# Hotel Booking System - Project Documentation

**Contributors**: Amit Bhujbal, Soham, Chirag Bheemaiah Palanganda Karumbaiah

# 1 Introduction

## 1.1 Motivation

We intend to develop a highly scalable and highly available Hotel Booking Platform. Customers can choose from a variety of hotels, comparing between them for the best price. They can choose their dates of stay, and manage their reservations with the hotels through the platform.

During the holiday season, finding and booking hotels can become a frustrating experience due to high demand, limited availability, and fluctuating prices. Customers often struggle to compare options efficiently, leading to missed deals or overbooked hotels. Additionally, managing reservations across different platforms adds to the complexity, causing inconvenience and dissatisfaction. Recognizing these challenges, we aim to develop a highly scalable and available hotel booking platform that simplifies the process.

## 1.2 Executive summary

The hotel booking process, particularly during peak travel seasons, often presents significant obstacles due to limited availability and inconsistent user experiences across platforms. To address these issues, we are developing a cloud-based hotel booking platform that prioritizes scalability and availability. The system is designed to handle high traffic volumes while maintaining seamless performance, ensuring a smooth user experience even during times of peak demand. By leveraging cloud technologies, our solution guarantees a robust infrastructure capable of efficiently managing customer reservations and hotel data, ultimately providing a reliable and convenient platform for users. This document explores the key design and implementation considerations that have shaped the development of this solution.

# 2 Problem Description

## 2.1 The problem

The hotel booking platform must support various user interfaces, including mobile applications, web browsers, and potentially third-party platforms like travel agencies. Users can access the platform from anywhere in the world, requiring the infrastructure to be globally accessible and reliable. Given that hotel bookings often involve time-sensitive decisions, such as securing accommodations during peak seasons, it is crucial that the platform can process requests swiftly with minimal latency. Additionally, as users interact with the system through multiple devices and communication methods, the infrastructure must be able to handle a large volume of concurrent requests without compromising availability or performance. Since the platform will handle sensitive user information, such as payment details and personal data, it is imperative that the system maintains the highest standards of security to prevent unauthorized access. To meet these requirements, we are leveraging Platform-as-a-Service (PaaS) to ensure scalability, availability, and security, while minimizing the complexities of infrastructure management. The PaaS solution will allow us to focus on building features and services while relying on cloud providers to handle resource provisioning, load balancing, and security at scale.

## 2.2 Business Requirements

**BR1**: The system should be highly scalable to accommodate a growing user base and peak travel periods.

**BR2**: The system must be highly available to cater to the users 24x7.

**BR3**: The application should be highly secure and compliant with regional specifications.

**BR4**: Optimize infrastructure costs.

**BR5**: Ensure low latency for booking transactions.

**BR6**: Implement robust disaster recovery and backup strategies.

**BR7**: Deploy comprehensive monitoring and logging tools to track system performance, errors, and security incidents.

**BR8**: Ensure error-free Identity and Access Management for employees and users.

**BR9**: The system must track detailed usage metrics, mapping users to specific resources they consume. It should generate transparent, itemized bills for accurate cost allocation.

**BR10**: Provide real-time data processing capabilities from different information sources.

**BR11**: Data storage solutions must be optimized for high availability, redundancy, and rapid access.

**BR12**: Transactions must handle concurrent requests seamlessly, without leading to inconsistencies.

**BR13**: The architecture must have flexibility to adapt to changing user requirements.

**BR14**: Automate workflows to enhance operational efficiency.

**BR15**: The test suite must cover all critical functionalities, ensuring the system’s integrity and stability through automated and manual testing practices.

**BR16**: Optimize the system’s operations and infrastructure to reduce carbon emissions.

**BR17**: Enable a multi-tenant architecture, ensuring that each tenant has isolated access to their own data and operations while sharing the overall infrastructure.

## 2.3 Technical Requirements

**BR 1**: The system should be highly scalable to accommodate a growing user base and peak travel period.

**TR 1.1**: The infrastructure should deploy instances including load balancers across multiple geographically distributed data centers to ensure fault tolerance, with a minimum of two availability zones per region.

**TR 1.2**: The system should automatically scale to accommodate at least 5000 requests per second during peak loads, employing scaling policies.

**TR 1.3**: The system should dynamically allocate and release resources based on demand patterns, employing scaling policies that adjust infrastructure according to traffic trends every 5 minutes**.**

**TR 1.4**: Real-time monitoring should continuously assess the resource utilization rate for each component, triggering horizontal or vertical scaling when resource use exceeds 70% for over 5 minutes.

**TR 1.5**: The system should automatically replace any degraded or unresponsive instance with a healthy one, ensuring zero interruption to end users.

***Justification*:** A booking system often experiences high demand during peak travel times or promotions. Automatic scaling helps prevent slowdowns or crashes, maintaining a smooth booking experience. Deploying in multiple clusters will help in distributing the load evenly to reduce stress during peak loads. During the peak loads, as mentioned in section 2.1, we are expecting millions of users to visit our site for bookings. So, the requirement we are designing is 5000RPS.

**BR 2**: The system must be highly available to cater to the users 24x7.

**TR 2.1**: The system should achieve a service availability level of 99.999% per year through automated failover and redundancy mechanisms.

**TR 2.2**: A global request routing mechanism should be used to direct user requests to the nearest data center based on geographic proximity, reducing latency and improving performance.

**TR 2.3**: The system should configure multiple IP addresses with each domain name (DNS) entry to allow for failover, ensuring continued availability in case of network disruption.

***Justification*:** Ensuring continuous availability is critical to avoid losing booking transactions and to maintain user trust in the platform, especially as users expect round-the-clock access. Deploying across availability zones will help avoid datacenter wide failures such as natural disasters. Maintaining redundant resources in the architectures ensures high availability by eliminating single point of failures.

**BR3**: The application should be highly secure and compliant with regional specifications.

**TR 3.1**: All data stored at rest should be encrypted with an AES-256 or higher encryption standard.

**TR 3.2**: All data in transit should be encrypted using TLS 1.2 or higher between all components and communication channels.

**TR 3.3**: Encryption keys should be stored securely using a separate and isolated key management service with strict access control.

**TR 3.4**: The system should implement a continuous auditing and logging solution for all user actions, maintaining real-time traceability for compliance.

**TR 3.5**: Automated patch management should ensure that all software components are updated with the latest security patches within 7 days of release.

**TR 3.6**: A Web Application Firewall (WAF) should be implemented to inspect and filter incoming requests, detecting and mitigating potential threats.

***Justification*:** Booking systems handle sensitive user data, including payment details, making robust encryption essential for safeguarding data both in transit and at rest. This protects user privacy and ensures compliance with security regulations. With a globally distributed architecture, varying regional compliance requirements must be addressed. Threats can arise from external attacks, such as DDoS and other cyberattacks, or from security vulnerabilities in the application code or underlying infrastructure software. These risks can be mitigated by implementing firewalls with rate-limiting capabilities and IP blacklisting options. Additionally, automated patch management is crucial for addressing software vulnerabilities in critical infrastructure dependencies.

**BR4**: Optimize infrastructure costs.

**TR 4.1**: Resource usage metrics such as CPU, memory utilization should be regularly evaluated to identify inefficiencies and reduce under-utilized resources by scaling down or reallocating capacity.

**TR 4.2**: The system should implement scheduled archiving of inactive data every 90 days to long-term storage solutions optimized for archival, reducing storage costs.

**TR4.3:** Place workloads in regions with lower operational costs.

**TR4.4:** Consolidate workloads using containers to maximize resource utilization.

**TR 4.5**: The system should support tenant-specific monitoring and reporting, with the ability to track monthly usage metrics for each client.

**TR4.6:** The system should prioritize using private networks for inter service communication to reduce costs

***Justification*:** For a booking system, compute, storage, and network are the main contributors to costs. Optimizing compute resources by using scalable solutions, such as auto-scaling or serverless architectures, ensures efficient use of capacity while minimizing expenses. Data can be classified based on access patterns, with frequently accessed data stored in high-performance storage, and inactive data archived to cost-effective long-term storage solutions after a set period. Placing workloads in regions with lower operational costs, while adhering to compliance requirements, helps reduce costs. Additionally, using containerization to consolidate workloads to specific users maximizes resource utilization, further cutting down on compute costs. Utilizing private networks ensures lower costs for inter-service communication

**BR5**: Ensure low latency for booking transactions.

**TR 5.1**: The system should deploy instances across multiple global regions to reduce latency, with automated routing directing requests to the region with the lowest response time.

**TR5.2:** The system should employ CDN’s to cache frequently accessed large data files across geographically dispersed fronts to ensure quick response times.

**TR5.3:**  Use dedicated private links to bypass public internet congestion and reduce latency for critical services.

***Justification*:** Low latency for booking transactions can be affected by several factors including geographic distance, network congestion, and resource availability.Deploying instances across global regions, automated routing, CDNs for caching, and dedicated private links help reduce latency by optimizing data access, minimizing travel time, and bypassing congestion, ensuring fast and reliable booking transactions.

**BR6**: Implement robust disaster recovery and backup strategies.

**TR 6.1**: The system should store continuous snapshots of data, replicating these snapshots across multiple zones and rotating them to ensure recoverability in case of a disaster.

**TR6.2:** The system should perform regular testing of failover mechanisms and recovery plans including simulating disasters to ensure the process is well-defined and working.

***Justification*:** Continuous snapshots of data, replicated across multiple zones, ensure that reliable backups are always available for quick recovery during disasters. Regular testing of failover mechanisms and recovery plans, including disaster simulations, ensures that the recovery process is well-defined and effective, minimizing downtime and ensuring business continuity.

