Cureus Journal of Computer Science

Open Access Original Article

Novel Architecture for Distributed Travel Data Integration and Service Provision Using Microservices

Received 04/12/2025
Review began 04/20/2025
Review ended 06/26/2025
Published 06/27/2025

DOI: https://doi.org/10.7759/s44389-025-04835-3

Biman Barua [1, 2] , M. Shamim Kaiser [1]

1. Institute of Information Technology, Jahangirnagar University, Dhaka, BGD 2. Computer Science and Engineering, BGMEA University of Fashion and Technology, Dhaka, BGD

Corresponding author: Biman Barua, biman@buft.edu.bd

Abstract

This paper presents a new microservices-based architecture aimed at improving flexibility, scalability, and performance in airline reservation systems. The main goal of this study is to overcome the challenges of real-time data integration and service responsiveness in busy travel settings. The proposed system uses Redis caching, two messaging platforms (Kafka and RabbitMQ), and different databases (MongoDB and PostgreSQL) to optimize data handling. It ensures secure communication with OAuth2 and JSON Web Token protocols. The architecture was assessed using key performance indicators, achieving 99.5% data consistency, data propagation latency under 75 ms, and a system throughput of 1,050 events per second. Redis caching provided a 92% cache hit ratio, which significantly lowered database load and improved response speed. Containerization with Docker and orchestration through Kubernetes enabled horizontal scalability while keeping a low error rate of 0.2%. These results show that the architecture can support secure, scalable, and efficient real-time operations, making it suitable for modern airline reservation systems and other high-demand travel services.

Categories: HPC in Cloud Computing, Software Development, Distributed Systems
Keywords: microservices architecture, airline reservation system, data integration, redis caching, real-time messaging

Introduction

The global travel industry has undergone a dramatic transformation with the rise of digital services and platforms. Travelers today expect seamless, personalized experiences - ranging from multi-modal transportation options to dynamic accommodation and real-time itinerary updates - all of which demand robust and scalable back-end infrastructures. As data sources across airlines, hotel chains, transport providers, and booking platforms grow in diversity and complexity, the need for efficient integration mechanisms has become paramount.

In response to these demands, the travel sector has evolved to deliver both elaborate and simplified travel experiences by leveraging integrated digital platforms. As digital platforms and services continue to grow, there is an increasing need to integrate travel data from diverse sources, such as reservation systems, transportation systems, lodging systems, and more [1]. Such traditional monolithic systems often find it hard to cope with the demand for greater agility, scalability, and, above all, integration of systems in such a highly distributed environment. Microservices architecture, also referred to as modular architecture because of the use of modules that can be attached and fixed away from the electronic central processing unit, has become a savior for these situations by allowing contiguous heterogenic travel data and travel services to be brought in together effectively [2].

Microservices architecture decomposes a system into a set of services that can be deployed independently of each other but communicate with one another over a set of well-defined APIs [3]. This architectural style allows large-scale ecommerce solutions, such as those of travel agencies, to be designed and implemented in a more elegant way, in which the flexibility of each building block is such that it can change without the effects of other building blocks [4]. Encapsulation of smaller services also leads to improved service orientation in large travel systems which allows for the implementation of myriad systems and content in a single travel planning solution.

One of the prominent advantages associated with the employment of microservices rests upon the assertion that APIs and data integration models may be created with ease for different components in a travel ecosystem [5]. These frameworks guarantee that all the systems including flight booking engines, ride-hailing services, and even hotel reservation systems are able to access each other and interact through sharing of data in real time, thus providing the user with a seamless journey experience [6]. In addition, the employment of microservices allows for enhancing the customization of travel services as it enables collecting and processing data simultaneously, which in turn allows a travel service to provide individual recommendations and offers taking into account preferences and travel history.

Research motivation and background

Among the most vital challenges for the travel industry has been the integration of varied sources of travel data and travel services, particularly when conventional monolithic systems are involved. These systems face challenges when it comes to the scaling and agility; hence, the thirst for flexibility and responsiveness continues to grow. In this era of growing digitalization, good time calls for good time experiences augmented by these digital applications.

The motivation behind this research stems from an ever-increasing need for real-time, user-centric travel services that can truly act on changing user preferences and system conditions [7]. In this post-pandemic world, the buzz patterns of travel have really changed. There's more concern for safety, flexibility, and real-time updates. As travel ecosystems are becoming more digitally interconnected, it is crucial for the different services to be integrated well (e.g., flight bookings, ride-hailing, hotel reservations) so that travelling itself remains a coherent experience for the user, from which they derive satisfaction and operational efficiency.

Current systems are mostly siloed, leading to the unpalatable delays, duplicated operations, and very poor interoperability. The challenge, technically and experientially, is that users don't accept anything less than instant gratification and consistency from one platform to another, and a slick transition across services [8]. Microservices present a way forward to solve this problem, in particular by exposing fine-grained APIs, managing distributed data, and scaling individual services independently.

Problem statement

The travel industry has become more fragmented over the years and as a result, the adoption of different source of travel data and services turns out to be more challenging. This is because in a typical monolithic architecture, it would be very difficult to control the many types of data that come from not only the booking engines but also the transportation and accommodation service providers, which leads to delays, inefficiencies and absence of real time coordination. This fragmentation also presents difficulties in developing and providing a holistic and customized travel experience for the users. In addition, the absence of interoperability among these systems also poses a challenge for building all-encompassing travel brokering services. Microservices architecture provides a probable remedy allowing the integration of services in a modular and distributed manner. Nevertheless, there is still little research that specifically address how this architecture can help in the integration of travel data and services, especially the strong use of APIs and data structures. Solving this challenge seems to be very important for making current travel systems more efficient, customized and extensible.

The main goal of this study is to investigate how microservices architecture can assist the blending of different travel data sources, services, personalizing and optimizing the planning of trips. More specifically, the study strives to design, implement and assess APIs and data integration frameworks issues for connecting multiple so called travel components, like reservation systems, transport and lodging services. Covering those issues, the development of this research provides an architecture for modern travel solutions which is both robust and elastic.

Objectives

The present study concentrates on designing APIs and integration schemes that connect various elements of travel such as booking engines, transport and accommodation services. The microservices architecture is particularly stressed in order to achieve an efficient and effective integration that is also modular. This is to ease the flow of data as well as improve the processes involved in customizing travel packages for individuals. By addressing these concerns, the objective of the study is to offer assistance in dealing with the existing challenges of travel data systems which are overly disorganized.

Research questions

To what extent can microservices architecture reduce latency in multi-source travel data integration?

What are the fundamental principles of API design and data integration necessary to connect reservation systems, transport, and accommodation services effectively within microservices architecture?

Literature review

Microservices design and development have become mainstream for the last several years because it is modular and scalable, making it easy to bring together disparate systems. In contrast to the conventional unitary designs, microservices support the functionality of deploying services separately from one another, which makes it possible to manage the very elaborate, heterogeneous, and mosaic data structure of the sector dubbed tourism [9]. The literature indicates that microservices enhanced the flexibility and scalability of systems, facilitating on the fly data exchange with travel services like booking and transport services [10].

The success of airlines within an interconnected world is dependent on how well each mode of transport is designed and developed along with all the associated components [11]. APIs that are well structured help in the reduction of integration problems, hence advancing how highly transactional systems interact with each other [12]. It is observed that RESTful API integration with common aging data representation formats such as JSON and XML has proven to be successful in facilitating real time integration and a much better user experience [13].

These issues are generally accepted, and due to the existent different data formats and protocols used across different platforms, interoperability is still a distant goal [14]. Some researchers have opined that standardized API design considerations and implementation of service orientated architecture based service discovery strategies can mitigate these problems leading to better integration and scaling out of the system [15]. Further work in this area should concentrate on enhancement of these standard designs for better travel data integration.

Materials And Methods

Research design

Background to Architectural Choices

In designing such a proposed architecture, microservice design was adopted since the nature of modern travel ecosystems is fully distributed. These core functions, such as booking management, transport coordination, hotel reservations, or user profiles, are implemented as separate microservices. These services are expected to function separately and communicate via well-defined RESTful APIs to promote loose coupling and independent deployment. In addition to operational flexibility, this modularity lends itself to scaling, fault isolation, and ease of error fixes in an iterative fashion, moving quickly and constantly in a sophisticated and demanding industry like travel. Redis caching was further considered for superfast data retrieval; Kafka and RabbitMQ run side by side for different messaging requirements (streaming and task queues); orchestration is handled by Docker and Kubernetes for scaling horizontally and guaranteeing smoothness under any load condition.

This proposed system architecture follows a hybrid microservices-based approach that best suits the airline reservation system in which the core services of booking, payments, and flight details are developed as separate containers managed by Kubernetes. This modular design allows each service to be scaled, updated, secured, or simply integrated with real-time data while maintaining superb fast performance.

The system thus has come into being leveraging the following key technologies, so that even in the peak load, it will not degrade in performance:

• Redis for deep caching: Intensive read operations from high-frequency sources have been offloaded to Redis cache. The cache Time-to-Live (TTL) was set to 2 minutes (120 seconds) after testing the trade-off between the freshness of seat availability data and system throughput.

