

Assignment 3

Architectural Patterns

Submitted By

TEAM_11

Vilal, Hanuma, Shriom, Madan

12/04/2024

**Architectural Patterns Analysis and Implementation for
Enhancing Univaq Street Science Event Experience:
A Comparative Study**

Vilal, Hanuma, Shriom, Madan

CS6.401 Software Engineering

1. Introduction

In the context of the given assignment, the understanding of requirements involves analyzing various aspects of the problem domain. The problem domain revolves around improving the visitor experience at the Univaq Street Science event during the European Researchers Night in L'Aquila, Italy. The event aims to bring together the research community and the public for a day of entertainment and information sharing. Key aspects of the problem domain include managing venue availability, parking lot availability, booking/ticketing services, accessibility services, weather forecast services, recommendation systems, and visualization/analytics for event administrators.

2. Observations

2.1. Key Observations:

1. Univaq Street Science event attracts approximately 35,000 visitors annually.
2. Weather forecast services can aid visitors in planning their activities.
3. Recommendations based on visitor preferences can enhance their experience.
4. Late hours of the event experience higher crowds compared to early hours.
5. Visualization and analytics tools are needed for event administrators.

2.2. Goal:

1. Improve the quality of visitor experience at Univaq Street Science.
2. Provide real-time updates on venue availability and parking lot status.
3. Enhance accessibility for all attendees, including those with disabilities.

2.3. Constraints:

1. Limited availability of parking lots, with only two ad-hoc lots created.
2. Scalability requirement to support up to 1000 requests/second.
3. Use of battery-powered sensors due to limited external power sources.

3. Requirements for the System

3.1. Functional Requirements:

1. **FR1: Venue Availability:** Visitors should be able to check the availability of venues and receive suggestions for alternative venues if the preferred one is full.
2. **FR2: Parking IoT Availability:** Visitors should be able to check the availability of parking spaces and receive suggestions for nearby lots based on venue preference.
3. **FR3: Booking/Ticketing Service:** Visitors should be able to book tickets for entertainment events, considering venue capacity.
4. **FR4: Accessibility Services:** Ensure accessibility for all attendees, including wheelchair-accessible routes, designated parking spaces, and sign language interpreters.
5. **FR5: Weather Forecast Services:** Provide visitors with weather forecasts to help them plan indoor or outdoor activities.
6. **FR6: Recommendation:** Allow visitors to mark preferences and receive event recommendations based on popularity and personal interests.
7. **FR7: Visualization and Analytics:** Provide event administrators with analytical insights into attendance, venue capacity, and parking lot occupancy.

3.2. Non-Functional Requirements:

1. **NFR1: Scalability:** The system should support up to 1000 requests/second due to the large number of attendees.
2. **NFR2: Reliability:** The system should be reliable, ensuring accurate information and minimal downtime.

3. **NFR3: Power Efficiency:** Battery-powered sensors should be intelligently managed to conserve power.
4. **NFR4: Performance:** The system should be responsive, providing real-time updates, recommendations and must provide 1 second response time or less in web browsers.
5. **NFR5: Security:** Ensure data privacy and protection against unauthorized access or breaches.