**BR7**: Deploy comprehensive monitoring and logging tools to track system costs, performance, errors, and security incidents.

**TR 7.1**: The system should provide real-time performance metrics for each service component, including CPU usage, memory utilization, network throughput, and disk I/O, with a minimum refresh interval of one minute.

**TR 7.2**: The system should have pre-defined alerting thresholds for each resource type to notify relevant teams when usage reaches certain percentages.

**TR 7.3:** The system should utilize tools to track application-level performance, user interactions, and transaction flows to detect slowdowns, errors, and anomalies.

***Justification*:** Performance issues can arise due to infrastructure failures or application code inefficiencies. By implementing real-time performance metrics, pre-defined alerting thresholds, and tracking application-level performance, the system can quickly detect and address these issues.

**BR8**: Ensure error-free Identity and Access Management for employees and users.

**TR 8.1**: Each component of the system should have clearly defined identity and access permissions, limiting access to only authorized users or processes.

**TR 8.2**: The system should enforce least-privilege access, ensuring that users and processes are granted only the permissions necessary to complete their assigned tasks.

**TR 8.3**: A periodic access review process should be implemented to ensure that permissions remain aligned with user roles, revoking unused or unnecessary privileges every quarter.

***Justification*:** The defined access control measures ensure that only authorized users or processes can access system components, which is crucial for maintaining security in a multi-tenant system. By enforcing least-privilege access, the system limits the potential impact of unauthorized access, ensuring users and processes are granted only the permissions necessary for their tasks. The periodic review of access rights ensures that permissions stay aligned with user roles, with unused or unnecessary privileges being revoked, reducing the risk of privilege escalation.

**BR9:** The system must track detailed usage metrics, mapping users to specific resources they consume. It should generate transparent, itemized bills for accurate cost allocation.

**TR 9.1**: The system should implement tenant-level usage tracking for each infrastructure component, with detailed usage metrics for resource consumption, supporting accurate, usage-based billing and reporting.

**TR 9.2:** The frequency of metric collected should be modeled according to the workload being processed by the system.  
***Justification*:** We expect our application to experience a generally consistent workload, with occasional spikes during peak travel periods. Implementing a workload-aware data collection process would ensure accurate fault detection and system reliability during high-demand scenarios.

**BR10**: Provide real-time data processing capabilities from different information sources.

**TR 10.1**: The system should maintain standardized data formats across components to ensure compatibility and avoid proprietary lock-in.

**TR 10.2:** The system should implement such workloads on specific high performance systems provisioned from the cloud provider.

**Justification:** Different API’s are processed to provide the customer with the most optimal prices for their booking plans. Problems would arise if every service were to convert the data into its required specification and output it in a different format. HPC’s should be utilized for maintaining low latencies and reducing costs.

**BR11**: Data storage solutions must be optimized for high availability, redundancy, and rapid access.

**TR 11.1**: The system should prioritize business-critical data, such as user transactions, bookings, and payment details, and must be replicated across at least two or more Availability Zones (AZs) to ensure fault tolerance, availability, and durability.

**TR 11.2:** The system should classify non-critical data such as temporary session data, marketing campaigns, and logs, and should be archived into cost-effective storage solutions. These datasets should be scheduled for automated deletion after a specific retention period of 30 days.

**TR 11.3:** To improve response times globally, the system must implement distributed caching mechanisms for frequently accessed data, such as user session details, hotel availability, and pricing information.

**Justification:** Our application handles diverse data types, necessitating classification based on importance and access frequency. This approach enables the selection of appropriate storage solutions tailored to workload requirements, ensuring optimal performance, cost-efficiency, and reliability.

**BR12**: Transactions must handle concurrent requests seamlessly, without leading to inconsistencies.

**TR 12.1:** The system must ensure data integrity during transactions by supporting the principles of atomicity, consistency, isolation, and durability (ACID).

**TR 12.2:** Row-level or table-level locking should be a feature of the system to prevent conflicts and inconsistencies during simultaneous data access.

**TR 12.3:** The system should usea default isolation level to balance performance and consistency while avoiding common issues like dirty reads.

**TR 12.4:**The system should introduce a basic retry strategy for failed transactions, ensuring idempotent operations to prevent duplicate data changes.

**Justification:** To support a global user base, a distributed architecture that scales independently is essential. However, maintaining consistency across distributed systems can be challenging. This can be addressed by implementing ACID properties, row or table-level locking, preventing dirty reads, and using a retry strategy for failed transactions to ensure data integrity and prevent issues like duplicate operations.

**BR 13**: The architecture must have flexibility to adapt to changing user requirements.

**TR 13.1:** The platform should adopt a microservices-based architecture to ensure each service is loosely coupled and can be developed, deployed, and scaled independently.

**TR13.2:** The system should implement comprehensive dependency injection and interface-based design patterns to enable easy modification and extension of system components.

**TR13.3:** The architecture must support dynamic configuration management and feature toggles to allow runtime changes without system redeployment.

**Justification:** For a hotel booking system, requirements frequently change due to seasonal demands, varying rates, and shifting business rules. The combination of microservices enables scalability during peak seasons, interface-based design supports quick addition of booking channels, dynamic configuration allows instant rate changes, and flexible data models accommodate special booking requirements.

**BR 14**: Automate workflows to enhance operational efficiency

**TR 14.1**: The system must implement automated scaling policies based on predefined metrics to handle varying booking loads efficiently.

**TR 14.2**: The architecture should incorporate serverless functions for automated background tasks like email notifications, payment processing, and inventory updates.

**TR 14.3:** The architecture must implement automated CI/CD pipelines to ensure rapid, reliable deployment of booking system updates and features.

**Justification:** For a hotel booking system, operational efficiency is crucial to handle varying booking volumes and frequent updates. Automated scaling optimizes resource use during peak periods, serverless functions manage background tasks autonomously, and CI/CD automation ensures smooth updates. Together, these TRs create an efficient cloud system, maximizing performance and minimizing manual intervention.

**BR15**: The test suite must cover all critical functionalities, ensuring the system’s integrity and stability through automated and manual testing practices.

**TR 15.1**: The system must implement comprehensive automated testing including unit tests, integration tests, and end-to-end tests covering critical booking flows and payment processing.

**TR 15.2**: The system should utilize continuous integration/continuous deployment (CI/CD) pipelines with automated test execution and quality gates for all deployments.

**TR 15.3**: The platform must support automated performance testing and load testing capabilities to simulate peak booking periods and concurrent user scenarios.

**TR15.4**: The system must implement automated monitoring and alerting for test environments, with detailed logging and error reporting mechanisms.

**Justification:** Ensuring error-free reservations and payment processing is vital for a hotel booking system. Automated testing of core booking flows, consistent deployment quality, performance validation during peak seasons, and proactive issue detection collectively ensure the system remains reliable during high-demand periods and critical transactions.

**BR 16**: Optimize the system’s operations and infrastructure to reduce carbon emissions

**TR 16.1**:The system should be hosted on cloud providers that utilize 100% renewable energy and implement energy-efficient hardware, with proven carbon-neutral operations.

**TR 16.2:** The system must integrate carbon footprint monitoring tools to track and report energy consumption and emissions, providing transparency and optimization insights for sustainability.

**TR 16.3:** The system should use serverless computing and elastic scaling to automatically adjust resources based on demand, reducing energy consumption by avoiding idle resources.

**Justification:** Optimizing for energy-efficient infrastructure reduces operational costs and supports sustainability. Serverless and scalable architecture ensures efficient resource use during peak periods, cutting waste. Carbon footprint monitoring provides actionable insights for continuous optimization, while a green cloud provider reinforces the commitment to sustainability, aligning with business and environmental goals.

**BR 17**: Enable a multi-tenant architecture, ensuring that each tenant has isolated access to their own data and operations while sharing the overall infrastructure.

**TR 17.1**: The system must implement tenant isolation using virtualization techniques such as containers or virtual machines (VMs) to ensure that each tenant’s data and operations are isolated from others while sharing the same infrastructure.

**TR 17.2:** The system should implement role-based access control (RBAC) to ensure that each tenant has access only to their own data and operations, preventing unauthorized access to other tenants' resources.

**TR 17.3**: The system must provide a shared infrastructure with isolated resources for each tenant, ensuring that tenants can scale independently without impacting the performance or security of others.  
***Justification:*** In a hotel booking system, tenant isolation and RBAC are essential for secure data management and access control, safeguarding customer privacy and security. The ability to independently scale tenant infrastructure ensures personalized services for each customer without impacting the performance of other users.

## 2.4 Tradeoffs

### **TR 1.1 (Fault Tolerance) vs. TR 16.3 (Energy Efficiency)**

When deploying infrastructure across multiple geographically distributed data centers, we ensure fault tolerance by maintaining operations during regional failures, which is crucial for handling high booking demands worldwide. However, we’ve noticed that this approach increases energy consumption due to the higher power requirements for servers and cooling systems, which doesn’t align with energy-efficient practices. For us, this creates a tradeoff—while fault tolerance is essential to guarantee smooth booking experiences, especially during peak times, it comes at the cost of higher energy usage, which can impact the efficiency of real-time booking confirmations.

### **TR 2.1 (99.999% Availability) vs. TR 3.5 (Patch Management within 7 Days)**

Achieving 99.999% availability is essential to ensure uninterrupted service during peak booking periods, especially for reservations with strict deadlines. However, applying patches within seven days may require system downtime or performance degradation, posing a risk to availability. This tradeoff affects workloads related to payment processing, where downtime can lead to transaction failures or customer dissatisfaction.

