delete tickets.

Key Features

1. **View All Tickets:**
   - The /get-tickets endpoint fetches all tickets in the system for display on the Admin Dashboard, ensuring administrators have a comprehensive view.
2. **Creating Tickets:**
   - Similar to user functionality, admins can create tickets through a dedicated form on the dashboard.
   - Real-time updates ensure that newly created tickets appear immediately on all relevant dashboards.
3. **Updating and Deleting Tickets:**

- Admins can update ticket details or delete obsolete tickets using the /update-ticket and /delete-ticket endpoints.
- Deletions trigger the ticket-deleted WebSocket event, ensuring all dashboards reflect the removal in real time.

4.3.5. Admin Dashboard Implementation

Real-time updates are central to the system's front-end functionality, ensuring that users, agents, and admins receive instantaneous feedback on ticket activities.

Key Components:

1. **WebSocket Integration:**
   - The socket.io-client library establishes persistent connections with the backend WebSocket server.
   - Event listeners (socket.on) are implemented to handle updates such as ticket-created, status-updated, and ticket-deleted.
2. **Dynamic UI Updates:**
   - Each dashboard dynamically updates its UI based on received WebSocket events. For example:
     - The User Dashboard reflects status changes to tickets.
     - The Agent Dashboard updates the list of assigned tickets.
     - The Admin Dashboard synchronizes changes across all tickets.
3. **Error Handling and Reconnection:**
   - The front-end includes basic error handling to manage disconnections or failed updates.

- The socket.io-client library ensures automatic reconnection in case of network interruptions, maintaining real-time functionality.

4.3.6. Front-End Routing and Role-Based Navigation

Routing and role-based navigation are implemented using React Router to ensure users are directed to the appropriate dashboard based on their role.

Key Features:

1. **Dynamic Routing:**
   - Upon login, the user's role is extracted from the session token, and they are redirected to the corresponding dashboard (User, Agent, Admin)
2. **Protected Routes:**
   - Routes are secured using authentication middleware on the front end, ensuring only logged-in users can access the system.
   - Unauthorized users are redirected to the login page.
3. **Role-Based Access:**
   - Navigation links and options are dynamically rendered based on the user's role, ensuring a streamlined and secure user experience.

4.4.Limitations and Challenges

4.4.1. Overview

Developing a real-time ticketing system using modern technologies like Kafka, WebSocket, and React presented a set of challenges. These ranged from understanding the nuances of each technology to integrating them cohesively to meet system requirements. This section highlights the key challenges encountered during the development process and the current limitations of the system, providing a foundation for future improvements.

4.4.2. Challenges Encountered:

1. Setting up Kafka with Docker:

   Kafka integration using Docker was initially challenging, particularly in configuring the broker, Zookeeper, and ensuring smooth communication between producers and consumers.

   - Debugging involved analyzing logs both in the Docker containers and backend console to identify connectivity or configuration issues.
   - A key learning point was ensuring proper Kafka topic creation and maintaining consistency in broker configurations to avoid message delivery failures.

2. **WebSocket Communication:**

   - Establishing real-time communication via WebSockets required careful attention to backend and frontend integration. Configuring event-based communication, such as broadcasting ticket-created or status-updated events, was complex in the early stages.

- Challenges included managing socket disconnections, reconnections, and ensuring all clients (User, Agent, and Admin dashboards) received consistent updates.
- Testing real-time events across multiple clients in parallel proved time-consuming, as each dashboard had to respond accurately without introducing latency.

3. **Role-Based Routing and Navigation:**
    - Implementing secure role-based routing on the frontend required precise handling of session tokens and user roles. Misconfigurations during early development resulted in users being redirected to incorrect dashboards or accessing unauthorized features.
    - Debugging these issues involved testing scenarios for all roles (User, Agent, and Admin) and ensuring that access control logic aligned with backend validations.
4. **System Synchronization:**
    - Maintaining synchronization between the frontend, backend, and databases was critical for ensuring that ticket updates appeared in real time. Minor timing mismatches between WebSocket events and database writes initially led to inconsistent states, such as tickets appearing with outdated statuses.
    - These issues were resolved through careful sequencing of WebSocket emissions after database transactions were confirmed.
5. **Log Analysis for Debugging:**
    - Debugging backend and frontend errors required consistent use of logs. Backend errors related to Kafka topic consumption and WebSocket broadcasting often lacked clear indications of root

causes, requiring trial-and-error fixes based on console logs and container outputs.