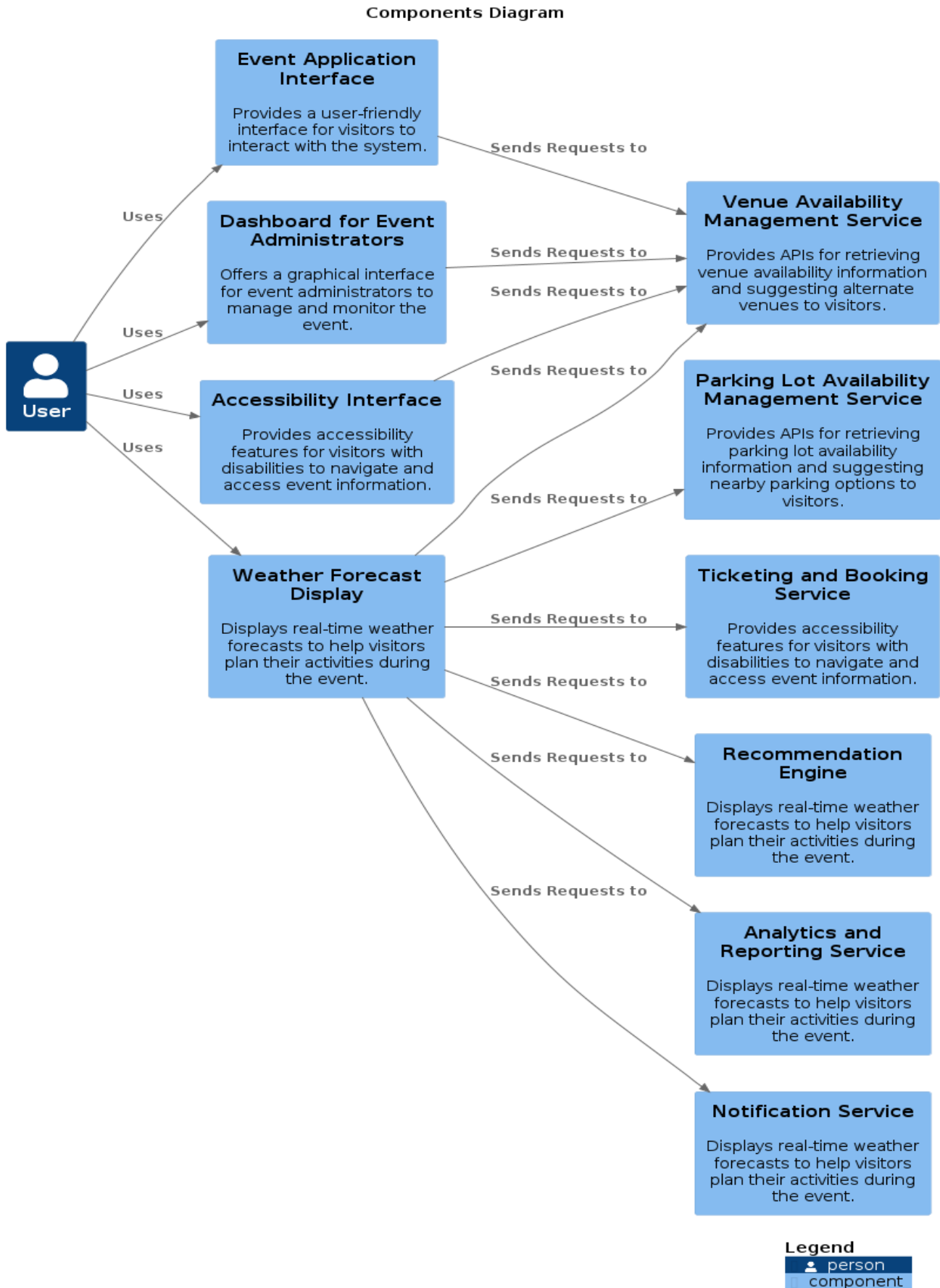
4. System Stakeholders



5. Concerns



6. Components Diagram



7. Architectural Design Decisions

For the Univaq Street Science event system, several architectural design decisions need to be made to ensure the system's effectiveness, scalability, and maintainability. Here are some key architectural design decisions:

7.1. Microservices Architecture vs. Monolithic Architecture

- **Decision:** Choose between a microservices architecture, where each functionality is implemented as a separate service, or a monolithic architecture, where all functionalities are integrated into a single application.
- **Justification:** Microservices offer greater scalability, flexibility, and fault isolation but introduce complexity in managing multiple services. Monolithic architecture simplifies development and deployment but may become cumbersome to maintain and scale as the system grows.

7.2. Event-Driven Architecture

- **Decision:** Adopt an event-driven architecture to decouple system components and enable real-time updates and responsiveness.
- **Justification:** Event-driven architecture facilitates loose coupling between system components, allowing them to communicate asynchronously via events. This enhances flexibility, scalability, and responsiveness, especially for functionalities such as venue availability updates and parking lot notifications.

7.3. Event-Driven Architecture

- **Decision:** Design robust and well-documented APIs for seamless integration with external systems, such as weather forecast providers, parking lot sensors, and ticketing platforms.
- **Justification:** Well-designed APIs enable interoperability, allowing the system to leverage external services and data sources effectively. This enhances functionality, accuracy, and reliability, enabling features like weather forecast integration, parking lot availability updates, and ticket booking.

7.4. Data Storage and Management

- **Decision:** Choose appropriate data storage solutions (e.g., relational databases, NoSQL databases, data lakes) for storing and managing event data, user preferences, venue availability, and parking lot status.
- **Justification:** Selecting the right data storage solutions ensures scalability, performance, and data integrity. Relational databases may be suitable for structured data, while NoSQL databases or data lakes may be preferred for handling unstructured or semi-structured data.

7.5. Scalability and Performance Optimization

- **Decision:** Design the system with scalability and performance optimization in mind, using techniques such as horizontal scaling, caching, load balancing, and asynchronous processing.
- **Justification:** The system should be capable of handling spikes in traffic and processing large volumes of data efficiently, especially during peak event hours. Scalability and performance optimization ensure smooth operation and responsiveness, enhancing the overall visitor experience.