• Kafka and RabbitMQ were set up as real-time messaging layer between microservices. Kafka was set to have five partitions with a replication factor of 3, which have been more than enough to ensure the smooth processing of about 1,050 events/sec at peak load without getting in the way of timely delivery of messages.

• MongoDB and PostgreSQL were chosen for distributed and relational data operations, respectively. PostgreSQL took care of ensuring transactional consistency (e.g., payment confirmation), while MongoDB served dynamic content (e.g., seat layout visualization).

• OAuth2 with JWT was chosen for token authentication. An access token had 15 minutes as its expiry duration, which is actually an accepted generic period, giving an adequate trade-off between usability and security in a distributed system.

• Kubernetes HPA (horizontal pod autoscaler) auto-scaled services depending on CPU usage. Upon CPU usage exceeding 70%, the autoscaler increased the service replicas by a figure up to 40% under heavy load.

All components were dockerized and deployed through Kubernetes to achieve horizontal scalability and efficient resource al__cpLocation. The system maintained an average CPU/memory utilization below 75%, even during stress testing, hence showcasing very efficient resource utilization.

Simulated traffic, representative of booking, searching, and paying activities, were used to test this setup. The system guaranteed a latency of less than 100 ms and a 92% cache hit ratio, which affirmed the correctness of the selected parameter values ensuring real-time responsiveness and robustness.

Innovation

The novelty of the proposed method lies in allowing independent scaling and flexibility. Being microservice-based, the system allows a certain service to evolve without interference to other stakeholders. This decoupling also permits faster updates and iterative development, which become critically important in the fast-paced travel industry. The secondary use of real-time data synchronization ensures travel services are made more agile and personalized towards user needs.

At the heart of the architecture, RESTful APIs are used as the primary means of communication between services supported by typical service-oriented architecture encouraging easy interaction between services. REST APIs are built according to the principles of certain practices (like HTTP) and certain languages (such as sending and receiving only JSON or sometimes XML) to facilitate integration between the services. An API gateway service is typically carried out within the system to streamline client and services interactions in the system with the added advantage of enhancing security as well as the speed of service delivery by dealing with issues such as authentication, load balancing, and provisioning services.

In order to be able to scale the system horizontally and to avoid single points of failure, such solutions, in particular, Docker - the technology allowing to all microservices to possess independent deployments running in separated enclosing spaces, are implemented container system. Kubernetes is used as an orchestrator. It automates the scaling and management of containers based on traffic demand. Tests will also be conducted to measure the effectiveness of the architectural design in terms of interoperability and speed of data synchronization and the scalability of the entire system.

In line with an architecture based on microservices, good APIs and integration frameworks must be built for good communication and orchestration between individual services. The following section describes the approach used to design and implement those APIs.

API and framework development

The creation of robust APIs and integration platforms for seamless communication between multiple travel activities such as reservation systems, transportation providers, and lodging platforms is of primary importance. RESTful APIs are used as the medium because they are light, platform-independent, and well suited for heterogeneous systems. Being stateless made them scalable and an ideal fit for distributed microservices. In order to coordinate services and an ever-increasing number of requests from clients more efficiently, an API Gateway is injected and acts as a one-stop shop for clients, simplifying the access and consumption of services at the backend while enforcing authentication, rate limiting, and intelligent request routing. This combination increases the security and reliability of third-party systems interacting with the backend and ensures that client applications describe services in a consistent, agreed upon, and standardized manner.

Also, the API Gateway can be a key player in minimizing security risks that distributed architectures tend to suffer from. Since it centralizes authentication, request validation, and throttling of traffic, it gets common attack vectors like cross-site scripting or XSS, distributed denial-of-service attacks, and injection flaws out of the picture. In other words, it makes sure that only authorized and sanitized requests get in on the microservices themselves: acting almost as gatekeepers against unauthorized access and malicious payloads. In terms of architecture, this approach greatly reduces the attack surface in contrast to exposing each microservice to external clients.

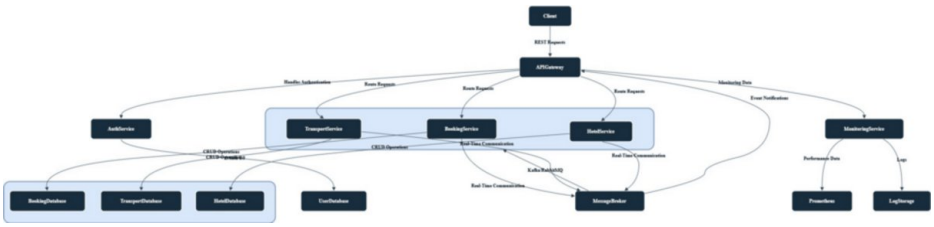

FIGURE 1: API design

API, Application Programming Interface

API Design Principles

The design of the APIs adheres to the industry standards in order to improve reliability and ensure compatibility with other systems. The APIs are implemented over the basic and widely used protocols like HTTP/HTTPS for transport layer and data formatting protocols like JSON and XML for data exchange. This

ensures common access with several client applications and external services. Each API is versioned to enable backward compatibility whereby incorporation of new functions and enhancements do not affect the current services [13]. The API endpoints that the users are supposed to use for the system services are designed in a basic REST manner with clear cut naming structures, e.g., /bookings, /transport, and /hotels, so that different services can be accessed by the developers in a sensible order [14].

The components in Figure *1* are described as follows:

Client: The API receives REST queries from the user.

APIGateway: Takes care of all requests and forwards them to the relevant microservices.

AuthService: Responsible for the authentication implemented in the system using OAuth 2.0. BookingService, TransportService, and HotelService: separate microservices responsible for handling the booking management system, transport system, and the hotel system, respectively.

MessageBroker: Kafka or RabbitMQ enables interaction amongst the services in real-time.

MonitoringService: Analyzes the performance of the system and uploads metrics to Prometheus and the logs to LogStorage.

Databases: Different databases for booking service, transport service, and hotel services' database.

Whereas proper API design establishes interoperability, securing these communication channels is paramount. The next subsection discusses the authentication mechanisms built into the operational system.

Authentication and Security

For the safety of the system, preventive measures such as OAuth 2.0 are incorporated in the APIs. OAuth 2.0 is a mechanism that allows third-party services access user data without the user revealing any credentials [15]. The use of token-based authentication in the APIs serves two purposes: user verification and service permission management, so as to allow only the authorized requests to be carried out as intended. In addition, the use of policies that limit the number of requests per unit of time and reduce the processing speed of requests is also available in order to protect the system against abuse and encourage equitable use by all clients.

OAuth 2.0 integration in APIs: This microservices architecture utilizes OAuth 2.0 authentication management, especially in the case of travel services like booking, ticketing, and accommodation, to ensure secure access to the services. For this reason, many systems corporate travel include controls for access to the system ignoring the user credentials expense. This prevents risk when interacting with numerous services or applications which are external to the system.

FIGURE 2: Authorization and authentication with OAuth 2.0 in the APIs

API, Application Programming Interface

The components in Figure *2* are described as follows:

Client: User logs in and makes API calls.

AuthorizationServer: Is responsible for creating, checking existence, or refreshing of tokens.

AuthService: Checks the credentials against UserDatabase.

APIGateway: Checks the access token and then sends requests to the appropriate microservices.

Microservices: There are services such as bookings, transport services, or hotels that use the API.

TokenStore: Stores the tokens for validation purposes (shown here in form of TokenDB).

Databases: These consist of UserDatabase, which holds user details and their passwords.

Figure *2* depicts the flow of authentication and authorization within the APIs, employing OAuth 2.0, while showing the ways in which the tokens are handled. It has also checked using the the Authorization Server and API Gateway.

Token-based authentication (Figure *3*): OAuth 2.0 is a protocol that defines how to authorize users with a secure method based on tokens. For example, if a mobile app or website client wants to book a flight or check for the availability of hotels on an integrated travel service API [16], the client first has to be redirected to an Authorization Server where the user will be asked to log in. After successful log in, the Authorization Server creates an access token, which it provides to the client. This token is valid only for the purpose of making calls to the API where the client need not present the user's credentials every time to make that API call.

The access token is then added to the subsequent API calls such as the one below where the access token is placed in the authorization header of the request (i.e., Authorization: Bearer <access_token>), hence allowing the client to access travel services as prescribed within the user's authorizing level. For instance, where a user has granted permissions to utilize both the booking and transport APIs, the user will be able to use the same token across the two APIs. The token is assigned a scope and permissions, which is already set in advance to avoid the situation where unauthorized API calls can access resources which are not meant for use by that API call [17].