### **TR 5.1 (Latency Reduction) vs. TR 1.3 (Dynamic Scaling)**

Reducing latency through global instance deployment ensures faster responses for search and booking queries from users in different regions. However, dynamic scaling during demand surges (e.g., during holiday sales) may temporarily increase response times as new resources are provisioned. This tradeoff affects search and recommendation engine workloads, where real-time responsiveness is critical.

### **TR 3.6 (WAF for Threat Detection) vs. TR 5.1 (Latency Reduction)**

Using a Web Application Firewall (WAF) enhances security by detecting and mitigating threats like SQL injections and DDoS attacks. However, it introduces latency, conflicting with the goal of responding to requests as fast as possible. This tradeoff impacts workloads related to user account management, where secure transactions must be balanced against performance expectations.

### **TR 6.1 (Continuous Snapshots for Disaster Recovery) vs. TR 4.1 (Cost Optimization)**

Continuous snapshots provide robust disaster recovery by ensuring recent backups of booking and payment data. However, they increase storage costs, conflicting with cost optimization objectives. This tradeoff affects workloads related to database management, where maintaining the integrity of transactional data is critical but expensive.

### **TR 9.3 (Periodic Access Review) vs. TR 18.1 (Automation of Routine Tasks)**

Periodic access reviews ensure compliance with security policies by requiring manual oversight of user permissions, which can prevent unauthorized access to sensitive data. However, this conflicts with the goal of automating routine tasks to minimize manual intervention. In security sensitive applications, relying on manual validation of permissions is more fail safe than an automated approach.

# 3 Provider Selection

## 3.1 Criteria for Choosing a Provider

**C1 - Global Data Centers and Fault Tolerance:**

Deploying instances across multiple geographically distributed data centers ensures high fault tolerance by mitigating the impact of regional failures. The requirement for a minimum of two availability zones per region ensures resilience and uninterrupted service, especially during natural disasters or outages.

**Impact on Workloads:** This criterion ensures consistent uptime for critical reservation systems and real-time search queries, allowing customers to access booking services reliably regardless of location.

**(TRs:** **1.1, 5.1, 2.2, 6.1)**

**C2 - Guaranteed Uptime with SLAs and Automated Failover:**

High-availability SLAs (99.999% uptime) and automated failover prevent service interruptions by enabling seamless transitions to redundant systems in case of failures. Regional compliance considerations ensure legal and operational reliability across diverse markets.

**Impact on Workloads:** This guarantees continuity for critical workloads like payment processing, user authentication, and confirmation notifications, where downtime could lead to financial losses and customer dissatisfaction.

**(TRs: 2.1, 2.3, 1.2, 1.5, 6.1, 6.2, 7.1)**

**C3 - Dynamic Scaling and Demand-Based Resource Allocation:**Dynamic scaling accommodates traffic peaks by adjusting resources in real-time, ensuring smooth operation during sudden spikes in demand (e.g., holiday seasons). The ability to handle at least 10,000 requests per second with adjustments every 5 minutes supports time-sensitive operations.

**Impact on Workloads:** This is crucial for handling large volumes of booking requests and inventory updates, ensuring no degradation in performance during peak usage.

**(TRs: 1.2, 1.3, 1.4, 1.5, 4.1, 4.4, 17.1)**

**C4 - State-of-the-Art Hardware for Compute Services:**Modern hardware provides maximum computational performance, while resource utilization monitoring ensures efficient scaling. High-performance systems reduce latency and optimize resource costs.

**Impact on Workloads:** Compute-intensive tasks such as search algorithms, recommendation systems, and availability checks benefit from faster processing, enhancing the user experience.

**(TRs: 10.2, 16.1)**

**C5 - Support for Activities Related to Software Development Life Cycle:**  
Comprehensive support for the Software Development Life Cycle (SDLC) ensures efficient design, development, testing, deployment, and maintenance of applications. Features such as integrated development tools, CI/CD pipelines, testing frameworks, and version control systems streamline workflows and enhance productivity.

**Impact on Workloads:**  
Workloads involving iterative development, frequent updates, and rapid deployment—such as implementing new booking features, optimizing payment processes, or rolling out seasonal promotions—benefit from streamlined SDLC support, improving time-to-market and maintaining system reliability.

**(TR’s: 13.1, 13.2, 13.3, 14.1, 14.2, 14.3, 15.1, 15.2, 15.3, 15.4)**

**C6 - Comprehensive Observability and Monitoring Services:**Real-time monitoring with detailed performance metrics (CPU, memory, network) helps optimize performance and manage costs through proactive alerting and troubleshooting.

**Impact on Workloads:** Monitoring ensures stable operation of APIs, database queries, and user sessions by identifying bottlenecks and enabling rapid resolution of performance issues.

**(TRs: 1.4, 3.4, 4.1, 4.5, 7.1, 7.3, 8.1, 9.1, 9.2, 11.1, 11.2, 16.2, 17.1, 17.2)**

**C7 - Encryption at Rest and In Transit with Key Management:**Strong encryption (AES-256 at rest and TLS 1.2+ in transit) protects sensitive customer and transaction data. Secure key management adds another layer of security, ensuring data integrity.

**Impact on Workloads:** This criterion secures personal and payment data in workloads like reservations, user profiles, and financial transactions, maintaining customer trust and regulatory compliance.  
**(TRs:** **3.1, 3.2, 3.3)**

**C8 - Secure and Cost-Effective Archival Solutions:**Scheduled archiving of inactive data in encrypted, cost-optimized storage solutions enables compliance with data retention policies while controlling costs.

**Impact on Workloads:** Historical data such as past bookings, audit logs, and transaction records are securely stored for compliance and analytics, freeing up active resources for real-time operations.  
**(TRs: 11.2, 4.2, 13.1)**

**C9 - Identity and Access Management (IAM) with Least Privilege Access:**Enforcing least privilege access ensures only authorized personnel can access sensitive resources, reducing security risks. Role-based permissions further enhance operational control.

**Impact on Workloads:** IAM protects administrative interfaces, partner portals, and backend systems, minimizing risks of unauthorized access to booking databases or financial records.  
**(TRs: 8.1, 8.2, 8.3, 9.1, 17.2, 17.3)**

**C10 - Compliance and Continuous Monitoring for Security:**Continuous auditing, logging, and automatic patch management maintain infrastructure security and ensure adherence to global compliance standards.

**Impact on Workloads:** Security-critical workloads like fraud detection, regulatory reporting, and user data management are protected from vulnerabilities, ensuring long-term operational stability.  
(**TRs: 3.4, 3.5, 7.3, 8.3, 9.1, )**

**C11 - Support for Tenant-Specific Tracking and Usage Metrics:**Isolated dashboards for multi-tenant environments allow tracking of metrics like transaction volumes, user-specific activity, and monthly usage for accurate billing and performance insights.

**Impact on Workloads:** This enables precise monitoring of users, ensuring transparent billing and providing actionable insights into booking performance and user behavior.  
(**TRs: 17.1, 17.2, 17.3)**

**C12 - Environmental Impact:**The cloud provider should prioritize sustainable practices, including energy-efficient data centers, carbon-neutral operations, and optimized resource utilization to minimize the environmental footprint.

**Impact on Workloads:**  
Energy-efficient infrastructure reduces operational costs for non-critical processes such as data archival, log storage, and periodic backups while supporting eco-friendly practices. For critical workloads like real-time booking and inventory updates, sustainable operations maintain performance without compromising environmental goals.  
**(TRs: 16.1, 16.2, 16.3)**

## 3.2 Provider Comparison

Through our extensive research and analysis, we found that Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) emerged as the leading cloud providers, recognized for their robust features and comprehensive offerings. We selected these three platforms for comparison due to their industry dominance, diverse service portfolios, and proven track records in delivering scalable, secure, and cost-effective cloud solutions. Our evaluation focused on identifying the strengths and capabilities of each provider to determine their suitability for various cloud-based applications and workloads.

**\*[Yes/No] (Ranking)**

| **Criteria** | **Provided by AWS** | **Provided by Azure** | **Provided by GCP** |
| --- | --- | --- | --- |
| C1 | Yes (1) | Yes (1) | Yes (1) |
| C2 | Yes (1) | Yes (1) | Yes (1) |
| C3 | Yes (1) | Yes (1) | Yes (1) |
| C4 | Yes (1) | Yes (1) | Yes (1) |
| C5 | Yes (1) | Yes (1) | Yes (1) |
| C6 | Yes (1) | Yes (1) | Yes (1) |
| C7 | Yes (1) | Yes (1) | Yes (1) |
| C8 | Yes (1) | Yes (1) | Yes (1) |
| C9 | Yes (1) | Yes (1) | Yes (1) |
| C10 | Yes (1) | Yes (1) | Yes (1) |
| C11 | Yes (1) | Yes (1) | Yes (1) |
| C12 | Yes (1) | Yes (1) | Yes (1) |

## 3.3 The Final Selection

Each provider (AWS, Azure, and GCP) [16] fulfills all 12 criterias in this analysis. Let us perform a deeper analysis to rank amongst these service providers.

For a hotel booking system, a global presence is crucial to meet the diverse needs of users across different regions.It is essential to provision infrastructure at geographically distributed locations with each region complying with local regulations for data storage and handling, ensuring that the system adheres to legal requirements specific to the region.

Our system's operations involve a wide variety of tasks, such as processing user requests, handling bookings, and managing data analytics. These tasks can be categorized into different workloads, each with its own set of characteristics. For example, some workloads may require high-performance computing for real-time bookings, while others may focus on data storage or analytics. Each of these workloads has unique demands in terms of compute power, storage, and network resources.