4.4.3. Current System Limitations

1. **Local Deployment Only:**
   - The application currently operates in a local development environment. While this suffices for testing and demonstration, it limits scalability and accessibility for broader use cases. Deploying the system to a cloud environment (e.g., AWS, Google Cloud) would provide better scalability and accessibility.
2. **Manual Testing:**
   - While manual testing validated the system's functionality, automated testing (e.g., Jest for React or Mocha for backend) was not implemented. This limits the ability to test the system efficiently under different scenarios or with high concurrency.
3. **WebSocket Scalability:**
   - The WebSocket implementation, while functional, is not yet optimized for high scalability. As the number of concurrent connections increases, performance bottlenecks may emerge. A load-balanced architecture or clustering WebSocket servers could address this in the future.
4. **Basic Error Handling:**

- Error handling in the current system is basic, focusing primarily on logging and retry mechanisms. For example, disconnection errors in WebSocket clients are logged but not conveyed to the user through UI notifications. Enhanced error reporting could improve usability.

5. **Limited Kafka Usage:**
    - While Kafka is integrated as the messaging backbone, its advanced features like partitioning, multi-topic consumers, and replication were not fully utilized. These features could improve scalability and reliability for larger datasets or more complex workflows.
6. **Database Query Optimization:**
    - MongoDB queries are functional but not yet optimized for performance at scale. Techniques like indexing, pagination, or caching could enhance the system's efficiency when handling large datasets.
7. **No External Monitoring Tools:**
    - The system lacks monitoring tools to track performance metrics for Kafka, WebSocket, or MongoDB. Integration tools like Prometheus or Grafana could provide valuable insights for future scaling and troubleshooting.

4.4.4. Lessons Learned

The development process underscored the importance of:

- Thorough documentation for each component to streamline debugging.

- Incremental testing to isolate and resolve issues early in the development lifecycle.
- Leveraging logs effectively to pinpoint issues in complex setups like Kafka and WebSockets.

4.4.5. Future Enhancements

Addressing the identified challenges and limitations would involve:

- Deploying the system on a cloud platform to enable wider accessibility and real-world testing.
- Introducing automated testing frameworks to reduce manual effort and improve test coverage.
- Optimizing WebSocket and Kafka setups for higher scalability
- Enhancing error handling mechanisms to provide better user feedback and system resilience.

5. Results and Evaluation

5.1. Overview

This chapter evaluates the developed Real-Time Ticketing System, focusing on its performance, functionality, and ability to meet the objectives outlined in Chapter 1. The evaluation assesses the system's core features, real-time capabilities, and user experience across different roles (User, Agent, Admin). It also reflects on challenges encountered during testing and discusses the system's limitations.

5.2.Functional Validation

The Real-Time Ticketing System was rigorously tested to validate its core functionalities. The manual testing process ensured that the application fulfilled its intended objectives.

1. **Ticket Creation and Management:**
    - Users successfully created tickets through the User Dashboard, with data validated on both the frontend and backend before being stored in the database.
    - Tickets were accurately displayed in the User, Agent, and Admin Dashboards, ensuring seamless role-based access control.
    - Real-time WebSocket updates ensured that all stakeholders were instantly notified of new tickets and status changes.
2. **Role-Based Access Control:**
    - Users could only access tickets they created, maintaining data confidentiality.
    - Agents had access to tockets assigned to their respective categories and could update ticket statuses.
    - Admins had full access to create, update, and delete tickets while overseeing all system activities.
3. **Real-Time Updates:**
    - WebSocket integration successfully enabled instant updates across dashboard without requiring page refreshes.

- Key events, such as ticket creation, updates, and deletion, were broadcast and synchronized across all connected clients.

4. **Error Handling:**
    - Invalid data inputs were handled gracefully with meaningful error messages displayed to users.
    - Unauthorized actions were blocked, ensuring system security and integrity.