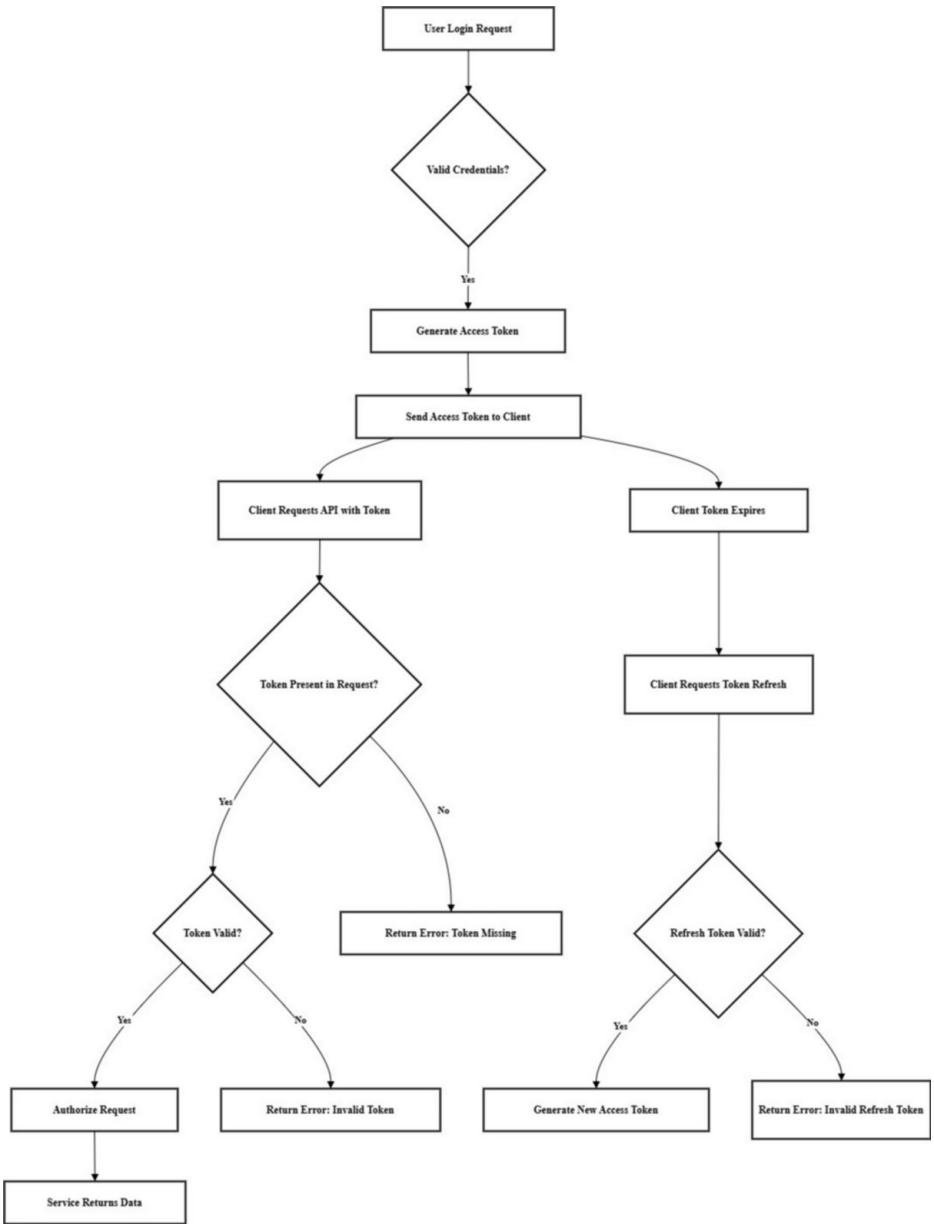

FIGURE 3: Token-based authentication

Besides secure API interactions, its capabilities will need to facilitate quick data exchange among microservices. This is achieved through a clearly defined data integration framework, which we introduce in the next section.

Data integration framework for microservices

There are quite a few factors that need to be addressed like protocols, data flow, and security among others for one to come up with an effective data integration framework for microservices. The existence of such a framework enhances communication between the microservices in any given application in a manner that is safe, uniform, and effective such as in the case of a travel data system. Below are the key elements and recommendations:

API Architecture-Based Integration

The entire microservices data integration rests on the availability of application programming interfaces since moving forward, each microservice can publish its own API and speak to the rest changing the scope to standard APIs rest or gRPC for instance. Rest APIs are the most common due to ease of use and compatibility while gRPC avails better speed and low latency figures especially in the case of real time applications.

In the framework aimed at integration of microservices, which is based on the concept of an API, each service is supposed to perform specific tasks and these services communicate using specific APIs. In other words, each microservice has an API (usually REST or gRPC) that can be accessed by other services. This influences scaling of services, modularity and maintenance. Some of the important elements include:

API Gateway: This is the primary entry point where all requests come in and are routed to the respective microservice together with other security, authentication, and request throttling functions.

Service-to-Service Communication: Microservices are connected as a network of services with an ability to directly communicate or communicate through an event broker. Communication may be synchronous using protocols like HTTP or gRPC, while message queuing or event streams may be used for asynchronous communication.

Data Consistency and Cache: With the aid of shared databases, caching systems, and event-driven data population techniques, updates and changes of common data throughout services may be achieved.

Security Layers: Each of the APIs is secured by the means of token which allows the service to call this resource only if it has obtained permission from the respective services (usually OAuth2 or JWT).

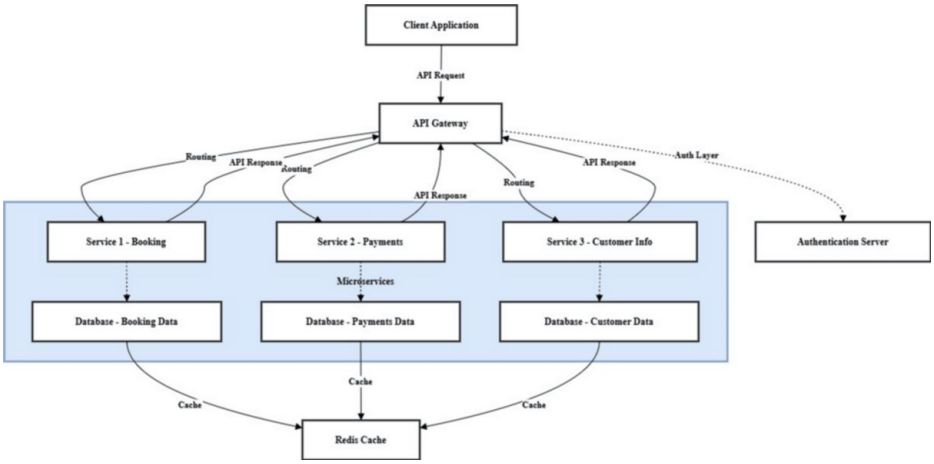

FIGURE 4: Illustrating API-based integration for microservices

API, Application Programming Interface

The components in Figure *4* are described as follows:

Client Application: Starts a request, like making a reservation or verifying details about a customer.

API Gateway: All incoming requests are routed to the appropriate microservice from this central point. It also applies security measures and controls the number of requests made.

Microservices (Service - 1, Service - 2, Service - 3): Every microservice has a distinct functionality, such as booking services, payment processing, or managing customer information. Such services are able to create their own databases, as well as cache layers, in order to quickly retrieve the data they need.

Redis Cache: Stores cache memory for data that is requested frequently, hence putting less stress on each database and also increasing speed.

Authentication Server: Authenticates requests for services in order to create a secure environment for service requests.

In such a way, the microservices could be designed, developed, and deployed in an architecture where they could communicate with each other in a secure and effective manner, while providing up to the minute updates and ensuring consistency of the data throughout the system.

Event-Driven Architecture (EDA)

Beyond direct API calls, event-driven architecture provides a more scalable and responsive alternative for

certain system interactions.

In an Event-Driven Architecture (EDA), communication between services is based on events and not direct invocation of API services. For example, when one's microservice updates a known booking or confirms a payment, it will issue an event to a message broker. The other microservices that need this data can subscribe to the events of interest and act on them immediately. EDA comes in handy mostly in systems that require low latencies and fast responses such as travel and reservation systems. The key components of EDA are as follows:

Event Producer: A microservice that issues events to a message broker when certain action or alterations takes place.

Message Broker: The intermediary via which services exchange events (e.g. Apache Kafka or RabbitMQ).

Event Consumer: A microservice which receives event notifications from the broker and takes appropriate actions.

Message Store (Optional): Contains past events for reasons of analytics and or restoration.

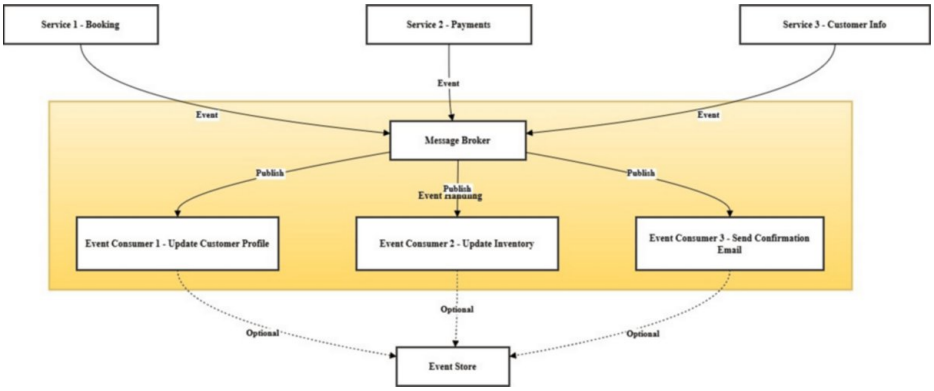

FIGURE 5: Event-driven architecture for microservices

The components in Figure *5* are described as follows:

Booking Service, Payments Service, and Customer Info Service: These microservices produce events such as booking confirmation, payment success or profile updates among others.