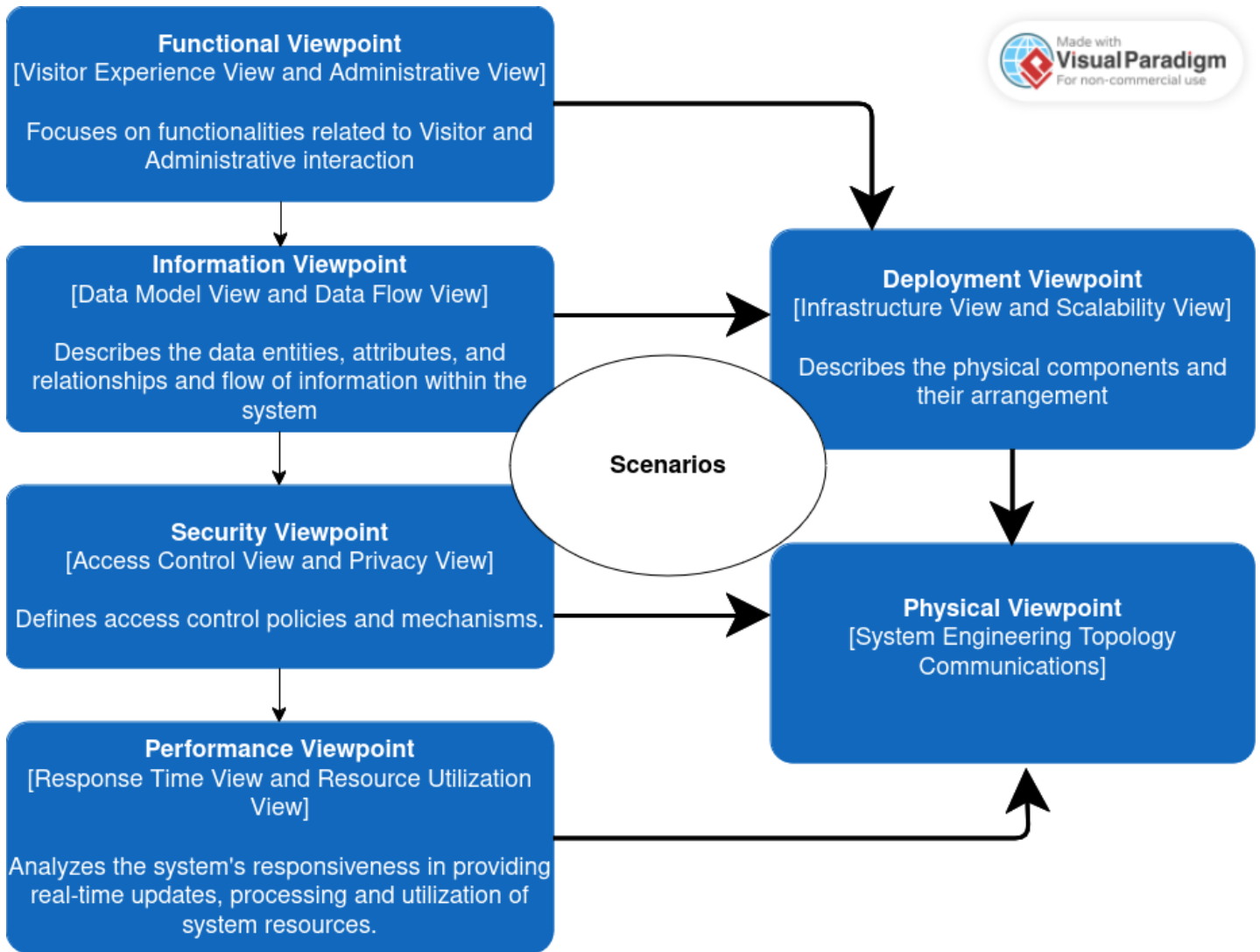
7.6. Security and Privacy Measures

- **Decision:** Implement robust security measures, including encryption, authentication, authorization, and data anonymization, to protect visitor information and ensure compliance with data privacy regulations.
- **Justification:** Security breaches or privacy violations can severely impact visitor trust and reputation. Implementing strong security measures helps safeguard sensitive data and mitigate risks associated with unauthorized access or data breaches.

7.7. Monitoring and Logging

- **Decision:** Implement comprehensive monitoring and logging solutions to track system performance, detect anomalies, and troubleshoot issues in real-time.
- **Justification:** Monitoring and logging provide visibility into system health, allowing administrators to identify and address issues proactively. This improves system reliability, availability, and maintainability, minimizing downtime and disruptions during the event.