5.3.Real-Time Performance

The system's real-time performance was evaluated on tis responsiveness and ability to handle simultaneous updates across multiple dashboards:

1. **WebSocket Communication:**
    - The WebSocket server maintained persistent connections with multiple clients, ensuring low-latency communication.
    - Updates (e.g., ticket creation, status changes) were consistently delivered to connected clients within milliseconds, meeting the real-time requirements.
2. **Kafka Integration:**
    - Kafka's producer-consumer model ensured reliable message delivery for ticket events.
    - While the system operated locally, Kafka's ability to handle large-scale data streams demonstrated its potential for scalability.
3. **Scalability Observations**
    - Although deployed locally, the system's modular architecture, Kafka integration, and WebSocket implementation suggested its ability to scale for larger user bases in distributed environments.

- Potential enhancements, such as adding load balancers and multi-broker Kafka configurations, could further optimize scalability

5.4.Challenges in Testing

During the development and testing phases, several challenges were encountered:

1. **WebSocket Integration:**
   - Initial issues with establishing stable WebSocket connections were resolved by debugging server and frontend configurations
   - Logs in both backend and frontend helped identify missing event listeners and misconfigured socket connections.
2. **Kafka Setup:**
   - Setting up Kafka locally with Docker presented challenges in configuring Zookeeper and managing broker connections.
   - Careful review of logs and iterative debugging ensured successful integration of Kafka with the backend.
3. **Frontend Synchronization:**
   1. Minor delays in reflecting updates on dashboards were resolved by optimizing event listeners and database queries.
4. **Testing Multi-User Scenarios:**
   2. Testing simultaneous actions (e.g., multiple agents updating tickets) required careful observation of real-time updates to ensure data consistency and synchronization.

5.5.Feedback

The system received positive feedback for its functionality and responsiveness:

1. **Self-Evaluation:**
    - Developing the Real-Time Ticketing System was an insightful learning experience, particularly in integrating WebSocket and Kafka for real-time communication. Overcoming challenges, such as debugging WebSocket connections and configuring Kafka with Docker, significantly enhanced both technical and problem-solving skills.
    - The system's modular architecture and secure role-based access control were areas of strength, providing a foundation for future scalability and enhancements.
    - While the application meets its intended objectives, further refinement in areas such as multi-broker Kafka setups and UI optimizations could enhance its scalability and user experience
2. **Peer Review**
    - The application was tested locally by colleagues who interact with the User, Agent, and Admin dashboards. They noted the system's responsiveness and ease of use, particularly its ability to deliver real-time updates without refreshing the interface.
    - Suggestions included refining the UI for better accessibility.

5.6.Summary

This chapter evaluated the Real-Time Ticketing System based on its functionality, real-time capabilities, and user experience. The system met its core objectives by enabling ticket creation, real-time updates, and secure role-based access control. While challenges were encountered during development and testing, they were successfully addressed, ensuring a robust scalable application. These results provide a strong foundation for future improvements and scalability.

6. Conclusions and Future Work

6.1. Summary of Key Contributions

The Real-Time Ticketing System presented in this dissertation demonstrates the practical application of modern technologies, such as WebSockets, Kafka, and an event-driven architecture, to address challenges in real-time communication and ticket management systems. The project's key contribution include:

1. **Real-Time Updates and Synchronization:**
    - The integration of WebSockets enabled instantaneous updates across user, agent, and admin dashboards, ensuring seamless synchronization of ticket-related activities.
2. **Event-Drive Architecture for Scalability:**
    - By leveraging Kafka for event streaming, the sytem introduced a scalable mechanism for reliable message queuing and asynchronous processing of ticket events.
3. **Role-Based Access Control (RBAC):**
    - A robust RBAC implementation ensured secure and differentiated access to system features, catering to the specific needs of users, agents, and administrators.
4. **Modular System Design:**

- The system's modular architecture facilitated streamlined development, efficient debugging, and ease of future enhancements.

5. **User-Centric Experience:**
    - Dashboards tailored to each role, combined with intuitive interfaces and real-time feedback, enhanced user experience and operational efficiency.