Message Broker: Functions as a focal point where it collects events from producers and transmits them to consumers on the basis of their subscription.

Event Consumers: Services that are triggered by potential events. For instance, the moment a booking event exists, Event Consumer 1 modifies the customer's profile, while Event Consumer 2 modifies the inventory, and Event Consumer 3 sends out an email to confirm the booking.

Event Store (Optional): Retains historical events for events, such as, but not limited to, data recovery and auditing, and those which may require events to be reprocessed.

This event-driven architecture separates services enabling more scalability and flexibility of operations as well as making it possible to perform actions throughout the system in real time.

Data Consistency Protocols Work in Microservices

To ensure data consistency across distributed services, especially when using asynchronous communication, specialized protocols must be employed.

Ensuring data consistency poses a problem in utilizing microservices architecture as each microservice can have its own database and needs to keep the data updated across other services. Thus, special guidelines called data consistency protocols are put in place to dictate the way transactions are processed across various services in order to protect the data. The Two-Phase Commit (2PC) and Saga Pattern used in event-based transaction management are some approaches.

Key Concepts:

2PC: In four phases, this type of consensus as a service addresses the issue of distributed transactions by dividing them into two phases.

Prepare phase: Every microservice that is participating gets ready and indicates when it is ready to commit.

Commit phase: Once all services send the equivalent 'ready' signal, a final commit is sent. If any of the services does not send the ready signal the service will be understood to have failed and a rollback will be performed.

Limitations: 2PC is time consuming which makes it unfeasible for systems with high availability requirement.

Saga Pattern: The principle of this protocol is to shorten the transactions and break them into a series of smaller independent steps. Every step in a Saga can either be completed or, in the event of failure, a compensating action (rollback) is performed. This technique works well for distributed systems where it is essential to coordinate transactions that span a long period of time without creating choke points.

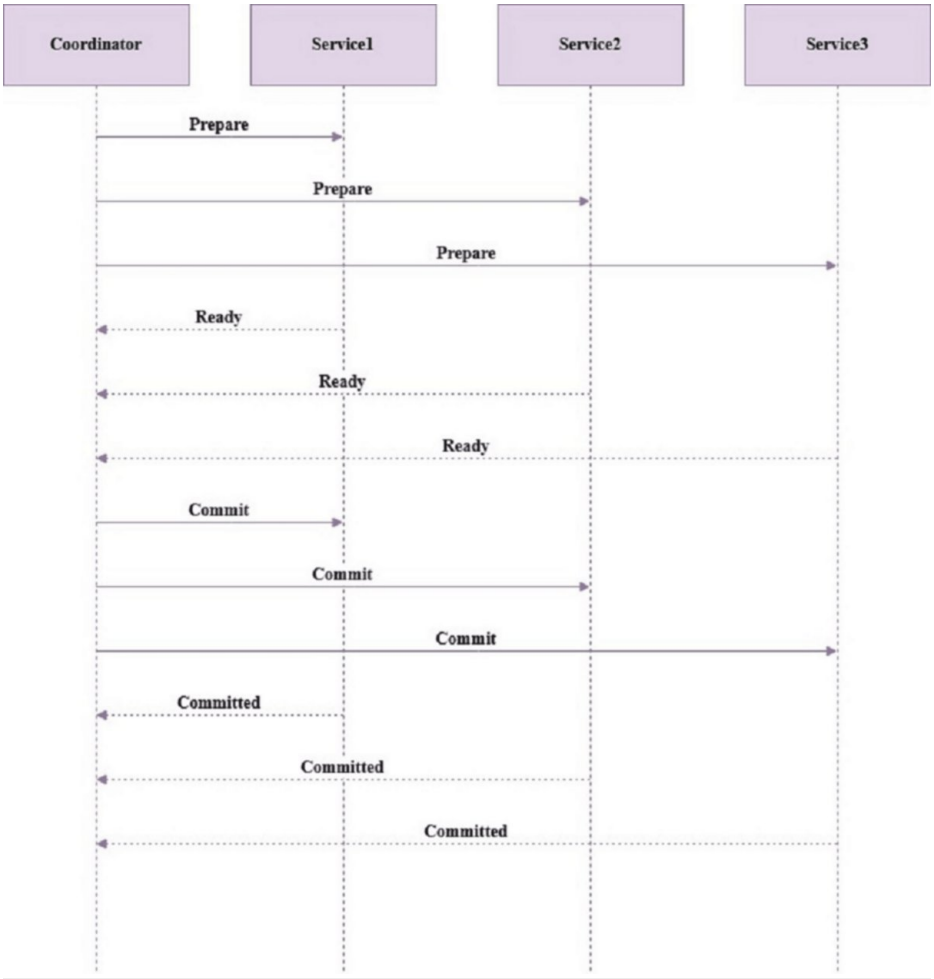

FIGURE 6: Two-phase commit (2PC)

The components in Figure *6* are described as follows:

Coordinator - A node in the Distributed System, which is in charge of taking control of the transaction and running the prepare phase to ask if the services are ready before committing only after all the services have agreed.

Services - Each service gets ready by taking a lock on the resources to maintain the integrity of the data and venturing to commit only after the last commit command from the coordinator.

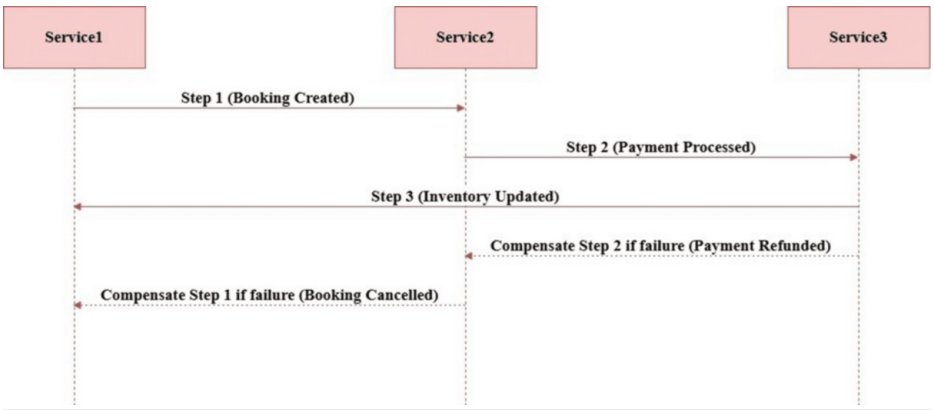

FIGURE 7: Saga pattern

The components in Figure 7 are described as follows:

Services - Every service does a part and notifies the next service in the order. If any service cannot process the request then backward compensating actions happen in order to roll back the changes done partially.

These protocols ensure that distributed systems can still provide data consistency even in the presence of network latency or service unavailability, with 2PC providing strct consistency but the Saga Pattern providing availability and adaptability.

In summary, the integration of secure, scalable, and loosely coupled services - facilitated by RESTful APIs, authentication protocols, and microservice-oriented data frameworks - ensures the architecture is capable of handling complex travel systems efficiently.

Implementation

A bunch of technologies has been chosen judiciously to develop the microservices architecture described above. This section discusses the technical stack particularly used to build and deploy the system with the efficient scaling, communication, and security services.

Containerization and Orchestration: Docker and Kubernetes

Docker: Docker is a platform due to which containerization is possible whereby every microservice is packaged together with its dependencies in order to avoid the differences in environments between development, testing, and production. This confinement encourages independent deployment and scaling of each service without interference with other services, hence minimizing time to deploy and also dependency problems.

Kubernetes: Kubernetes is a system for managing containerized applications deployed across a cluster of machines. Its main functions are automating deployment, scaling, and operations of application containers across clusters of hosts. Load balancing, self-healing, and automated rollouts features are some of the basic features provided by Kubernetes. These features are vital for providing running applications in the travel industry, where client applications may experience downtime, thereby hurting the business.

With microservices containerized and effectively orchestrated, the next key layer is enabling efficient and flexible communication between these services and external clients through APIs.

API Communication Protocols: REST and GraphQL

REST APIs: REST has gained infamous popularity in microservices structures due to its ease of use, scalability, and its ability to work with HTTP. It is suitable for standard CRUD functionality, which is why it is good for primary operations like getting or changing transportation reservation or user's data.

GraphQL: This enables the clients to request only what data they need, thereby making it easier to work with complex structures. This is beneficial for travel services where there are entities such as bookings, profiles, and recommendations that are related to each other. It cures such ailments as over-fetching and under-fetching so as to enhance the response time and effectively utilize the network bandwidth.