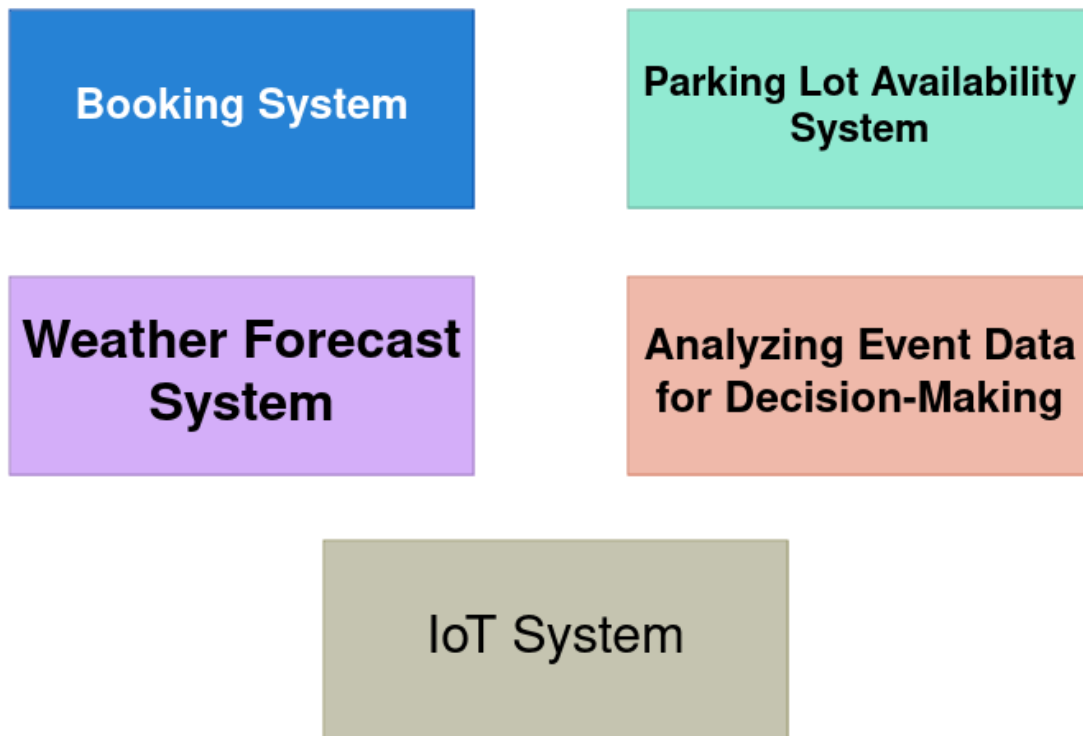
8. Views and Viewpoints



9. Scenario

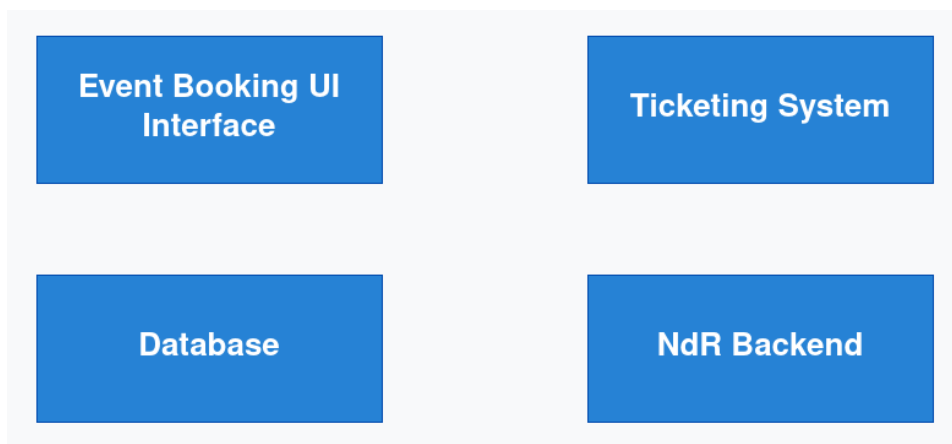


10. Subsystems

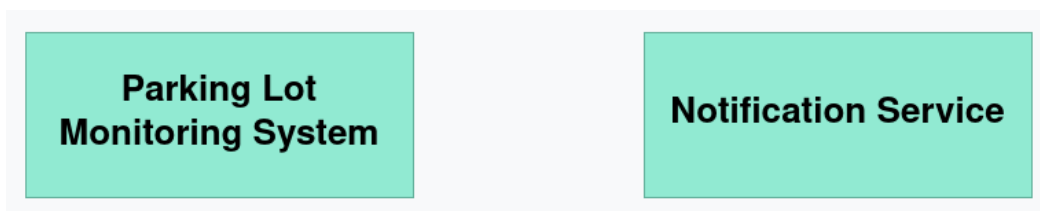


11. Further Subsystems

11.1. Booking System



11.2. Parking Lot Availability System



12. Microservices

In the context of the Univaq Street Science event system's booking system, here are potential microservices and their justifications:

12.1. Booking Service:

- Handles the booking of tickets for entertainment events. By isolating this functionality into a separate microservice, it becomes easier to manage and scale independently. It also facilitates modular development and maintenance.

12.2. Venue Capacity Management Service:

- Manages the seating capacity of event venues and tracks available seats. Isolating this functionality allows for dynamic adjustments to venue capacities and real-time updates on seat availability, enhancing the booking experience for visitors.

12.3. Payment Processing Service:

- Handles payment transactions for ticket bookings. Separating payment processing into its own microservice ensures secure and reliable handling of financial transactions. It also allows for integration with various payment gateways and compliance with payment industry standards.

12.4. Notification Service:

- Sends notifications to visitors regarding booking confirmations, event updates, and reminders. Isolating notification functionality enables efficient communication with visitors through various channels (e.g., email, SMS, push notifications), enhancing user engagement and experience.

12.5. Integration and Communication Service:

- Integrates with external systems and services, such as weather forecast providers, parking lot sensors, and ticketing platforms. Isolating integration and communication functionality facilitates seamless communication between microservices and external systems, ensuring interoperability and reliability.

13. Implemented Architectural Patterns in Ticket Booking System

For the booking system in the Univaq Street Science event system, I would select the Publish Subscribe Pattern and the Service Oriented Pattern. Here's how each pattern would be utilized:

13.1. Publish Subscribe Pattern:

- This pattern is suitable for handling event-driven communication between components.
- In the booking system, when a visitor books a ticket for an entertainment event, the booking service publishes an event indicating the successful booking.
- Subscribed components, such as the notification service and the analytics service, receive the event and take appropriate actions:
 - The notification service sends a confirmation message to the visitor's device.
 - The analytics service updates event attendance metrics and generates reports on ticket sales.