6.2.Limitations

While the Real-Time Ticketing System successfully meets its primary objectives, several limitations were identified during development and testing. These limitations provide opportunities for further refinement and enhancement:

1. **Local Deployment Only:**
    - The system implemented and tested in a local environment, which limited its ability to stimulate real-world scenarios with distributed components. Deployment on a cloud platform would provide better insights into scalability and real-time performance under higher loads.
2. **WebSocket Scalability:**
    - Although WebSocket communication was effective in the local setup, the current implementation may encounter bottlenecks with a high number of concurrent connections. A load-balanced architecture or clustered WebSocket servers could address this challenge.
3. **Limited Use of Kafka's Advanced Features:**
    - Kafka's basic producer-consumer model was utilized for event streaming, but its advanced capabilities, such as partitioning, multi-topic consumers, and replication, were not implemented. These features could improve fault tolerance and performance under heavy traffic.

4. **Basic Error Handling and Logging:**
    - The system's error handling was limited to basic logging and retries. Advanced mechanisms, such as user-facing error messages and automated error recovery, were not implemented, which could impact user experience in case of failures.
5. **Manual Testing Over Automated Testing:**
    - While manual testing validated the system's functionality, it lacked the rigor of automated testing frameworks, such as Jest for frontend and Mocha for backend. Automated testing would ensure comprehensive validation and reduce the risk of undetected errors.
6. **Database Query Optimization:**
    - MongoDB queries were functional but not optimized of high-performance scenarios. Techniques like indexing, caching, and pagination could significantly improve query efficiency for larger datasets.
7. **No Monitoring or Analytics Tools:**
    - The absence of monitoring tools, such as Prometheus or Grafana, limited the ability to track system performance, identify bottlenecks, or analyze user interactions.

6.3.Future Work

The Real-Time Ticketing System provides a robust foundation for real-time communication and role-based ticket management. However, there are numerous opportunities to enhance the system's functionality, scalability, and usability. The following areas highlight potential future work:

1. **Deployment to a Cloud Environment:**

- Transition the application to a cloud platform, such as AWS, Google Cloud, or Microsoft Azure, would enable real-world testing and enhance scalability.
- Features like auto-scaling, serverless architecture, and distributed databases could be leveraged to handle larger user bases and high-volume traffic efficiently.

2. **Enhanced WebSocket Scalability:**
   - Implementing load balancing or clustering for WebSocket servers would improve the system's ability to manage a high number of concurrent connections.
   - Features iterations could integrate WebSocket protocol upgrades or alternative real-time solutions, such as Server-Sent Events (SSE) or GraphQL Subscriptions, for scalability.
3. **Advanced Kafka Features:**
   - Extending Kafka's usage to include partitioning and replication would enhance fault tolerance and allow for better handling or high-throughput data streams
   - Multi-topic support could facilitate event categorization, enabling more granular communication between system components.
4. **Automated Testing Frameworks:**
   - Introducing tools like Jest for frontend and Mocha for backend testing would streamline validation and debugging process.
   - Automated testing could cover unit, integration, and end-to-end testing, ensuring the system remains stable during updates or scaling.
5. **Integration of Monitoring and Analytics Tools**:
   - Adding tools like Prometheus, Grafana, or Elasticsearch could provide real-time insights into system performance, usage metrics, and potential bottlenecks.
   - These tools would be particularly beneficial for identifying issues in WebSocket communication, Kafka event processing, or database queries.
6. **Enhanced Role-Base Access Control (RBAC)**

- The current RBAC system could be extended to include dynamic role assignment and more granular permissions.
- Future iterations could support temporary roles or access permissions for specific tasks, improving flexibility in collaborative scenarios.

7. **Improved Error Handling and User Notifications:**
    - Implementing user-facing messages and a more sophisticated retry mechanism would enhance usability during network disruptions or backend failures.
    - Visual indicators for connectivity issues (e.g., WebSocket disconnections) could improve the user experience.
8. **Database Optimization:**
    - Introducing indexing, caching, and pagination mechanism would significantly improve database performance, especially as the dataset grows.
    - A move to a distributed or sharded database setup could further enhance scalability.
9. **UI/UX Enhancements:**
    - Refining the user interface with a focus on accessibility and responsiveness would improve usability across different devices and screen sizes.
10. **Integration with External APIs:**
    - The system could be extended to integrate with third-party platforms like email services (for automated ticket updates) or customer relationship management (CRM) tools.
    - The integration would enhance functionality and support additional workflows, such as automated follow-ups or ticket categorized based on external data.
11. **Multilingual Support:**
    - Adding support for multiple languages would make the system more inclusive, particularly for organizations with global team or customers.
12. **Security Enhancements:**

- Strengthening security mechanisms, such as implementing rate-limiting, IP whitelisting, or two-factor authentication (2FA), would protect the system from potential attacks.
- Regular penetration testing and audits could be introduced to identify and address vulnerabilities

13. Scaling Kafka for High Availability:

- Deploying Kafka in a multi-broker setup with failover support would ensure uninterrupted event processing even during hardware or network failures.