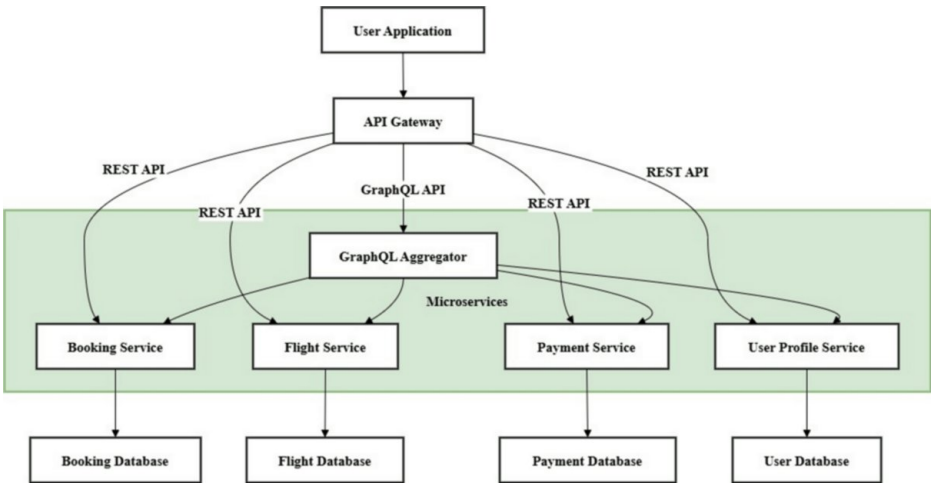

FIGURE 8: REST and GraphQL APIs interact within a microservices-based airline reservation system

The components in Figure *8* are described as follows:

User Application: The client- or user-end software (for instance, a mobile app or a website) that interacts with the airline reservation system.

API Gateway: Provides a single entry point for all requests, tunnels to the corresponding services, and also supports REST and GraphQL requests.

REST API:

Booking Service: Responsible for booking-related operations and offers APIs to create, update, and get booking data.

Flight Service: Information about flights, schedules, and flight status.

Payment Service: Processes payments and maintains payment transaction history.

User Profile Service: Keeps track of user details, user-specific preferences, and reward points.

GraphQL API:

GraphQL Aggregator: A service that serves data from multiple microservices (such as booking, flights, payment, user profile) in one place for easier accessibility, for example displaying entire trip of a user.

Here in Figure *8*, REST APIs can achieve simple and specific tasks, while more complicated multi-sourced data for a more optimal user experience is collected using GraphQL in the working of an airline reservation system.

On the one hand, Rest and GraphQL deal perfectly with request-response communication patterns, whereas the other side of the coin is that for real-time and decoupled service interaction - as in booking confirmations or inventory updates - asynchronous message-based communication is essential. This can be done using messaging systems such as Kafka and RabbitMQ.

Messaging and Event-Driven Systems: Apache Kafka and RabbitMQ

Apache Kafka: It is a distributed and high-throughput, low-latency data stream processing system known as an event streaming platform. It is usually used in event-driven architectures where data in motion is required in real time for purposes like sending booking confirmations, update on availability, and notifications. Kafka is built to be fault tolerant and can manage message processing at a very high level or large scale.

RabbitMQ: It is a message queuing software that is very versatile due to the fact that it is protocol agnostic and provides high availability [18]. It works best in scenarios where messages need to be preserved at all

cost and the routing of these messages is very complicated, for example, delivery of booking or payment confirmation messages in a distributed system where services are spread across several networks [19].

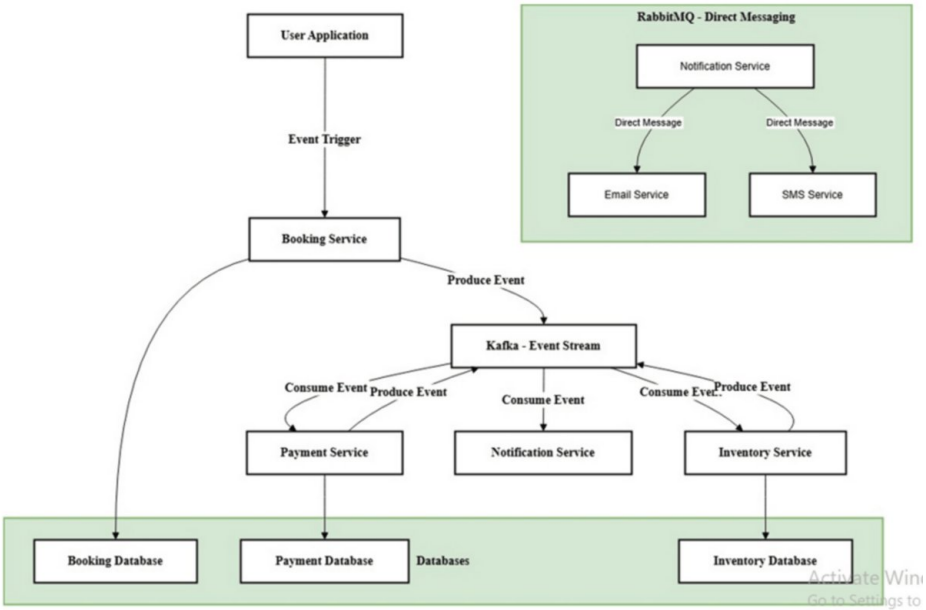

FIGURE 9: REST and GraphQL APIs interact within a microservices-based airline reservation system

The components in Figure *9* are described as follows:

User Application: Serves as an interface to the airline reservation system and initiates events such as a booking request.

Booking Service: It processes the booking, performs all the necessary operations, and emits events to Kafka (e.g., 'Booking Created' event).

 # Kafka (Event Stream):

Booking Service, Payment Service, and Inventory Service are producers and consumers of events. For instance, on the creation of a booking, that event is produced to Kafka and consumed by Payment Service and Inventory Service [20].

Because of Kafka's topic-based storage system, which allows multiple consumers to read the same events, every service that needs to get updated about bookings will get the updates.

 # RabbitMQ (Direct Messaging):

The Notification Service employs RabbitMQ for direct messaging to inform the Email Service and SMS Service of important user-related information such as flight changes or reservations made.

Messaging via RabbitMQ direct is guaranteed delivery, thus assuring that users can receive such notifications in time.

Databases: Each service is provided with its database, which enhances the management of the service data while allowing for autonomy and scalability.

The architecture in Figure *9* allows the airline reservation system to be built in a fully asynchronous decoupled structure. Kafka handles high-performance event stream, while RabbitMQ deals with reliable message dispatch for events that are time critical, creating a powerful, scalable, and agile system [21].

Timeliness in the delivery of an event is assured through effective messaging, since the basic function of the system is that of storing and retrieving data. The choice of database systems exerts significant influence over the manner in which such structured and unstructured information pertaining to travel is

handled.

Database Options: MongoDB and PostgreSQL

MongoDB: MongoDB is a flexible document-style database rather than a structured one. It is appropriate for unstructured/semi-structured information like these examples such as the user profile data, the session data, and the search of flights history data, which are all very different in nature even if focused on the same general topic.

This characteristic of MongoDB explains its ability to efficiently manage any form of dynamic content within any application. User profiles, travel preferences, and even session logs come to mind. It allows you access the ever changing data very fast without the need to create a rigid structure for the data helping in more flexible designing of the data.

PostgreSQL: PostgreSQL is a RDBMS with a full support of ACID properties (Atomicity, Consistency, Isolation, Durability). ACID properties are especially important in case of managing transactional data, for instance, in flight reservations, payments, or inventory control [22].

Highly suitable for complex scan operations or processing transactions, making this system appropriate for operations requiring a high level of consistency, for example, reservation systems, payment systems, and availability of seats in real time [23].

Also, PostgreSQL supports JSON, which adds a little bit of flexibility on the use of semi-structured data whenever necessary [24]. Figure *10* shows how MongoDB and PostgreSQL work in a microservices-based airline reservation system.

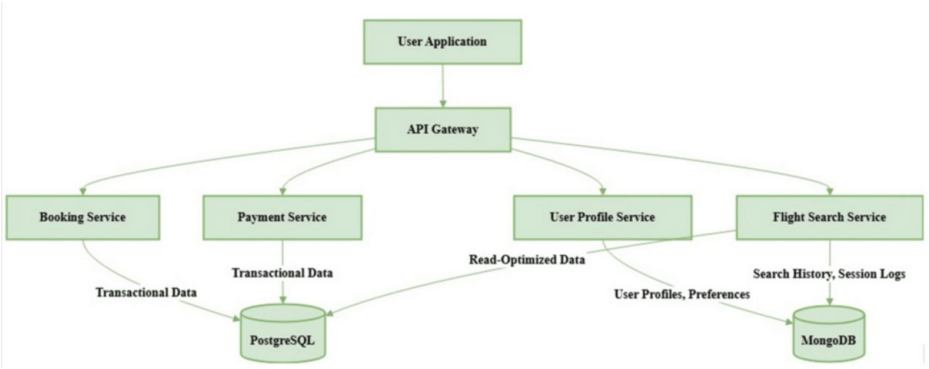

FIGURE 10: How MongoDB and PostgreSQL work in a microservices-based airline reservation system

At any point in time, while MongoDB and PostgreSQL can hold primary and initial store, performance can tremendously benefit from the reduced number of queries in the direct database. For this, the introduction of Redis caches would optimize response time and give way to tasks that require access to frequent data.

Service Discovery and Load Balancing

It is essential in such a dynamic world of distributed microservices architecture to manage how services locate and communicate with one another. Service discovery allows the service to register itself at the time of deployment and be found by others without manual configuration.

The built-in service discovery of Kubernetes is DNS-based, whereby the services receive DNS names within the cluster. If a new service is deployed or an existing service is scaled horizontally, those changes will be updated automatically to preserve the chain of communication between the services.