13.1.1. Quality Attributes and Their Explanations

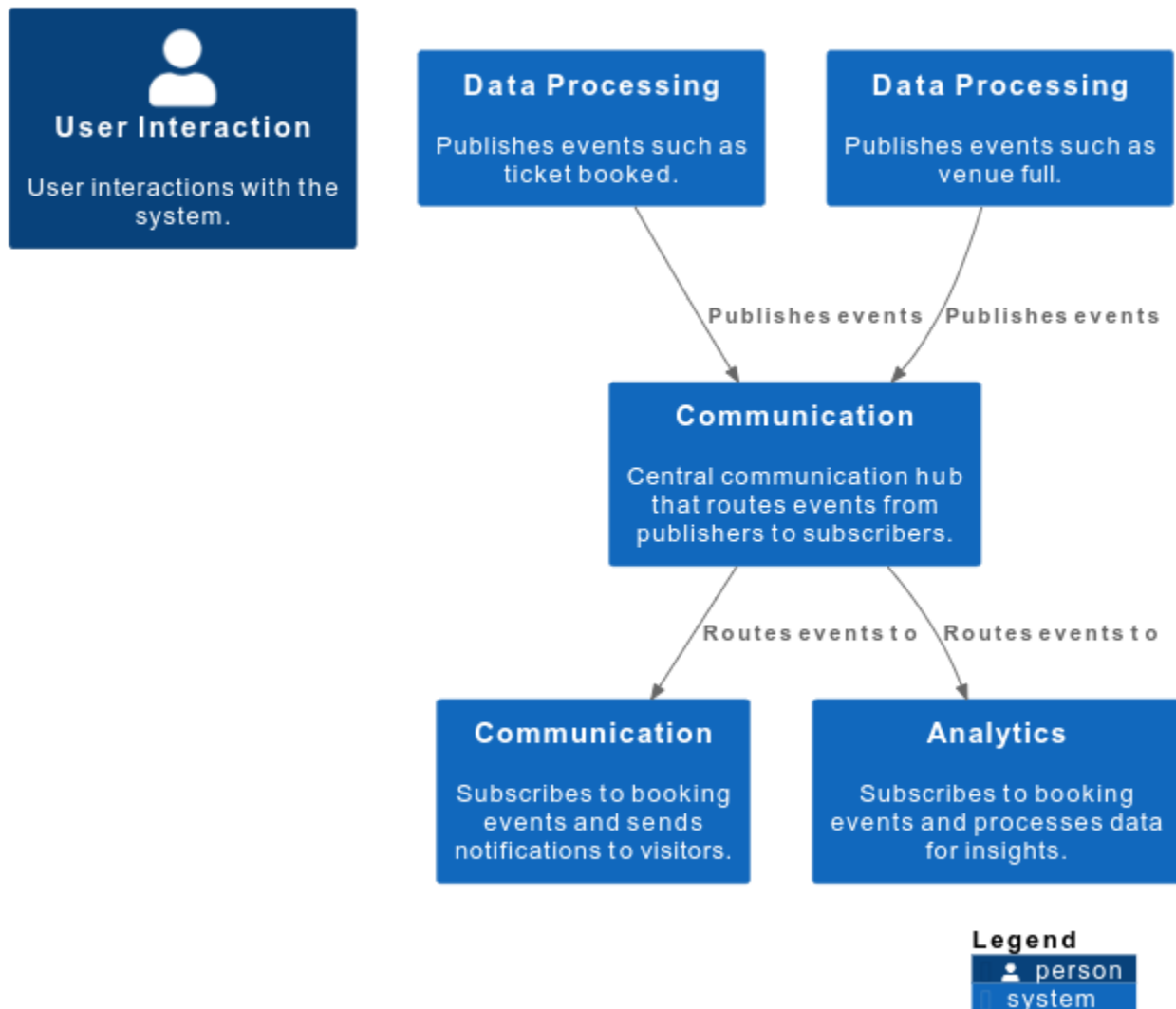
- **send_confirmation():** This function is part of the consumer script. It uses a `KafkaConsumer` instance to consume booking events from a Kafka topic named "booking_events." For each consumed event (in JSON format), the function sends a booking confirmation to the visitor by printing a message with visitor ID, event ID, and the number of tickets booked.
- **book_ticket():** This function is part of the producer script. It uses a `KafkaProducer` instance to produce booking events (in JSON format) to the "booking_events" Kafka topic. The function takes visitor ID, event ID, and the number of tickets as input arguments, creates a booking event message, and sends it to the Kafka topic. It then prints a confirmation message indicating that the booking was successful.

The key quality attributes demonstrated by this code are:

- **Scalability:** Using Kafka for asynchronous communication between the producer and consumer allows for high throughput and the ability to handle large volumes of booking events.
- **Reliability:** Kafka ensures reliable event delivery, providing at-least-once delivery semantics, which minimizes the risk of losing booking events.
- **Decoupling:** The producer and consumer are decoupled through the Kafka topic. This means the producer does not need to know how the consumer processes the events, and the consumer does not need to know how the events were produced.
- **Flexibility:** The architecture allows easy scalability and modification. New consumer or producer services can be added or existing ones modified without affecting other components, as long as they respect the established contracts (message schema).
- **Performance:** Kafka is known for its high performance in handling large volumes of messages and supporting real-time streaming of data.

13.1.2. High level Context Diagrams

Context Diagram: Publish-Subscribe Pattern



13.1.3. Implementation: Codebase

1. We installed Kafka by using the [this blog](#) as reference.
2. Started the kafka server and ensured that it is running successfully. See below
3. For more detail follow the Github Link.

```
kafka@vbhs-desktop: ~  
hanuma@vbhs-desktop:~$ su -l kafka  
Password:  
kafka@vbhs-desktop:~$ sudo systemctl start kafka  
[sudo] password for kafka:  
kafka@vbhs-desktop:~$ sudo systemctl status kafka  
● kafka.service  
   Loaded: loaded (/etc/systemd/system/kafka.service; disabled; vendor preset: enabled)  
   Active: active (running) since Thu 2024-04-11 18:18:21 IST; 22h ago  
     Main PID: 3818805 (sh)  
       Tasks: 85 (limit: 18730)  
      Memory: 976.5M  
         CPU: 6min 44.272s  
    CGroup: /system.slice/kafka.service  
            └─3818805 /bin/sh -c "/home/kafka/bin/kafka-server-start.sh /home/kafka/config/server.properties > /home/kafka/kafka.log 2>&1"  
            └─3818807 java -Xmx1G -Xms1G -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+ExplicitGCInvokesConcurrent -XX:MaxInlineLevel=15 -Djava.awt.headless=true "-Xlog  
Apr 11 18:18:21 vbhs-desktop systemd[1]: Started kafka.service.  
(lines 1-12/12 (END))
```