### **Ranking**

1. **AWS**
2. **Azure**
3. **GCP**

AWS, being the largest cloud provider out of the Big Three, is well-positioned to support such a complex system. Its global infrastructure, which spans multiple regions and availability zones, allows for the deployment of servers in locations that are geographically close to the users. This ensures low-latency access and high availability, critical factors for the success of a hotel booking system. AWS also provides a whole host of configurable servers with different pricing models suitable for time varying workloads. Moreover, AWS can leverage the principle of **Massive Economies of Scale** [28]. By operating at a large scale, AWS can provide a wide variety of configurable servers at competitive prices.

Azure is a close competitor, especially for organizations already using Microsoft products. Its tight integration with Microsoft tools, hybrid cloud solutions, and strong security features make it a top choice for enterprises.

GCP excels in data-driven applications and machine learning, with powerful tools for big data and AI. However, it is still expanding its offerings in some enterprise services, particularly for traditional IT management and networking. [29]

## 3.3.1 The List of Services Offered by the Winner

**1. Amazon EC2 (Elastic Compute Cloud)[13]**: Amazon EC2 provides scalable compute capacity in the cloud. Launch and manage virtual machines, known as instances, with flexible configurations and pricing models. EC2 instances can be scaled based on demand, ensuring that the system can handle varying loads by adding or removing instances automatically.

**2. Amazon S3 (Simple Storage Service)**: Amazon S3[11] is a highly scalable and durable object storage service. It is ideal for storing static data such as backups, images, and logs. S3 automatically handles data replication across multiple availability zones, ensuring durability and high availability. It can also be configured for lifecycle policies, enabling efficient data archival and cost optimization.

**3. Amazon RDS (Relational Database Service)**: Amazon RDS simplifies database management by automating tasks like backups, patch management, and scaling. It supports multiple database engines such as MySQL, PostgreSQL, and Oracle. RDS provides high availability with Multi-AZ deployments, ensuring that data remains accessible even during failovers.

**4. Amazon CloudWatch**: Amazon CloudWatch[4] provides monitoring and observability for AWS resources and applications. It collects and tracks metrics like CPU usage, memory, and disk I/O, and allows you to set alarms to trigger actions, such as scaling resources or sending notifications when thresholds are crossed.

**5. Amazon Route 53**: [9]Amazon Route 53 is a highly available and scalable global Domain Name System (DNS) web service. It routes end-user requests to the nearest data center or region based on geographic proximity, reducing latency. Route 53 also supports failover routing, directing traffic to healthy endpoints if the primary endpoint becomes unavailable.

**6. AWS Elastic Load Balancing (ELB)[6]**: Elastic Load Balancing distributes incoming application traffic across multiple targets such as EC2 instances, containers, and IP addresses. It ensures that no single instance is overwhelmed with traffic, providing fault tolerance and improving application availability.

**7. AWS Identity and Access Management (IAM)**: AWS IAM[12] allows you to control access to AWS resources securely. You can create and manage AWS users and groups and assign permissions to allow or deny access to specific services or actions, ensuring that only authorized users have access to sensitive data and resources.

**8. AWS CloudFront**: AWS CloudFront is a content delivery network (CDN) that caches static and dynamic content in edge locations worldwide. It improves application performance by reducing latency and accelerating delivery of content to end-users across the globe.

# 4 The First Design Draft

# 4.0 Finalized List of TRs

**Tenant Identification**

* TR 17.1: Isolate tenant data

**Monitoring**

* TR 1.4: Monitoring resources

**Performance Efficiency**

* TR 5.1: Global deployment to reduce latency
* TR 2.2: Global Routing policy
* TR 1.2: Elasticity
* TR 13.1: Microservice architecture choice
* TR 1.3: Update rate for elasticity
* TR 20.1: Scaling algorithm
* TR 10.2: Workload-specific hardware
* TR 1.5: Replacement policy for degraded instances

**Cost Optimization**

* TR 1.2: Elasticity
* TR 11.2: Archive inactive data

**Security**

* TR 3.6: WAF
* TR 3.1, TR 3.2, TR 3.3: Encryption
* TR 3.4: Regional Compliance
* TR 8.1: Identity and Access Management

**Reliability**

* TR 2.1: Availability
* TR 11.1: Data Replication

**Conflicting TRs**

1. **Tenant Identification vs Latency:**

TR 17.1 conflicts with TR 13.1 if data isolation requires a complex architectural design that reduces microservice efficiency. TR17.1 could increase complexity and potentially slow down TR 5.1 efforts.

1. **Monitoring vs Elasticity:**

TR 1.4 might conflict with TR 1.2 if monitoring overhead impacts the system's ability to scale quickly during peak booking times (e.g. holiday seasons).

1. **Scaling vs Workload-specific Hardware:**

TR 20.1 and TR 10.2 could conflict when trying to optimize for different types of critical loads such as peak booking activities where resources are provisioned as fast as possible.

TR 1.3 might create challenges with TR 2.1, especially during critical booking periods.

1. **Encryption vs. Elasticity and Latency:**

TR 3.1, 3.2, 3.3 could significantly impact TR 5.1, TR 20.1 adding additional processing overhead. TR8.1 might introduce further authentication overhead during high-traffic booking scenarios.

1. **Cost-Effective Archival vs Delays in Retrieval:**

TR 11.2 might conflict with TR 2.1 if archived data is needed quickly significantly affecting TR 5.1

1. **Data Replication vs. Cost Efficiency:**

TR 11.1 conflicts with TR 1.2. Replicating resources will lead to additional costs of deployment and maintenance.

## 4.1 The basic building blocks of the design

**Amazon S3:** We will utilize S3 to store static content like hotel images, user-uploaded documents, and archived transaction logs. By implementing server-side encryption (SSE) and lifecycle policies, we will ensure data security and compliance with data protection standards. This will satisfy TR 11.2 and TR 3.3.

**Amazon CloudFront:** CloudFront will be used to deliver static assets globally with low latency. Edge locations will cache frequently accessed resources, enhancing user experience when browsing hotel listings. We will enforce HTTPS to secure content delivery, aligning with TR 5.1 and TR 3.3.

**AWS Route 53**: We will configure Route 53 as our DNS service to optimize request routing based on user location. This will minimize latency and improve load times for hotel search and checkout processes, addressing TR 2.2 and TR 2.1.

**AWS WAF:** We will implement WAF to protect our system from common web attacks like SQL injection and XSS, especially during critical user interactions such as registration and payment. Custom WAF rules will be applied to safeguard sensitive endpoints, fulfilling TR 3.6.

**Application Load Balancer (ALB):** To ensure high availability and scalability, we will deploy ALB to distribute traffic across multiple backend services. Health checks will be configured to automatically route traffic away from unhealthy instances, maintaining system reliability during peak periods. This will satisfy TR 1.2 and TR 2.1.

**Amazon EKS[5]:** We will use EKS to manage and scale our containerized microservices, which will power functionalities like hotel search, booking processing, and user account management. Auto-scaling will be enabled to accommodate fluctuating demand, supporting TR 1.2, TR 13.1, and TR 2.1.

**Amazon EC2:** Specific services like recommendation engines and dynamic pricing algorithms will be hosted on EC2 instances. Auto-scaling groups and instance replacement policies will be implemented to ensure system availability and scalability, addressing TR 1.3, TR 1.5, and TR 20.1.

**Amazon RDS:** We will leverage RDS to store structured data like user profiles, booking details, and payment history. Multi-AZ deployments and encryption at rest will safeguard data integrity and security, satisfying TR 11.1, TR 3.1, and TR 2.1.

**AWS Lambda:** We will utilize Lambda to execute event-driven tasks like sending email notifications, processing feedback forms, and resizing images. This serverless approach will provide scalability and cost efficiency, addressing TR 3.4 and TR 10.2.

**Amazon CloudWatch**: CloudWatch will be used to monitor critical system components and generate alerts for anomalies. This will help us maintain a seamless user experience and address TR 1.4.

**AWS IAM:** We will implement IAM to enforce granular access controls and secure authentication mechanisms like MFA, to enforce RBAC for our developers satisfying TR 8.1.

## 4.2 Top-level, informal validation of the design

**Tenant Identification**

We will isolate tenant data (TR 17.1) by configuring Amazon RDS with a multi-tenant schema. Each tenant (user group) will be assigned a unique identifier for database operations. Row-level security policies will strictly isolate tenant data. Additionally, S3 bucket policies and prefix-based object segmentation will separate tenant-uploaded files like booking confirmations and invoices. IAM policies will further restrict access to tenant-specific resources, ensuring compliance with isolation requirements.

To address potential conflicts between tenant isolation (TR 17.1) and the microservices architecture (TR 13.1), we will adopt a domain-driven approach where each microservice enforces its data isolation boundaries. This ensures data security while maintaining efficient service communication and deployment scalability, mitigating latency concerns (TR 5.1).

**Monitoring**

For resource monitoring (TR 1.4), we will use Amazon CloudWatch to collect and aggregate logs from all application components, including EKS pods, RDS databases, and EC2 instances. Custom CloudWatch dashboards will visualize key metrics such as API latency, query execution time, and request throughput. CloudWatch Alarms will notify the operations team of anomalies, like high error rates in the hotel search service or underperforming EC2 instances. We will also analyze logs from AWS WAF to detect and respond to suspicious activity in real time.

To minimize the impact of monitoring overhead (TR 1.4) on elasticity (TR 1.2), monitoring will focus on lightweight, essential metrics like CPU utilization during peak demand. Additionally, the auto-scaling mechanism will use aggregated metrics to ensure quick scaling without delays.