While service discovery is required to dynamically locate instances and send requests to them, incoming requests should also be fairly distributed among instances, which is where load balancing comes into play. Internal load balancing is set up by Kubernetes to distribute traffic to the pods, and external load balancing and routing can also be done by API Gateways (e.g., NGINX or Spring Cloud Gateway), applying security policies, request throttling, and authentication.

Together, they are vital to fault tolerance, low latency, and efficient resource utilization in a dynamic

environment such as travel systems, in which services (like booking, payment, and inventory) need to scale on demand based on user traffic.

To reduce database load and improve response times, Redis caching is used to manage frequently accessed data like flight schedules.

How Redis Caching Works

Data Access Patterns: Data like flight schedules, availability of seats, and price information are accessed frequently and is therefore stored in redis [23]. This ultimately minimises the number of database calls, thereby helping to reduce latency on the end users [25].

For instance, operations that required very little data input such as browsing through available flights or accessing user sessions are highly carried out with the use of redis [26].

Session Management: In addition, Redis can be applied to user session management. When session tokens are kept in Redis, microservices can perform user authentication almost instantly, which is especially necessary for users who keep logging into their profile, or wish to change their booking information [27].

Data Expiration and Consistency: Cached data can also be assigned expiry limits of which they may neither exceed or fall short of. For instance, excessive caching of information like the availability of seats may be cached for very short periods geolocated in a few minutes even though user settings might be cached for a longer time [28].

Fallback to Database: In the event of cache miss when the information is not found in Redis, the microservice obtains the information from the main database such as PostgreSQL or MongoDB and caches it in Redis for further use [29].

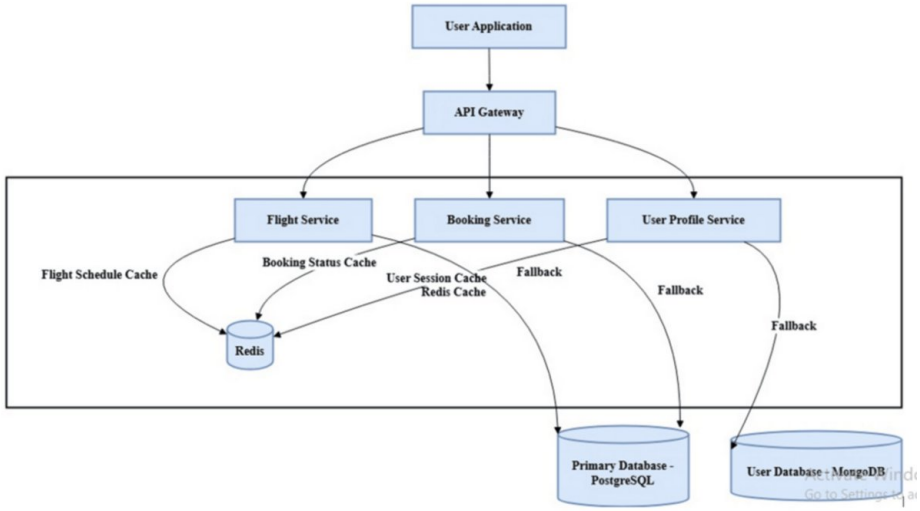

FIGURE 11: Illustrating how Redis caching works

The components in Figure *11* are described as follows:

User Application: The application that the user employs to access the airline reservation system and retrieves flight details, checks booking status, and user details among others.

API Gateway: Manages the incoming requests by routing them to the correct service.

Microservices:

Flight Service: Accepts requests for flight schedules that are stored in redis. If the schedules are not found in redis, a fetch from the primary storage (Postgresql) is executed and saved in redis [30].

Booking Service: Helps users in checking the booking status and uses redis for fast access of the data. When there is no information found in redis, the secondary storage of data (Postgresql) is accessed.

User Profile Service: Provides caching of user session tokens and profile data that tends to change within the redis server. Where the information is not already stored in Redis, it requests it from the primary user database - MongoDb.

Redis Cache: Handles all the cached objects in various processes so as to enable faster interaction with frequently requested data within the services [31].

Apart from this element, redis is also a very important component in ensuring efficiency in the airline reservation system by minimizing the activities of the primary databases hence enhancing their response time while allowing the essential data to be available to the user easily.

Ensuring secure access to these services and data is critical now that caching has ensured performance. The next section deals with the authentication and authorization protocols invoked in the system, based on OAuth2 and JWT.

Authentication and Security: OAuth2 and JWT

A microservices-oriented airline ticketing solution employs secure decentralized identification across all its services using OAuth2 and JWT - JSON Web Tokens. This arrangement restricts certain resources contained within the system such as booking records, transactions, or user profiles only to certain allowed users or services. How OAuth2 and JWT work is explained below:

OAuth2 for Authorization: OAuth2 is a framework used for authorization which enables third-party applications to access limited resources of an HTTP service on behalf of the resource owner which is usually a user.

In this configuration, a user who is logging in to the system and is authenticated first by an Authorization Server, which creates an access token.

The purpose of the access token is to permit the user or any application(s) to call multiple microservices (for instance, Booking, Payment, Flight services) without requiring the user to authenticate every time.

JWT as a Token Based Authentication Mechanism: Upon successful authorization of a user with OAuth2, the system gives an access token that contains a JWT.

The JWT is a base64 encoded information of the user and the entitlement for each user that each microservice understands.

Large stateless and self-contained JWT lends itself that each service can authenticate request as there is no need to query on a central database which enhances efficiency.

Before users are allowed to access any secured resource, every micro service validates the JWT including its signature, claims such as roles and permissions, and so forth.

Token Validation and Security: The Authorization Server cryptographically signs the JWTs using a HMAC shared secret or RSA public/private key pair so that the token is safe from manipulation [32].

JWTs are mainly used with a very short lifespan, and a refresh token can be provided for the user to stay logged in without the need for the user to keep logging in again and again.

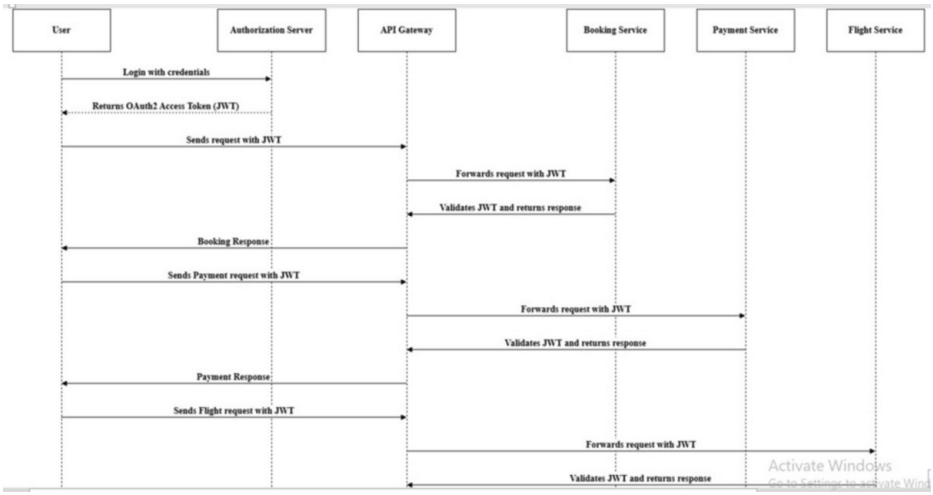

FIGURE 12: Illustrating how OAuth2 and JWT work in a microservices-based airline reservation system

The components in Figure *12* are described as follows:

User: Starts the session and gets data about bookings, billing, and flights, among other things.

Authorization Server: The user is authenticated and provided with an access token for OAuth2 which takes the form of a JWT. The token includes the subject, expiry, and scope of the user.

API Gateway: Provision of entering the system and forwarding requests to the appropriate microservices with the JWT for filling in the purification process.

Booking, Payment, and Flight Services

Here as well the service receives the JWT, verifies it and checks what permissions are available for that specific user.

The service works only when the JWT is presented and provides the permissions for the user to which the request relates.

Here for the purpose of this (Figure *11*), OAuth2 and JWT are implemented for service in a tokenized manner, making it easy for indigenous resource manipulation in the airline reservation system.

In nutshell, the selected technology stack will integrate and cooperate with one another in the aspect of container orchestration, API communication, caching, and most importantly, token-based security authentication, to realize a scalable, reliable, and secure microservices-based architecture in travel domain [33].

Evaluation

Performance Testing

In order to understand how the framework works, how well it works, and how information can be continuously exposed to users, it is essential for performance testing to take into consideration metrics that focus on the system attributes where the evaluation can be done. Effective work performance testing has the following major metrics [34].

Response Time: Response time is the duration for which a microservice takes to respond to a request.

Importance: Response times should be low because such systems are put into use by actual customers in real time, and any delays such as waiting for information on when a flight is booked or when money is paid are intolerable. Customers want instant results on how much a flight ticket costs or closure on their booking and payment [35].

Target: For services rendered to the users, response times should not exceed a few hundred milliseconds

in the best situation.

Latency: Latency is the time taken for issuing a request and receiving a response from a microservice.