4. We have two directories with the names **publish_subscribe** and **service_oriented**. Inside the directory **“Assignment-3_Team-11_Code_Base”**.
5. Inside **“publish_subscribe”** directory there is two files with the names **“consumer.py”** and **“producer.py”**
6. As the names suggest these are two architectural patterns that we implemented. Each folder has its respective **producer.py** and consumer.py files.

```
kafka@vbhs-desktop:~$ cd bin  
kafka@vbhs-desktop:~/bin$ ls  
connect-distributed.sh  kafka-cluster.sh      kafka-delete-records.sh  kafka-log-dirs.sh      kafka-run-class.sh      kafka-verifiable-consumer.sh  zookeeper-server-start.sh  
connect-mirror-maker.sh  kafka-configs.sh      kafka-dump-log.sh        kafka-metadata-quorum.sh  kafka-server-start.sh    kafka-verifiable-producer.sh  zookeeper-server-stop.sh  
connect-plugin-path.sh   kafka-console-consumer.sh  kafka-e2e-latency.sh     kafka-metadata-shell.sh  kafka-server-stop.sh     publish_subscribe              zookeeper-shell.sh  
connect-standalone.sh    kafka-console-producer.sh  kafka-features.sh        kafka-mirror-maker.sh    kafka-storage.sh         service_oriented               trogdor.sh  
kafka-acls.sh            kafka-consumer-groups.sh  kafka-get-offsets.sh     kafka-producer-perf-test.sh  kafka-streams-application-reset.sh  windows                        zookeeper-security-migration.sh  
kafka-broker-api-versions.sh  kafka-consumer-perf-test.sh  kafka-jmx.sh             kafka-reassign-partitions.sh  kafka-topics.sh          zookeeper-security-migration.sh  
kafka-client-metrics.sh    kafka-delegation-tokens.sh  kafka-leader-election.sh  kafka-replica-verification.sh  kafka-transactions.sh  
kafka@vbhs-desktop:~/bin$ cd publish_subscribe/  
kafka@vbhs-desktop:~/bin/publish_subscribe$ ls  
consumer.py  producer.py  
kafka@vbhs-desktop:~/bin/publish_subscribe$ cd ..  
kafka@vbhs-desktop:~/bin$ cd service_oriented/  
kafka@vbhs-desktop:~/bin/service_oriented$ ls  
consumer.py  producer.py  
kafka@vbhs-desktop:~/bin/service_oriented$
```

Let us now look at the files and expected outputs. These files are also uploaded with this assignment in the same format.

producer.py:

```
Activities Terminal Fri Apr 12 17:09:03
kafka@vbhs-desktop: ~/bin/publish_subscribe

GNU nano 6.2 producer.py
from kafka import KafkaProducer
import json
import time
import sys

kafka_host = 'localhost:9092'
booking_topic = 'booking_events'

producer = KafkaProducer(bootstrap_servers=[kafka_host],
                        value_serializer=lambda x: json.dumps(x).encode('utf-8'))

def book_ticket(visitor_id, event_id, num_tickets):
    # Simulate booking logic
    booking_details = {
        'visitor_id': visitor_id,
        'event_id': event_id,
        'num_tickets': num_tickets,
        'timestamp': time.time()
    }
    producer.send(booking_topic, value=booking_details)
    print(f"Booking successful: {booking_details}")

if __name__ == "__main__":
    # Check if command-line arguments are provided correctly
    if len(sys.argv) != 4:
        print("Usage: python3 producer.py <visitor_id> <event_id> <num_tickets>")
        sys.exit(1)

    # Extract command-line arguments for booking
    visitor_id = sys.argv[1]
    event_id = sys.argv[2]
    num_tickets = int(sys.argv[3]) # Convert num_tickets to integer

    # Book tickets using provided input
    book_ticket(visitor_id, event_id, num_tickets)
```

consumer.py

```
Activities Terminal Fri Apr 12 17:23:13
kafka@vbhs-desktop: ~/bin/publish_subscribe
kafka@vbhs-desktop: ~/bin/publish_subscribe

GNU nano 6.2 consumer.py
from kafka import KafkaConsumer
import json

kafka_host = 'localhost:9092'
booking_topic = 'booking_events'

consumer = KafkaConsumer(booking_topic, bootstrap_servers=[kafka_host],
                        value_deserializer=lambda x: json.loads(x.decode('utf-8'))))

def send_confirmation():
    for message in consumer:
        booking_details = message.value
        visitor_id = booking_details['visitor_id']
        event_id = booking_details['event_id']
        num_tickets = booking_details['num_tickets']
        print(f"Sending confirmation to visitor {visitor_id} for event {event_id} ({num_tickets} tickets)")

if __name__ == "__main__":
    send_confirmation()
```

Steps to Run and the expected outputs

1. Start the kafka service using “**sudo systemctl start kafka**”
2. Run the consumer.py script to listen to the messages. Once we run the producer.py script it listens to the message and outputs the confirmation.

```
kafka@vbhs-desktop:~/bin/publish_subscribe$ python3 consumer.py
Sending confirmation to visitor 456 for event 123 (2 tickets)
```

3. Now in another tab, run producer.py to simulate ticket booking. Through the command line only pass visitor id, event id and number tickets, like shown below

```
kafka@vbhs-desktop:~/bin/publish_subscribe$ python3 producer.py 456 123 2
Booking successful: {'visitor_id': '456', 'event_id': '123', 'num_tickets': 2, 'timestamp': 1712924458.5802207}
kafka@vbhs-desktop:~/bin/publish_subscribe$
```

13.2. Service Oriented Pattern:

- This pattern promotes modularization and loose coupling by decomposing the system into independent services.
- In the booking system, various functionalities, such as ticket booking, venue capacity management, and payment processing, are implemented as separate services.
- Each service exposes a well-defined interface (API) for interaction with other components.
- For example, the ticket booking service handles booking requests from visitors, the venue capacity service manages available seats in event venues, and the payment service processes payment transactions.
- This decomposition allows for scalability, flexibility, and easier maintenance of the booking system.