**Performance Efficiency**

To optimize performance and reduce latency globally (TR 5.1, TR 2.2), we will use AWS Route 53 for latency-based routing to direct users to the nearest regional endpoints. Amazon CloudFront will cache static assets like hotel images and JavaScript files at edge locations, ensuring low-latency access worldwide. For dynamic workloads like booking queries, EKS will be configured with horizontal pod auto-scaling[7] to meet fluctuating user demands, satisfying TR 1.2 (elasticity).

To handle resource-intensive tasks like dynamic pricing algorithms and hotel recommendation engines, we will leverage Amazon EC2 instances (TR 1.3, TR 20.1). We will select compute-optimized instance types like the C5 family (e.g., c5.large or c5.xlarge) for their high performance-to-cost ratio, which are ideal for CPU-intensive workloads. These instances will power applications like real-time pricing adjustments and recommendation engines, processing large datasets and delivering personalized hotel suggestions quickly.

To minimize costs without compromising performance, we will strategically select different EC2 instance types based on the nature and priority of specific workloads:

* **Spot Instances for Non-Critical or Batch Processing Tasks:** We will leverage spot instances, offering significant cost savings (up to 90% off on-demand prices), for workloads that are flexible in timing or do not require consistent availability. Examples include batch data processing, generating nightly booking analytics, and pre-computing recommendations during non-peak hours.
* **On-Demand Instances for Real-Time and Critical Workloads:** On-demand instances will be reserved for critical services that require predictable availability and immediate scalability, such as dynamic pricing engines and hotel search and booking APIs.
* **Reserved Instances for Steady-State Workloads:** For predictable and continuous workloads, like database hosting or user authentication services, we will purchase reserved instances.
* **Burstable Instance Types for Low-Resource Services:** We will use burstable instance types like T3 for low-demand or less resource-intensive tasks such as session management or monitoring auxiliary services.
* **Dedicated Instances for Security and Compliance:** For services handling highly sensitive data, like payment processing or compliance reporting, we will use dedicated instances to ensure physical isolation and compliance with regional data protection regulations.

To complement the processing capabilities of EC2, we will use Elastic Block Store (EBS) volumes for high-throughput, low-latency storage of temporary data generated by resource-heavy workloads. EBS volumes will be provisioned with performance-optimized configurations (e.g., GP3 or IO2 volumes) and placement groups will facilitate efficient communication between instances. The microservices architecture on EKS supports decoupled functionality like hotel search, booking management, and user accounts (TR 13.1). Each service will be deployed as an independent pod with resource-specific configurations, such as GPU-backed nodes for image processing (TR 10.2).

Load balancing can effectively resolve conflicts between scaling algorithms (TR 20.1) and workload-specific hardware (TR 10.2) by distributing traffic across different instance types based on workload requirements. For example, compute-optimized EC2 instances (like the C5 family) can handle real-time, high-performance tasks like pricing algorithms, while burstable instances manage lightweight tasks like session management, and spot instances handle batch jobs. AWS Elastic Load Balancer (ELB) can intelligently route traffic to the most suitable resource, ensuring that real-time critical services remain responsive during peak demand periods by prioritizing high-performance instances. Load balancing integrates with auto-scaling policies to dynamically allocate resources based on current demand, maintaining cost efficiency without sacrificing performance. Additionally, it ensures high availability by routing traffic away from unhealthy instances to healthy ones, guaranteeing continuous service even during scaling events. This combination of intelligent workload distribution and automatic failover helps balance the need for both scalability and specialized hardware.

**Cost Optimization**

To minimize costs without compromising performance, we will implement Amazon S3 lifecycle policies to archive inactive data to Glacier Deep Archive (TR 11.2). The potential retrieval delays (TR 11.2 vs TR 2.1) will be mitigated by pre-fetching data likely to be needed during high-demand periods, such as historical booking patterns during holiday seasons. Elasticity configurations for EC2 and EKS will ensure resources are scaled down during low-demand periods (TR 1.2). Data replication strategies (TR 11.1) will focus on critical regions and services only, balancing reliability and cost efficiency.

**Security**

To secure the application against external threats, we will use AWS WAF to enforce rules mitigating common vulnerabilities like SQL injection and cross-site scripting (TR 3.6). WAF will be integrated with CloudFront to protect against DDoS attacks at the edge. Sensitive data will be encrypted using AWS KMS and stored securely in RDS and S3 buckets (TR 3.1, TR 3.2, TR 3.3). Encryption keys will be rotated periodically. Regional compliance (TR 3.4) will be ensured by hosting data in specific AWS regions. We will use IAM roles with granular permissions to restrict access to production environments. Multi-factor authentication (MFA) will be enforced for all user accounts, satisfying TR 8.1. Audit trails from CloudTrail will provide visibility into all IAM-related activities.

To address conflicts between encryption overhead and performance (TR 5.1, TR 20.1), encryption will be implemented with minimal latency by using hardware-accelerated cryptographic operations in EC2 instances. IAM roles with granular permissions will restrict access to production environments, and multi-factor authentication (MFA) will enforce secure user access (TR 8.1).

**Reliability**

To ensure high availability and reliability in line with TR 2.1, we will use Elastic Load Balancers (ALBs) to distribute incoming traffic evenly across multiple backend services. For the database layer, Amazon RDS will leverage Multi-AZ deployments to provide automatic failover. To further enhance reliability, we will implement Multi-Region deployments to replicate critical data across geographically separate locations. We will have a rollback strategy using point-in-time recovery and manual failover mechanisms. To safeguard against data loss, RDS will utilize automated backups and manual snapshots, fulfilling the requirements of TR 11.1.. For static content, Amazon S3 Cross-Region Replication will duplicate data across multiple regions. Auto Scaling Groups for both EC2 instances and EKS clusters will be configured to automatically replace degraded instances, ensuring the system can maintain its performance and reliability, aligning with TR 1.5.

To resolve the conflict between reliability and deployment costs, we prioritize availability as the cornerstone of our application. While this may result in higher deployment costs, ensuring robust availability will foster customer trust and contribute to the long-term success of the product.

By leveraging the power of AWS services, we have designed a solution that prioritizes performance, security, reliability, and cost-efficiency. The multi-tenant architecture ensures data isolation and privacy, while the monitoring and alerting systems enable proactive maintenance and issue resolution. The combination of EC2 instances, EKS, and serverless functions provides the flexibility and scalability needed to handle fluctuating workloads and future growth. Additionally, the implementation of security best practices, including encryption, access controls, and regular security audits, safeguards sensitive data and protects against potential threats.

## 4.3 Action Items and Rough Timeline

Skipped

# 5 The Second Design

## 5.1 Use of the Well-Architected Framework [2]

The framework suggests following distinct steps to follow while designing applications[30] in the cloud:

1. **Stop guessing your capacity needs:**

If you make a poor capacity decision when deploying a workload, you might end up sitting on expensive idle resources or dealing with the performance implications of limited capacity. With cloud computing, these problems can go away. You can use as much or as little capacity as you need, and scale up and down automatically.

1. **Test systems at production scale:**

In the cloud, you can create a production-scale test environment on demand, complete your testing, and then decommission the resources. Because you only pay for the test environment when it's running, you can simulate your live environment for a fraction of the cost of testing on premises.

1. **Automate to make architectural experimentation easier:**

Automation allows you to create and replicate your workloads at low cost and avoid the expense of manual effort. You can track changes to your automation, audit the impact, and revert to previous parameters when necessary.

1. **Allow for evolutionary architectures:**

In a traditional environment, architectural decisions are often implemented as static, onetime events, with a few major versions of a system during its lifetime. As a business and its context continue to evolve, these initial decisions might hinder the system's ability to deliver changing business requirements. In the cloud, the capability to automate and test on demand lowers the risk of impact from design changes. This allows systems to evolve over time so that businesses can take advantage of innovations as a standard practice.

1. **Drive architectures using data:**

In the cloud, you can collect data on how your architectural choices affect the behavior of your workload. This lets you make fact-based decisions on how to improve your workload. Your cloud infrastructure is code, so you can use that data to inform your architecture choices and improvements over time.

1. **Improve through game days:**

Test how your architecture and processes perform by regularly scheduling game days to simulate events in production. This will help you understand where improvements can be made and can help develop organizational experience in dealing with events.

\begin{enumerate}

\item \textbf{Eliminate the uncertainty of capacity planning}

Inaccurate capacity planning during workload deployment can result in underutilized resources, leading to financial inefficiencies, or insufficient resources, causing performance issues. Cloud computing mitigates these challenges by enabling dynamic scalability, allowing organizations to provision resources as needed and adjust capacity automatically to match demand.

\item \textbf{Conduct testing at production scale}

The cloud facilitates the creation of production-scale test environments on demand. Organizations can perform testing, validate systems under real-world conditions, and decommission resources upon completion. This approach is cost-effective, as payment is required only for the duration of the testing, making production-scale simulations accessible without the significant expense of on-premises infrastructure.

\item \textbf{Leverage automation to enhance architectural experimentation}

Automation enables the efficient creation and replication of workloads, minimizing manual effort and associated costs. It also facilitates tracking, auditing, and reverting changes, thereby fostering an iterative and data-driven approach to architectural experimentation and refinement.

\item \textbf{Facilitate evolutionary architectural designs}

Traditional environments often enforce static, inflexible architectural decisions, limiting adaptability to changing business needs over time. In contrast, the cloud's capabilities, such as automation and on-demand testing, reduce the risks associated with iterative design changes. This adaptability enables systems to evolve in alignment with organizational innovation and emerging requirements.

\item \textbf{Employ data-driven architectural decisions}

Cloud computing provides mechanisms to collect and analyze data on the performance and behavior of workloads in response to architectural choices. These insights support evidence-based decision-making, enabling continuous improvements. Additionally, treating infrastructure as code ensures that data-driven refinements can be systematically incorporated into architectural strategies.