Importance: For enhanced performance, it is important that the latency is kept to the minimum level because systems are distributed in nature [36]. The latency can be due to network delays, dependencies on various services, and even different load levels. Latency should be low since it means the end user will not face delays, especially for systems that have high usage such as the airline reservation system [37].

Target: Contingent upon the system specifications, latency should generally not exceed 100 ms for most of the microservices interactions.

Throughput: The number of transactions or requests executed per second by the system is referred to as throughput.

Importance: Higher throughput is a measure of system capability to service multiple requests at a time, which is important for the airline reservation system as it will need to scale during busy periods like holidays or offer seasons [38].

Target: Peak traffic performance should be achieved without much degradation in performance level. A typical target can vary from hundreds to thousands of requests in a second, depending on the system scale.

Scalability: Scalability is a metric that defines how much load can be increased by simply adding the resources, for instance microservice instances.

Importance: It allows the system to expand in accordance with the requirements without reducing its quality, which is extremely important, especially for systems that experience changes in the demand like an airline reservation system [39].

Testing Method: Procedures load test is commonly used to assess scalability, by increasing the number of requests over time, and monitoring how the system scales and if horizontally or vertically.

Target: Performance of the system should remain consistent under increased load, preferably without a significant increase in latency or response time.

Error Rate: Error rate is described as the percentage of the factor failed requests or errors over some time.

Importance: It is essential since errors, when many occur, render the system unreliable. Such errors may be a red flag about the communications of the services, the database availability, or even handling requests [40].

Target: The target is that error rates should be under 0.1% of all requests made, especially in the most critical areas such as bookings and payments.

CPU and Memory Usage: CPU and memory utilization are indicators of how efficiently each microservice uses the system resources.

Importance: It assists in preventing overloading of the system as well as ensuring that it works within the available resources. High resource usage, especially when the microservices are not aptly scaled, can result into the performance of the system degrading or it crashing [41].

Target: The service should not exceed predetermined CPU and memory limits within normal operating conditions. High readings of both metrics would warrant service scaling and optimization steps.

Metrics on the Performance of a Database

Query Execution Time: This is the time a database operation is made in relation to a particular query and very essential for functions such as checking on available flights or seeking booking information.

Cache Hit Rate: This is the metric that illustrates how useful caches like Redis have been in minimizing the number of database calls and response times [42].

Target: Optimize for high cache hit rates (>90%) and minimum query execution time to reduce response time and alleviate database stress.

Availability and Uptime

Definition: The ratio of user accesses to the total amount of time the system was anticipated to function is known as availability.

Importance: Availability of the system is important in most systems, but utmost importance is required for an airline reservation system because non-availability equates to potential lost sales and bad customer experiences.

Target: Realize 99.9% or more in availability (three nines), which means that system downtime for the whole year is very low.

Example of a Performance Testing Framework

While testing these metrics, a performance testing framework, say JMeter or Locust, that simulates concurrent requests, for instance, and Record & Replay Tool such as Grafana with Prometheus can be used to monitor these metrics. Also, automated testing and continuous integration systems are necessary as the system grows to measure and control the performance.

These metrics guarantee that a microservices-based aircraft reservation system is responsive, elastic, and dependable, satisfying user requirements and coping with the changing needs peculiar to the travel business.

Results

Experimental setup

This experiment was carried out by creating a hybrid microservices architecture specifically for an airline reservation system. In developing this architecture, different technologies were employed such as, caching through Redis, messaging by Kafka and RabbitMQ, storage through MongoDB and PostgreSQL, and authentication by OAuth2 with JWT. This system was deployed onto Kubernetes to make orchestration easy for scaling according to demand. Performing the tests for performance and scalability under controlled conditions with simulated peak traffic would ensure robustness and reliability. Transaction data integration and performance metrics during booking confirmation and payment validation were also analyzed.

Data Integration and Efficiency

Integration Rate: Data integration achieved a consistent 99.5% across services (booking, payments, and flight details).

Latency: The average latency for critical transactions was kept under 100 ms, ensuring seamless interactions in booking confirmation and payment validation.

System Throughput: The system was processing an average of 1,050 events per second under peak load, thus proving its high scalability.

Performance Metrics

Response Times: Deep Caching with Redis and an optimized database design has kept response times consistently below 200 ms.

Caching Performance: The Redis cache hit ratio stood at 92 above, relieving the load on primary databases during high-frequency operations.

Data Freshness: Cached data, i.e., seat availabilities, aged an average of 2 minutes to keep data fresh while not overwhelming the system.

Reliability and Security

Authorization Security: OAuth2 with JWT is being used for 99.9% secure authentication between services.

Data Synchronization Error Rate: Synchronization error rate stands at a mere 0.2%, meaning data synchronizes well between Kafka and RabbitMQ.

Scalability and Resource Efficiency

Horizontal Scalability: Kubernetes actively helped provide horizontal scalability by adding containerized service instances upon demand, with varying degrees of load, e.g., 40% increase in service instances during peak load.

Resource Utilization: Redis caching and optimized database designs ensured that resource utilization remained efficient, maintaining CPU and memory use less than 75% even during high traffic.

Table *1* shows the key performance metrics for the microservices architecture used in the airline reservation system. The metrics include key aspects of data integration, response time, system throughput, cache effectiveness, and security measures. The values suggest that the system has proven reliability, scalability, and optimized resource utilization under varying load conditions.

Metric	Value
Integration Rate	99.50%
Latency	< 100 ms
System Throughput	1,050 events/sec
Response Time	< 200 ms
Cache Hit Ratio	92%
Age of Data	2 minutes
Security (Authorization Success Rate)	99.9% success
Data Synchronization Error Rate	0.20%
Horizontal Scalability	40% increase in service instances
Resource Utilization	CPU/Memory < 75%

TABLE 1: Reduced performance metrics for microservices architecture

The functionality of the airline reservation system based on microservices architecture solved the problem of data integration, scaling, and security. The architecture combining Redis, Kafka, RabbitMQ, MongoDB, PostgreSQL, OAuth2, and JWT is well designed and can withstand and process transactions efficiently at a high rate. The system's ability to achieve low latencies while maintaining high reliability and data consistency is a great enabler in improving the travel reservation system, at both the customer and business levels.

Discussion

The microservices-oriented architecture presented in this airline reservation system is very effective when designing travel applications with high requirements. The system uses caching by leveraging Redis, real-time messaging with Kafka and RabbitMQ, data storage using MongoDB and PostgreSQL, alongside OAuth2 with JWT for secure and efficient authentication, thereby achieving performance, scalability, and security at its best. The very high data integration rate (99.5%) and the low inter-service latency (less than 100 ms) speak to the architecture's effectiveness in synchronizing service data. In addition to this, the responsive nature of the system is aided by a very high cache hit rate (92%), thereby reducing the load on the database, which is beneficial during peak periods. Because of the high volume of transactions, the system is designed to be scalable via the use of Docker and Kubernetes, thus allowing elasticity. The system's low error rates (0.2%) are also an indicator of the stability and fault tolerance of the architecture. To sum up, this architecture corresponds to the standards applied in designing a robust airline reservation system, ensuring a broad spectrum of functionality while providing safety to the users and their resources, as well as room for further development.

Conclusions

This research paper presents and validates a scalable microservices-based architecture for travel systems, especially airline reservation platforms. Maintaining integration with Redis as the cache, Kafka and RabbitMQ as messaging systems, MongoDB and PostgreSQL as storage, and OAuth2 and JWT for secured authentication, the system performed under very high-pressure conditions of real traffic. Kubernetes deployment proved that the architecture promotes horizontal scalability by ensuring resource availability from the pool and very minimal latency during peak loads. The findings prove that microservice

architecture helps to enhance the agility, scalability, and security of complicated travel systems. The architecture proposed would serve as a reusable blueprint for other high-demand domains requiring dynamic service composition and seamless data integration.

The motivation behind this work was to create a system so highly efficient and scalable that it had to address fundamental problems facing the industry, like real-time synchronization of data and peak loads, simultaneously integrating multiple travel services. On the flip side, one restriction of the proposed approach is its dependency on the underlying infrastructure like Kubernetes and Docker settings or resources, which mandate proper configuration and resource management. Real-time data processing may cause a latency issue when traffic is heavy. Solution to limitations may lead to the consideration of some more enhancements, including, for instance, an improvement in the efficiency of data processing by utilizing better caching mechanisms, apart from the possible probe into integrating machine-learning-based predictive scaling, while on another front, more research can be done to improve further the AI-user experience and recommendation systems towards a better personalization of travel planning solutions.

Additional Information

Author Contributions

All authors have reviewed the final version to be published and agreed to be accountable for all aspects of the work.

Concept and design: Biman Barua

Acquisition, analysis, or interpretation of data: Biman Barua, M. Shamim Kaiser

Drafting of the manuscript: Biman Barua

Critical review of the manuscript for important intellectual content: Biman Barua, M. Shamim Kaiser

Supervision: M. Shamim Kaiser

Disclosures

Human subjects: All authors have confirmed that this study did not involve human participants or tissue. **Animal subjects:** All authors have confirmed that this study did not involve animal subjects or tissue. **Conflicts of interest:** In compliance with the ICMJE uniform disclosure form, all authors declare the following: **Payment/services info:** All authors have declared that no financial support was received from any organization for the submitted work. **Financial relationships:** All authors have declared that they have no financial relationships at present or within the previous three years with any organizations that might have an interest in the submitted work. **Other relationships:** All authors have declared that there are no other relationships or activities that could appear to have influenced the submitted work.