13.2.1. Quality Attributes and Their Explanations

Flask Web Service:

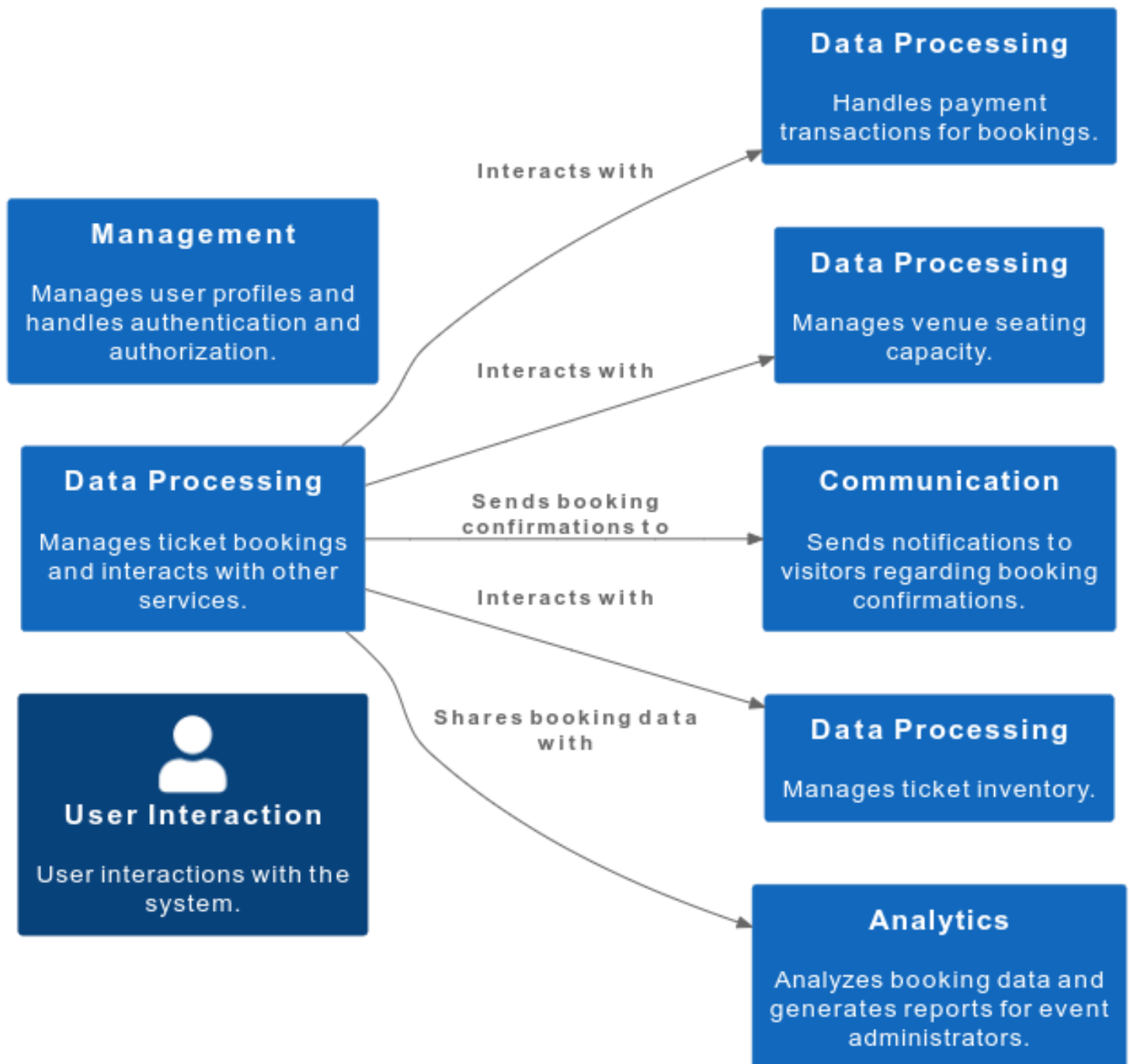
- The Flask application provides an API endpoint (/book_ticket) to handle booking requests sent via HTTP POST.
- It receives JSON-formatted booking requests, deserializes them, and sends them to a Kafka topic (booking_requests) using a KafkaProducer instance.
- Once the booking request is successfully sent to the Kafka topic, the service returns a JSON response to the client indicating success.



Kafka Consumer Script:

- This script listens to the booking_requests Kafka topic using a KafkaConsumer instance.
- When a new booking request message arrives, it deserializes the message and processes the booking request (e.g., printing booking request details for the event and number of tickets).

Key quality attributes demonstrated by this code:

- **Scalability:** The service-oriented architecture using Kafka enables horizontal scaling by allowing multiple instances of the API endpoint and processing script to handle different requests simultaneously.
- **Decoupling:** The use of Kafka for asynchronous communication between the API endpoint and processing script allows them to operate independently without tightly coupled dependencies.
- **Flexibility:** The architecture allows easy integration with other services. Additional services can be added without disrupting existing components.
- **Reliability:** Kafka's message queuing ensures that booking requests are reliably delivered from the API endpoint to the processing script.
- **Maintainability:** Flask and Kafka provide a high-level, readable codebase that is easy to maintain and update.

Context Diagram: Service-Oriented Pattern**Legend**

	person
	system

13.2.3. 13.2.3 Implementation: Codebase

producer.py

```
Activities Terminal Fri Apr 12 17:34:05
kafka@vbhs-desktop: ~/bin/service_oriented

GNU nano 6.2 producer.py
from flask import Flask, request, jsonify
from kafka import KafkaProducer
import json

app = Flask(__name__)
kafka_host = 'localhost:9092'
booking_topic = 'booking_requests'

producer = KafkaProducer(bootstrap_servers=[kafka_host],
                        value_serializer=lambda x: json.dumps(x).encode('utf-8'))

@app.route('/book_ticket', methods=['POST'])
def book_ticket():
    data = request.get_json()
    visitor_id = data['visitor_id']
    event_id = data['event_id']
    num_tickets = data['num_tickets']

    booking_details = {
        'visitor_id': visitor_id,
        'event_id': event_id,
        'num_tickets': num_tickets
    }
    producer.send(booking_topic, value=booking_details)
    return jsonify({'message': "Booking request sent successfully"}), 200

if __name__ == "__main__":
    app.run(port=5000)
```

consumer.py

```
Activities Terminal Fri Apr 12 17:34:25
kafka@vbhs-desktop: ~/bin/service_oriented

GNU nano 6.2 consumer.py
from kafka import KafkaConsumer
import json

kafka_host = 'localhost:9092'
booking_topic = 'booking_requests'

consumer = KafkaConsumer(booking_topic, bootstrap_servers=[kafka_host],
                        value_deserializer=lambda x: json.loads(x.decode('utf-8'))))

def process_booking_requests():
    for message in consumer:
        booking_details = message.value
        event_id = booking_details['event_id']
        num_tickets = booking_details['num_tickets']
        print(f"Processing booking request for event {event_id} ({num_tickets} tickets)")

if __name__ == "__main__":
    process_booking_requests()
```