\item \textbf{Enhance resilience through game-day exercises}

Regularly scheduled game days, simulating production-level events, allow organizations to assess the robustness of their architectures and processes. These exercises identify areas for improvement and develop organizational expertise in managing and mitigating real-world challenges effectively.

\end{enumerate}

## 5.2 Discussion of Pillars [3]

Discuss in detail one pillar per team member.

1. **Security**

Security in cloud architecture encompasses the ability to protect information, systems, and assets while delivering business value through risk assessments and mitigation strategies. It involves implementing strong data protection, access controls, and continuous monitoring to safeguard against threats. For a hotel booking platform handling sensitive customer data (payment information, personal details) and financial transactions, security is paramount. This directly addresses BR3 (compliance and security) and BR9 (identity management).

**Implementation through Technical Requirements:**

1. Data Protection

- TR3.1: AES-256 encryption for data at rest

- TR3.2: TLS 1.2 for data in transit

- TR3.3: Isolated key management service

These measures protect customer payment information and personal details from breaches.

2. Access Control

- TR9.1: Clearly defined identity and access permissions

- TR9.3: Quarterly access review process

This ensures only authorized personnel can access sensitive booking data and administrative functions.

3. Threat Prevention

- TR3.6: Web Application Firewall (WAF)

Protects against common web attacks like SQL injection, XSS, which are particularly important for a booking platform processing financial transactions.

**Impact on Business**

The implementation of robust security measures directly influences the platform's trustworthiness and financial success. By ensuring encrypted data storage and transmission, along with strict access controls, the platform builds customer confidence in handling sensitive payment and personal information. This trust translates into higher booking rates and customer retention. The Web Application Firewall and regular security audits protect against financial losses from fraudulent transactions, while compliance with payment processing regulations prevents costly penalties. Perhaps most importantly, protecting against data breaches safeguards the platform's reputation – a critical asset in the competitive hotel booking industry where a single security incident could lead to significant customer exodus and long-term brand damage.

**2. Performance Efficiency**

Performance Efficiency focuses on the efficient use of computing resources to meet system requirements and maintain that efficiency as demand changes and technologies evolve. It ensures the system responds quickly to user actions and scales appropriately under load. A hotel booking platform must handle peak travel seasons and provide instant search results. This addresses BR1 (scalability), BR5 (low latency), and BR14 (fast search).

**Implementation through Technical Requirements**

1. Global Performance

- TR5.1: Multi-region deployment

- TR2.2: Geographic request routing

Ensures users worldwide get fast access to the booking platform.

2. Elastic Scaling

- TR1.2: Handle 5220 requests/second

- TR20.1: Automated scaling policies

- TR1.5: Auto-replacement of degraded instances

Maintains performance during peak booking seasons (holidays, special events).

**Impact on Business**

The performance of a hotel booking platform directly correlates with revenue generation and market competitiveness. Fast search results and responsive booking processes significantly reduce abandonment rates, as modern users expect near-instant responses. The platform's ability to handle 5,220 requests per second during peak travel seasons ensures uninterrupted service during crucial high-revenue periods like holidays and special events. Global performance optimization through multi-region deployment enables market expansion, allowing the business to serve customers worldwide with consistent speed. This reliable performance leads to higher conversion rates, as users are more likely to complete bookings when the experience is smooth and responsive. Additionally, efficient performance allows the platform to effectively compete with major players in the market, as speed and responsiveness are key differentiators in the online booking industry.

**3. Reliability**

Reliability refers to a system's ability to consistently perform its intended functions correctly and recover quickly from failures. It ensures the system remains available and operational even under adverse conditions. Hotel bookings must be processed 24/7 without fail. This addresses BR2 (high availability) and BR6 (disaster recovery).

**Implementation through Technical Requirements**

1. High Availability

- TR1.1: Multi-AZ deployment

- TR2.1: 99.999% availability target

- TR2.3: Redundant IP addresses

Ensures the booking platform remains operational even during infrastructure failures.

2. Disaster Recovery

- TR6.1: Continuous data snapshots

- Cross-zone replication

Protects against data loss and enables quick recovery.

3. Fault Tolerance

- TR1.5: Automatic instance replacement

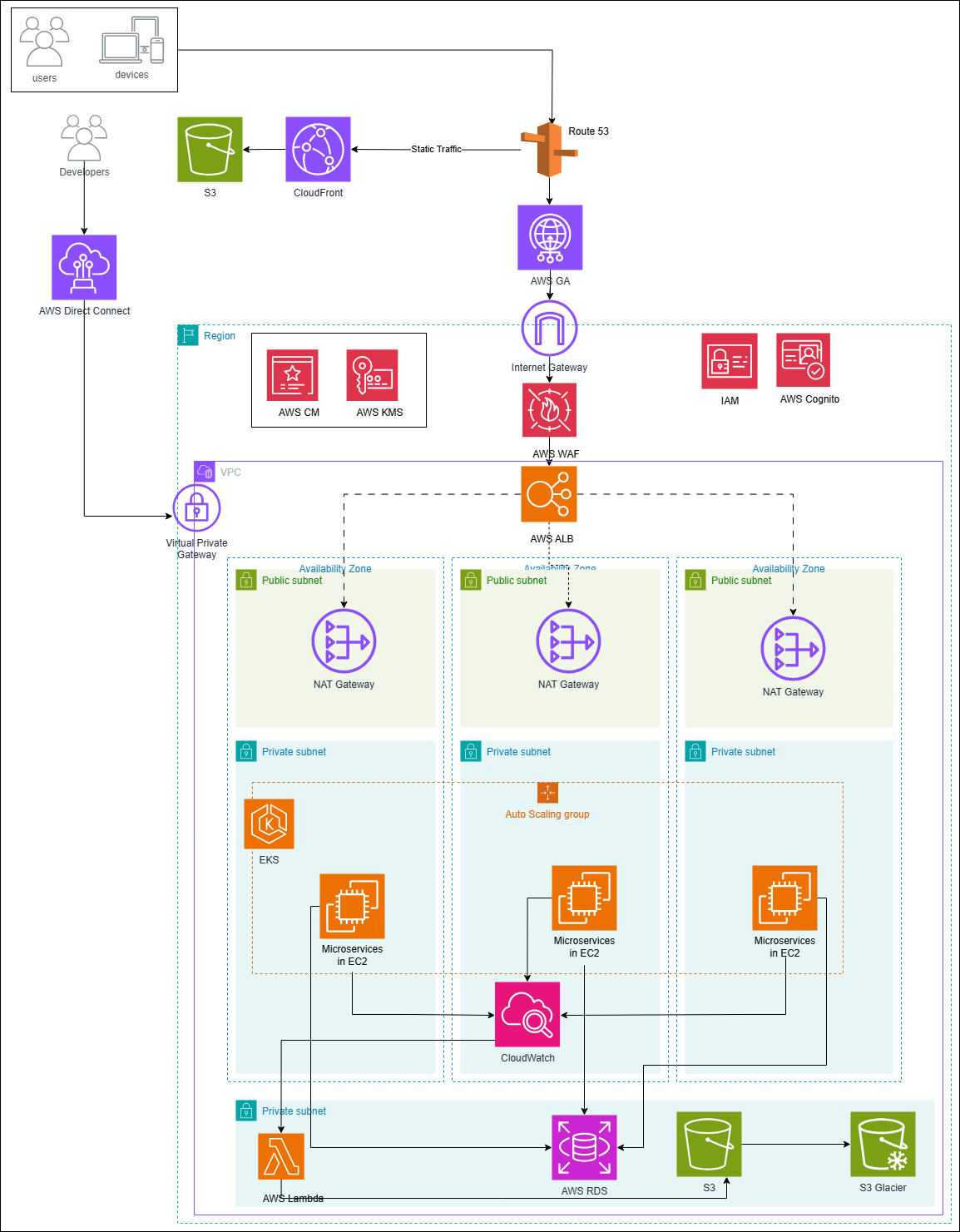
- Multi-region redundancy

- Prevents single points of failure from affecting bookings.

**Impact on Business**

For a hotel booking platform, reliability directly impacts revenue streams and customer trust. The implementation of 99.999% availability ensures the platform can process bookings 24/7, critical in a global market where bookings occur around the clock. Multi-region deployment and automatic failover mechanisms prevent service interruptions, protecting against revenue loss during system failures. Continuous data snapshots and cross-zone replication safeguard booking information, preventing the costly and reputation-damaging scenario of lost reservations. This reliability builds long-term customer trust, essential for repeat bookings and positive word-of-mouth recommendations. During peak travel seasons, when booking volumes surge, the platform's reliability ensures consistent operation, maximizing revenue opportunities. The ability to maintain service during infrastructure failures or disasters sets the platform apart from less robust competitors, contributing to market leadership and sustained business growth.

## 5.3 Use of Cloudformation Diagrams. Present your design using Cloudformation templates.



**Design of Data Plane**

1. Users interact with the Booking Application via the Web Client on their systems, performing transactions like searching for hotels, checking room availability and prices, or booking a hotel room. For all these actions, the requests come to our backend systems.
2. User requests are sent to the Route 53 DNS service, which resolves the IP address of the AWS Global Accelerator[8], routing the request to the nearest regional cluster’s Application Load Balancer (ALB).
3. The AWS Gateway intercepts the request, and AWS WAF checks it for malicious activity. If the request is valid, it is passed forward.
4. The request enters our VPC and hits the ALB.
5. The Application Load Balancer receives the request and, based on the Round-Robin algorithm, forwards it to the appropriate microservice for processing.
6. The microservice processes the request by accessing AWS RDS to retrieve data or execute booking transactions. It performs the necessary actions based on the parameters in the request.
7. AWS Key Management Service (KMS) ensures data encryption at rest, and AWS Certificate Manager ensures secure transmission to the correct endpoint.
8. The processed response is sent back to the user via the Application Load Balancer and AWS Global Accelerator.
9. AWS CloudWatch monitors all microservices and their instances, capturing logs and reporting errors with alerts. It also tracks custom metrics for overall system performance.
10. AWS Lambda Function is used to extract Logs from CloudWatch and push them to S3 periodically.
11. Microservices run on EC2 instances and are configured with the required resources as defined for each microservice.