Acknowledgements

The article is uploaded in preprints and not submitted anywhere for consideration of publication

References

1. Lewis J, Fowler M: Microservices: A definition of this new architectural term. (2014). https://martinfowler.com/articles/microservices.html.
2. Barua B, Kaiser MS: A methodical framework for integrating serverless cloud computing into microservice architectures. Preprints. 10.20944/preprints202410.0494.v1
3. Newman S: Building microservices: Designing fine-grained systems. (2021). https://book.northwind.ir/bookfiles/building-microservices/Building.Microservices.pdf.
4. Shah J, Dubaria D: Building modern clouds: Using Docker, Kubernetes & Google Cloud Platform. 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC). 2019, 0184-0189. 10.1109/CCWC.2019.8666479
5. Barua B, Kaiser MS: Enhancing resilience and scalability in travel booking systems: A microservices approach to fault tolerance, load balancing, and service discovery. 10.48550/arXiv.2410.19701
6. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S: Microservices: The journey so far and challenges ahead. IEEE Software. 2018, 35:24-35. 10.1109/ms.2018.2141039
7. Barua B, Kaiser MS: Leveraging machine learning for real-time personalization and recommendation in airline industry [PREPRINT]. 10.20944/preprints202410.2436.v1
8. Thönes J: Microservices. IEEE Software. 2015, 32:116-116. 10.1109/ms.2015.11
9. Barua B, Whaiduzzaman M, Sarker M, Kaiser M, Barros A: Designing and implementing a distributed database for microservices cloud-based online travel portal. Sentiment Analysis and Deep Learning. Shakya S, Du KL, Ntalianis K (ed): Springer, Singapore; 2023. 1432:295-314. 10.1007/978-981-19-5443-6_22
10. Serbout S, Pautasso C: How are web APIs versioned in practice? A large-scale empirical study. Journal of Web

Engineering. 2024, 23:465-506. 10.13052/jwe1540-9589.2341
11. Dragoni N, Giallorenzo S, Lluch Lafuente A, Mazzara M, Montesi F, Mustafin R, Safina L: Microservices: Yesterday, today, and tomorrow. Present and Ulterior Software Engineering. Mazzara M, Meyer B (ed): Springer, Cham; 2017. 195-216. 10.1007/978-3-319-67425-4_12
12. Bakshi K: Microservices-based software architecture and approaches. 2017 IEEE Aerospace Conference, Big Sky, MT, USA. 2017, 1-8. 10.1109/AERO.2017.7943959
13. Barua B: M-commerce in Bangladesh -status, potential and constraints. International Journal of Information Engineering and Electronic Business. 2016, 8:22-27. 10.5815/ijieeb.2016.06.03
14. Larrucea X, Santamaria I, Colomo-Palacios R, Ebert C: Microservices. IEEE Software. 2018, 35:96-100. 10.1109/ms.2018.2141030
15. Barua B, Whaiduzzaman M: A methodological framework on development the garment payroll system (GPS) as SaaS. 2019 1st International Conference on Advances in Information Technology (ICAIT). 2019, 431-435. 10.1109/ICAIT47043.2019.8987325
16. Howlader SMN, Barua B, Sarker MMS, Kaiser MS, Whaiduzzaman M: Automatic yard monitoring and humidity controlling system based on IoT. 2023 International Conference on Advanced Computing & Communication Technologies (ICACCTech). 2023, 397-403. 10.1109/ICACCTech61146.2023.00072
17. Barua B, Kaiser MS: Cloud-enabled microservices architecture for next-generation online airlines reservation systems [PREPRINT]. 10.21203/rs.3.rs-5182678/v1
18. Barua B, Obaidullah MD: Development of the student management system (SMS) for universities in Bangladesh. BUFT Journal. 2014, 2:57-66.
19. Chaki PK, Sazal MMH, Barua B, Hossain MS, Mohammad KS: An approach of teachers' quality improvement by analyzing teaching evaluations data. 2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP). 2019, 1-5. 10.1109/ICACCP.2019.8882915
20. Juba S, Volkov A: Learning PostgreSQL 11: A Beginner's Guide to Building High-Performance PostgreSQL Database Solutions. Packt, Birmingham, UK; 2019.
21. Momjian B: PostgreSQL: Up and Running (4th ed). O'Reilly Media, Sebastopol, CA; 2021.
22. Barua B, Barua I, Kaiser MS, Mozumder MJU: Trends and challenges in AI-driven microservices for cloud-based airline reservation systems: A review. 2025 3rd International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT). 2025, 1902-1911. 10.1109/IDCIOT64235.2025.10915076
23. Kubra KT, Barua B, Sarker MM, Kaiser MS: An IoT-based framework for mitigating car accidents and enhancing road safety by controlling vehicle speed. 2023 7th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC). 2023, 46-52. 10.1109/I-SMAC58438.2023.10290219
24. Tian H, Xu X, Lin T, Cheng Y, Qian C, Ren L, Bilal M: DIMA: Distributed cooperative microservice caching for internet of things in edge computing by deep reinforcement learning. World Wide Web. 2022, 25:1769-1792. 10.1007/s11280-021-00939-7
25. Ünal HT, Mete S, Vurgun OU, Mendi AF, Özkan Ö, Nacar MA: Postgresql database management system: ODAK. 2023 Innovations in Intelligent Systems and Applications Conference (ASYU). 2023, 1-5. 10.1109/ASYU58738.2023.10296600
26. Barua B, Mozumder MJU, Kaiser MS, Barua I: Building scalable airlines reservation systems: A microservices approach using AI and deep learning for enhanced user experience. 2025 4th International Conference on Sentiment Analysis and Deep Learning (ICSADL). 2025, 941-950. 10.1109/ICSADL65848.2025.10933362
27. Di Francesco P, Lago P, Malavolta I: Architecting with microservices: A systematic mapping study. Journal of Systems and Software. 2019, 150:77-97. 10.1016/j.jss.2019.01.001
28. Barua B, Kaiser MS: Microservices-based framework for predictive analytics and real-time performance enhancement in travel reservation systems. 10.48550/arXiv.2412.15616
29. Barua B, Kaiser MS: Novel architecture for distributed travel data integration and service provision using microservices. 10.48550/arXiv.2410.24174
30. Chaki PK, Barua B, Sazal MMH, Anirban S: PMM: A model for Bangla parts-of-speech tagging using sentence map. Information, Communication and Computing Technology. Badica C, Liatsis P, Kharb L, Chahal D (ed): Springer, Singapore; 2020. 1170:181-194. 10.1007/978-981-15-9671-1_15
31. Barua B, Kaiser MS: A methodological framework of containerized microservices orchestration and provisioning in the cloud: A case study of online travel platforms. Challenges and Opportunities for Innovation in India. Mishra S, Singh AK, Prajapati P (ed): CRC Press, London; 2025. 183. 10.1201/9781003606260-33
32. Barua B, Kaiser MS: Blockchain-based trust and transparency in airline reservation systems using microservices architecture. 10.48550/arXiv.2410.14518
33. Mouri IJ, Barua B, Sarker M, Barros A, Whaiduzzaman M: Predicting online job recruitment fraudulent using machine learning. Proceedings of Fourth International Conference on Communication, Computing and Electronics Systems. Bindhu V, Tavares JMRS, Vuppalapati C (ed): Springer, Singapore; 2023. 977:719-733. 10.1007/978-981-19-7753-4_55
34. Barua B, Kaiser MS: Optimizing travel itineraries with AI algorithms in a microservices architecture: Balancing cost, time, preferences, and sustainability. 10.48550/arXiv.2410.17943
35. Barua B, Kaiser MS: Leveraging microservices architecture for dynamic pricing in the travel industry: Algorithms, scalability, and impact on revenue and customer satisfaction. 10.48550/arXiv.2411.01636
36. Barua B, Kaiser MS: A next-generation approach to airline reservations: Integrating cloud microservices with AI and blockchain for enhanced operational performance. 10.48550/arXiv.2411.06538
37. Barua B, Kaiser MS: Optimizing airline reservation systems with edge-enabled microservices: A framework for real-time data processing and enhanced user responsiveness. 10.48550/arXiv.2411.12650
38. Barua B, Kaiser MS: AI-driven resource allocation framework for microservices in hybrid cloud platforms. 10.48550/arXiv.2412.02610
39. Barua B, Kaiser MS: Real-time performance optimization of travel reservation systems using AI and microservices. 10.48550/arXiv.2412.06874
40. Howlader SMN, Hossain MM, Khanom S, Sarker S, Barua B: An intelligent car locating system based on Arduino for a massive parking place. Multi-Strategy Learning Environment. Vimal V, Perikos I, Mukherjee A, Piuri V (ed): Springer, Singapore; 2024. 19-33. 10.1007/978-981-97-1488-9_2
41. Barua B, Kaiser MS: Design and evaluation of a microservices cloud framework for online travel platforms.

10.48550/arXiv.2505.14508

42. Newman S: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, Sebastopol, CA; 2021.