Running steps:

1. Start Kafka Server (sudo systemctl start kafka).
2. Go to the folder service_oriented through terminal and run consumer.py script
3. It starts listening to messages.

```
kafka@vbhs-desktop:~/bin/service_oriented$ python3 consumer.py
Processing booking request for event 456 (2 tickets)
```

4. Now open another tab, start kafka service again.
5. Go to the folder through service_oriented through the terminal and run producer.py which runs the flask app

```
kafka@vbhs-desktop:~/bin/service_oriented$ python3 producer.py
* Serving Flask app 'producer'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [12/Apr/2024 17:30:12] "GET / HTTP/1.1" 404 -
127.0.0.1 - - [12/Apr/2024 17:31:52] "POST /book_ticket HTTP/1.1" 200 -
```

6. Now use CURL or POSTMAN to send http post request like below

```
curl -X POST -H "Content-Type: application/json" -d '{"visitor_id": "123",  
"event_id": "456", "num_tickets": 2}' http://localhost:5000/book_ticket
```

7. This command simulates a booking request being sent to your Flask app, which in turn publishes a message to the Kafka topic.

```
kafka@vbhs-desktop:~$ curl -X POST \  
-H "Content-Type: application/json" \  
-d '{"visitor_id": "123", "event_id": "456", "num_tickets": 2}' \  
http://localhost:5000/book_ticket \  
{ "message": "Booking request sent successfully" }  
kafka@vbhs-desktop:~$
```

14. A comprehensive analysis for Key quality metrics:

Quality Metrics	Publish Subscribe Pattern	Service Oriented Pattern
Throughput	The number of booking events that can be produced and consumed per second. Kafka's high throughput allows efficient handling of large volumes of booking events.	The number of booking requests processed by the system per second. Kafka's high throughput allows efficient handling of large volumes of booking requests.
Latency	The time it takes for a booking event to be processed (from when it is produced to when it is consumed). Kafka provides low-latency processing, which is beneficial for a real-time booking system.	The time taken for a booking request to be processed from the API endpoint to when it is consumed by the processing script.
Scalability	The ability of the booking system to handle increased load by scaling up (adding more producers or consumers) or scaling out (adding more Kafka brokers).	The ability of the system to handle increased load by scaling up (adding more instances of the Flask application and KafkaConsumer scripts) or scaling out (adding more Kafka brokers).

Availability	Kafka's distributed nature ensures high availability and fault tolerance, minimizing downtime and disruptions.	Kafka's distributed architecture ensures high availability and fault tolerance, providing continuous service even during hardware or network failures.
Reliability	The consistency and accuracy of booking events being produced and consumed. Kafka guarantees message delivery and consistency.	Kafka's message delivery guarantees ensure that booking requests are delivered and processed reliably.
Maintainability	How easy it is to maintain the booking system, add new features, or fix bugs. The use of Kafka and modular functions makes the system more maintainable.	The modular, service-oriented architecture simplifies maintenance, allowing for easier updates and changes to different parts of the system.

15. A comprehensive analysis of which pattern is better! and why?

The choice of pattern that performs better depends on the specific requirements and goals of the system. In the given scenario, both the Publish-Subscribe Pattern and the Service-Oriented Pattern have their advantages. Let's discuss the two patterns and their benefits:

Publish-Subscribe Pattern:

Pros:

- **Decoupling:** Publishers and subscribers are decoupled, enabling independent development and scalability.
- **Asynchronous Communication:** Event-driven architecture allows for flexible and efficient processing of events without waiting for responses.
- **Scalability:** Adding more publishers or subscribers does not affect existing components.
- **Real-Time Notifications:** Subscribers can receive real-time notifications for events, making it ideal for systems that require timely updates.

Cons:

- **Complexity:** The pattern can introduce complexity in managing event flows and handling event filtering and routing.
- **Debugging:** Troubleshooting issues in asynchronous, event-driven systems can be challenging due to indirect interactions.

Service-Oriented Pattern:

Pros:

- **Loose Coupling:** Services are loosely coupled, enabling modular design and independent development.
- **Interoperability:** Services can communicate using standard protocols, making it easier to integrate with other systems.
- **Maintainability:** Services are designed around business functions, making them easier to understand and maintain.
- **Scalability:** Services can be scaled independently based on demand.

Cons:

- **Latency:** The use of service-to-service communication can introduce additional network latency.
- **Overhead:** Service-oriented systems may have overhead related to network communication and data serialization.

16. Conclusion:

Both patterns offer significant benefits for the given system. However, based on the scenario's requirements, the Service-Oriented Pattern is better in my choice for the following reasons:

- **Flexibility and Maintainability:** The Service-Oriented Pattern allows for modular design and independent development of services, making it easier to maintain and update the system as requirements change.
- **Interoperability:** The pattern supports standard protocols and can easily integrate with other systems and services.
- **Scalability:** Each service can be independently scaled based on demand, allowing the system to handle high traffic effectively.

Overall, the Service-Oriented Pattern aligns well with the goal of providing a flexible, maintainable, and scalable booking system for the Univaq Street Science event.