## 

## 5.4 Validation of the design

1. Tenant Identification:

In a containerized multi-tenant application, AWS Cognito generates a JSON Web Token (JWT) containing the tenant ID upon user authentication. This JWT is passed via HTTP headers to the containerized application, where a middleware layer validates the token using Cognito’s public keys and extracts the tenant ID. The tenant ID is then used to establish the tenant context, driving actions like database filtering, routing, and resource access. In a microservices architecture, the tenant context propagates across services through API requests, ensuring data isolation and tenant-specific business logic. This architecture ensures secure, scalable tenant isolation within containers.

IAM (Identity and Access Management) can validate developers in the cloud architecture by assigning unique IAM roles or policies that define their permissions based on their responsibilities. Developers authenticate using AWS credentials and their actions are validated against these IAM policies to ensure compliance with access controls. Resource utilization can be monitored by enabling AWS CloudTrail to track API calls and actions, along with AWS CloudWatch for real-time monitoring of resource metrics like CPU, memory, and network usage. Additionally, tagging resources with developer-specific identifiers allows fine-grained usage tracking and reporting, ensuring accountability and optimized resource management.

Thus this validates the TRs - TR17.1.

1. Monitoring:

AWS CloudWatch plays a vital role in ensuring the effectiveness of the proposed architecture by providing comprehensive monitoring and insights into resource utilization. Custom metrics in CloudWatch allow us to track specific parameters for each resource in the infrastructure, such as CPU usage, memory consumption, or disk I/O. By running a lightweight process on resources like EC2 instances or EKS workloads, we can periodically send these metrics to CloudWatch, enabling a clear understanding of their daily usage patterns. This information helps calculate idle times and identify underutilized resources, empowering administrators to make informed decisions about scaling down or reallocating resources to optimize costs.

CloudWatch Alarms add another critical layer to the monitoring strategy by proactively alerting administrators when resource utilization crosses predefined thresholds. For example, alarms can be set to trigger when an EC2 instance’s CPU usage exceeds 80% for an extended period or when the storage of an RDS instance approaches its maximum limit. These alarms ensure that potential performance bottlenecks or service disruptions are identified early, allowing for timely corrective actions. Additionally, integrating these alarms with automated response mechanisms, such as triggering Auto Scaling or notifying via SNS, enhances the system's resilience and responsiveness.

For shared resources like NAT Gateways and ALBs, CloudWatch provides aggregated metrics that offer insights into their overall utilization. Monitoring these metrics ensures that shared components are neither overburdened nor underutilized, striking the right balance between performance and cost efficiency. CloudWatch’s ability to provide a centralized view of resource health, combined with its robust alerting capabilities, ensures that our architecture remains reliable and efficient while keeping operational overhead to a minimum.

This validates TR1.4.

1. Reliability:

The architecture ensures High Availability by deploying resources across multiple Availability Zones (AZs) within a region, providing redundancy and fault tolerance. The use of Auto Scaling Groups for EC2 instances ensures that the system can dynamically adjust to traffic demands, maintaining availability even during high-load scenarios. The Application Load Balancer (ALB) further distributes traffic across instances in different AZs, reducing the risk of single points of failure. Additionally, services like Amazon Route 53 and AWS Global Accelerator provide reliable DNS and global traffic routing, ensuring that users experience minimal latency and uninterrupted service during regional outages or failures.

For Data Replication, the architecture incorporates multiple strategies to ensure consistency and durability. Amazon RDS, deployed within private subnets, automatically replicates data across AZs using Multi-AZ deployments, providing resilience against instance or storage failures. Similarly, S3 offers durable storage with automatic data replication across multiple facilities within a region. For long-term data retention and archival, S3 Glacier further ensures that critical data is securely replicated and preserved. These replication mechanisms, combined with the reliability of AWS-managed services, ensure that data remains highly available, consistent, and protected from loss or corruption.

**Test Plan for Hotel Booking Service Infrastructure**

The test plan for the hotel booking service infrastructure is designed to comprehensively validate its high availability, performance, and reliability. By systematically examining various operational scenarios, we aim to ensure the system can effectively handle diverse challenges while maintaining consistent service quality.

**High availability testing** will focus on the infrastructure's resilience under challenging conditions. This involves **simulating Availability Zone failures**, verifying the system's ability to redirect traffic, and maintain operational continuity. Auto-scaling mechanisms will be rigorously tested across different traffic scenarios, from low-traffic periods to peak booking seasons, ensuring dynamic resource allocation and consistent performance.

**Data integrity testing** will be critical, with a particular focus on verifying robust replication strategies. Multi-AZ RDS failover scenarios will be simulated to confirm automatic failover processes, prevent double bookings, and maintain accurate room availability tracking. Storage reliability tests will encompass both active booking documents and long-term archival, ensuring secure and consistent data management.

**Performance evaluation** will simulate a wide range of real-world traffic patterns, including typical business days and high-demand periods during events and holidays. The testing will create complex workloads combining room searches, booking confirmations, payment processing, and customer support interactions. Global traffic routing will be assessed to validate minimal latency and consistent performance across different geographical regions.

**Disaster recovery testing** will be comprehensive, simulating regional failures and complete infrastructure unavailability. These tests will verify the system's ability to automatically failover, minimize service disruption, and maintain data integrity. Additionally, security and compliance checks will validate network configurations, encryption mechanisms, and adherence to booking data protection standards.

Key performance indicators will be closely monitored, including **response times, error rates, instance health, database failover duration, and overall user experience consistency.**

This validates the following TR’s: TR 11.2, TR 1.2.

1. Performance Efficiency:

For our Hotel Booking System,

Considering the given load assumptions, we need to design Architectural Solutions to achieve performance efficiency-

**Daily Visitors**: 10 million across the globe

**Average Requests per Session**: 15

**Total Requests per Day**:  
10 × 106 × 15 = 150 × 106 requests,

**Requests per Second (RPS)** (assuming 24 hours evenly distributed, global platform):  
(150×106) / 86400 ≈ 1,740 RPS

**During Peak Hours**:

Assume a **3x peak multiplier** to account for concentrated activity: 1,740×3 ≈ **5,220 RPS**

To handle the 5220 Requests Per Second within a cluster, we have Microservice architecture in place for individual scaling of services. To achieve efficiency over that, we should design a good architecture with Load Balancer in place.

The load balancer we chose to implement would be the AWS application Load Balancer. Functioning as a layer-7 load balancer, it has access to HTTP fields of the incoming requests to make load balancing decisions. Based on the workload of the requests further routing can be done to utilize specialized hardware such as more powerful EC2 instances for larger load processing.

**Global Deployment-**

To provide low latency, high availability, and robust global coverage, the infrastructure is distributed across three key availability zones. North America is served by the US East region in Virginia, which handles 40% of the traffic with services including the primary RDS database, caching, and core system functionality. Europe is supported by the EU Central region in Frankfurt, accommodating 30% of the traffic. Asia-Pacific traffic is managed through the AP Southeast region in Singapore, which supports 20% of users. Seasonal traffic spikes of 60-80% during holidays, such as Christmas and summer vacations, necessitate a scalable infrastructure to handle peak demand effectively. This configuration ensures efficient service delivery across all user regions while maintaining high reliability and scalability.

**Infrastructure Deployment**

We are implementing a containerized approach to deploy our infrastructure using Kubernetes for dynamic container orchestration. The master node is managed by AWS Elastic Kubernetes Service (EKS), while the worker nodes run the kubelet service and kube-proxy. This setup allows efficient management of worker nodes and integrates seamlessly with AWS Auto Scaling Groups to dynamically adjust the number of provisioned containers. Resource limits and requests can also be configured to align with workload demands, ensuring optimal resource utilization.

**Scaling and Autoscaling**

To handle dynamic workloads, we rely on Kubernetes' Horizontal Pod Autoscaler (HPA) [1]. The HPA monitors CPU utilization across active pods and automatically scales resources up or down based on traffic demands. During peak periods, such as holidays or New Year’s, when there are unpredictable spikes in activity, this mechanism ensures the system scales effectively to maintain performance and minimize latency.

**Latency Optimization**

Ensuring low latency is critical for the hotel booking system, especially during peak booking seasons. We utilize AWS Route 53 DNS for latency-based routing, which directs users to their nearest deployment region. This ensures that requests are processed with minimal response times, providing a better user experience during periods of high concurrency.

**Load Balancing Strategy**

The Application Load Balancer (ALB) dynamically integrates new pods into the load-balancing pool based on their labels. This ensures that the infrastructure remains updated as pods are added or removed. The ALB constantly monitors the health of pods and routes traffic only to healthy ones, ensuring reliability and consistent performance.