14. Real-Time Analytics Dashboard:

- Creating an analytics dashboard for admins would provide real-time insights into ticket statistics, user activity, and system performance, enabling data-driven decision-making.

15. **Mobile Application Development**:

- Developing a mobile version of the application would extend its reach and usability, catering to users who prefer accessing the system on the go

6.4.Concluding Remarks

The development of the Real-Time Ticketing System has been a challenging yet rewarding journey, blending theoretical knowledge with practical implementation to address real-world challenges in real-time communication and data synchronization. The project demonstrates how modern technologies like WebSockets, Kafka, and MongoDB can be effectively integrated to deliver a robust and scalable platform for managing tickets operations.

Throughout the dissertation, significant emphasis was placed on designing a modular and system architecture, implementing real-time capabilities, and ensuring role-based access control. These efforts culminated in a system that achieves its core objectives while leaving room for future enhancements.

The project has been instrumental in developing a deeper understanding of cutting-edge technologies, problem-solving strategies, and system design principles. The challenges encountered during the development process—ranging from debugging WebSocket connections to configuring Kafka—provided invaluable lessons in resilience and adaptability. Moreover, the iterative testing and refinement phases underscored the importance of meticulous planning and validation is software development

While the system meets its intended goals, it also highlights opportunities for further exploration and improvement. The potential future work outlined in this chapter provides a roadmap for extending the system's a capabilities and scalability, ensuring its relevance and adaptability in dynamic organization contexts.

In conclusion, the Real-Time Ticketing System not only serves as a practical solution to ticket management challenges buy also stands as a testament to the power of Innovative thinking and effective application of technology. The insights gained from this project will undoubtedly inform and enrich future endeavors, contributing to the ongoing evolution of real-time systems.

References

[1] **Apache Kafka. (2025).** *Kafka Documentation.* Retrieved January 10, 2025, from https://kafka.apache.org/documentation/.

[2] **Mozilla Developer Network. (n.d.).** *WebSockets API Documentation.* Retrieved January 10, 2025, from https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[3] **National Institute of Standards and Technology (NIST**). (n.d.). *Role-Based Access Control (RBAC).* Retrieved January 10, 2025, from https://csrc.nist.gov/projects/role-based-access-control.

[4] **GeeksforGeeks**. (2025). *Security in Distributed Systems.* Retrieved January 10, 2025, from https://www.geeksforgeeks.org/security-in-distributed-system/.

[5] **GeeksforGeeks**. (2025). *Adaptive Load Balancing System Design.* Retrieved January 10, 2025, from https://www.geeksforgeeks.org/adaptive-load-balancing-system-design/.

[6] **Almeida, A., Brás, S., Sargento, S., & Pinto, F. C. (2023**). Time series big data: a survey on data stream frameworks, analysis and algorithms. *Journal of Big Data*, 10, Article number: 83. Retrieved from https://journalofbigdata.springeropen.com/articles/10.1186/s40537-023-00760-1

[7] **MongoDB.** (n.d.). Real-time Data Analytics On MongoDB Atlas. Retrieved January 10, 2025, from https://www.mongodb.com/en-us/use-cases/analytics/real-time-analytics

[8] **Gorton, I. (2022).** ***Foundations of Scalable Systems: Designing Distributed Architectures.*** O'Reilly Media. Retrieved from https://www.oreilly.com/library/view/foundations-of-scalable/9781098106058/

[9] **Hannousse, A., & Yahiouche, S. (2020).** ***Securing Microservices and Microservice Architectures: A Systematic Mapping Study*.** arXiv preprint arXiv:2003.07262. Retrieved from https://arxiv.org/abs/2003.07262

[10] Stylidis, C. (2025). Real-Time Ticketing System Codebase. Retrieved January 12, 2025, from https://github.com/cstilidisdev/Education/tree/main/IHU/Master/Dissertation/Real%20Time%20Ticketing%20System