To balance the load effectively, we use a **Load Optimization Ratio (LOR) algorithm**[21]. This algorithm evaluates the system using the equation **|L1(T) − L2(T)| < ϵ** [1], where L1 and L2​ represent the loads on two pods at a given time **T = 30 min, and ϵ=250**. The time T is considered 30 minutes because we can have small windows of peak booking sessions for a region in a day. If the time interval is too small, we will be checking a lot which will lead to additional processing overheads and if the time interval is too big, we might miss some issue in the load balancer leading to one pod serving a bigger share of requests. We consider ϵ to be 250 requests in this case which is equal to 5% of our expected requests per second. We cannot keep ϵ too large as there are some booking requests that require large amounts of processing capacity. This threshold provides flexibility to account for high volumes of booking transactions while maintaining an even distribution of traffic across pods. The load balance is checked every hour to prevent overloading any individual pod.

**Health Monitoring and Feedback**

To maintain system responsiveness, pod health checks are conducted every **15 seconds.** This ensures that unhealthy pods are quickly identified and removed from the traffic pool. Health statuses are continuously transferred to the Kubernetes API Server and stored in etcd. The load balancer uses this information to make informed decisions, routing traffic only to pods that are functioning optimally.

### **Dynamic Scaling and Adaptability**

The system is designed to adapt dynamically to fluctuating demand. New pods are automatically integrated into the infrastructure through Kubernetes' label-based configuration. This seamless addition ensures scalability without disruption, allowing the system to maintain high availability and reliability even during unexpected surges in traffic.

### **Benefits of the Approach**

By combining Kubernetes' scalability, the HPA’s auto scaling capabilities, health-driven routing, and latency-based DNS, our hotel booking system ensures optimal performance and resource utilization. This design supports a globally distributed user base, minimizes service disruptions, and delivers a consistent and responsive user experience during both regular and peak usage periods.

This validates TR’s: TR5.1, TR2.2, TR1.2, TR13.1, TR1.3, TR20.1, TR10.2, TR1.5

1. Cost Optimization:

For TR1.2 (Elasticity), Kubernetes on AWS EKS ensures efficient resource utilization by dynamically scaling workloads based on demand. During peak booking periods, Kubernetes can automatically add pods to handle increased traffic, while scaling down during off-peak times minimizes operational costs. This elasticity ensures that resources are only consumed as needed, reducing unnecessary expenditures while maintaining performance.

For TR11.2 (Archive inactive data), the system uses Amazon S3 for primary storage and S3 Glacier for archiving inactive data. Booking records or logs that are no longer frequently accessed are automatically moved to S3 Glacier using lifecycle policies. This significantly lowers storage costs while preserving data for compliance or future analysis. By archiving older data efficiently and enabling elastic scaling, the architecture balances cost savings with the performance needs of the Hotel Booking System.

This validates TR’s - TR1.2, TR11.2.

1. Security

The architecture ensures the security of the Hotel Booking System through AWS services and best practices. For TR3.6, AWS WAF protects the application from web-based threats like SQL injection and cross-site scripting by filtering malicious traffic before it reaches the backend, ensuring uninterrupted and secure service for users.

Encryption requirements (TR3.1, 3.2, 3.3) are addressed using AWS KMS for managing encryption keys. Data in S3, RDS, and S3 Glacier is encrypted at rest, while TLS secures communication between users and application components, protecting sensitive customer data like booking details and payment information. Regional compliance (TR3.4) is maintained by deploying all resources within a single AWS region, ensuring data residency and adherence to local regulations.

For TR8.1, AWS IAM enforces access control by granting the least privilege necessary to users and services. Role-based access ensures that only authorized personnel can access sensitive parts of the Hotel Booking System, while multi-factor authentication adds an additional security layer. These measures safeguard customer data and maintain the application’s integrity.

This validates TR’s: TR3.6, TR3.1, TR3.2, TR3.3, TR3.4, TR8.1

## 5.5 Design principles and best practices used

1. Democratize Advanced Technologies:

By leveraging AWS-managed services such as EKS for Kubernetes, RDS for database management, and S3 for storage, our Hotel Booking System simplifies the integration of advanced technologies. These services offload complex tasks such as container orchestration, database administration, and storage scaling to AWS, allowing the team to focus on application development. Kubernetes further enables the microservice architecture, making the system modular, scalable, and easier to manage.[20]  
(TR13.1 - Microservice architecture choice, TR1.2 - Elasticity)

1. Go Global in Minutes:  
   The system is designed to leverage AWS Global Accelerator and deploy workloads across multiple AWS Regions, ensuring low-latency booking services for users worldwide. With this setup, the system enhances customer experience, provides fault tolerance, and ensures high availability even during regional outages. [20]  
   (TR1.2 - Elasticity, TR20.1 - Scaling algorithm)
2. Consider Mechanical Sympathy:  
   The system uses AWS Application Load Balancer (ALB) to distribute traffic efficiently among microservices deployed in Kubernetes. The ALB dynamically adjusts routing based on workload-specific needs and performance metrics, ensuring that compute resources are aligned with application demands. Additionally, RDS is used for transaction-heavy operations like booking and payment processing due to its optimized performance for relational queries. [20]  
   (TR10.2 - Workload-specific hardware, TR20.1 - Scaling algorithm)
3. Implement Elastic and Resilient Replacement Policies:  
   Kubernetes automatically replaces unhealthy pods and nodes, ensuring the system remains operational with minimal manual intervention. When a node fails or exceeds resource thresholds, the Auto Scaling Group provisions new instances, maintaining system health and elasticity. This ensures that the booking application dynamically adjusts to changing traffic demands without interruptions. [20]  
   (TR1.5 - Replacement policy, TR1.2 - Elasticity)
4. Implement Cloud Financial Management:  
   By leveraging AWS tools such as Cost Explorer, Budgets, and Savings Plans, the Hotel Booking System implements cloud financial management to ensure cost efficiency. This approach helps track usage patterns, optimize expenditures, and allocate budgets to different workloads effectively. For example, the system uses Reserved Instances for predictable workloads and Spot Instances for non-critical tasks, reducing overall costs while maintaining high availability. [19]  
   (TR1.2 - Elasticity)
5. Measure Overall Efficiency:  
   AWS services like CloudWatch and Cost and Usage Reports provide insights into the system's performance and cost metrics. By analyzing these metrics, the Hotel Booking System evaluates the efficiency of its architecture, ensuring that the cost of delivering services aligns with the revenue generated from bookings. This ongoing monitoring helps fine-tune the infrastructure and improve business outcomes. [19]  
   (TR11.2 - Archive inactive data)
6. Implement a Strong Identity Foundation:  
   The Hotel Booking System ensures secure access to AWS resources by implementing IAM roles with the principle of least privilege. Access is granted only to specific services required for individual components, and static credentials are replaced by short-lived session tokens generated through AWS STS. Centralized identity management is enforced using AWS SSO, enabling secure access across the system. [18]  
   (TR8.1 - IAM)
7. Apply Security at All Layers:  
   A defense-in-depth approach is implemented in the architecture. AWS WAF protects the application from web-based attacks like SQL injection and XSS at the edge level, while security groups and network ACLs restrict traffic within the VPC. Load balancers enforce TLS termination for secure communication, and IAM policies ensure granular access to specific resources. [18]  
   (TR3.6 - WAF)
8. Protect Data in Transit and at Rest:  
   All sensitive data, such as user credentials and booking information, is encrypted using AWS KMS. S3 buckets storing static content, and RDS instances hosting transactional data, use server-side encryption. Data in transit is secured using HTTPS for all communication between the client, API Gateway, and backend services, ensuring end-to-end encryption. [18]  
   (TR3.1, TR3.2, TR3.3 - Encryption)
9. Automatically Recover from Failure:  
   The Hotel Booking System monitors key business KPIs, such as booking success rates and API response times, through AWS CloudWatch. These KPIs trigger alarms that initiate automated recovery actions using AWS Auto Scaling and Elastic Load Balancing to redistribute traffic. For database failures, automated backups and Multi-AZ failover mechanisms in Amazon RDS are used to ensure quick recovery with minimal downtime. [17]  
   (TR2.1 - Availability)
10. Stop Guessing Capacity:  
    The application uses Auto Scaling policies configured to monitor CPU utilization and request throughput. Based on these metrics, additional EC2 instances are automatically provisioned during peak demand, while underutilized instances are terminated during low demand. This ensures optimal resource usage and avoids over-provisioning or resource saturation. Database scaling is achieved through Aurora Serverless [10], which dynamically adjusts capacity to meet fluctuating demands. [17]  
    (TR2.1 - Availability)

## 5.6 Tradeoffs revisited

Our architectural decision-making will follow a two-step approach:

1. Feature Comparison Evaluate potential solutions by analyzing their features, considering AWS capabilities and our application's specific requirements.
2. Even Swaps Method systematically resolves technical trade-offs by comparing and balancing competing architectural attributes.

The final recommendation will synthesize insights from both analyses, ensuring a comprehensive, data-driven architectural selection.

1. **Choosing the right load balancer**

The strategic selection of a Load Balancer is crucial for resolving key technical conflicts in our hotel booking platform, particularly addressing challenges related to **latency (TR 5.1)**, **scalability (TR 1.2)**, **monitoring overhead (TR 1.4)**, **cost efficiency (TR 11.1)**, and **data isolation/security (TR 17.1)**. By implementing an advanced Load Balancer with intelligent routing capabilities, multi-tenant support, and low-latency performance, we can effectively mitigate the tensions between data isolation, system responsiveness, and scaling requirements that would otherwise compromise the platform's architectural integrity.

**Feature based comparison:**

[(n) indicates rank for that alternative(top row) against the particular objective mentioned in the left column]

| **Objective** | **AWS ELB** | **AWS ALB** | **AWS NLB** |
| --- | --- | --- | --- |
| **Protocol Support** | TCP/HTTP/HTTPS (1) | HTTP/HTTPS (1) | TCP/UDP (2) |
| **Performance** | Moderate, suitable for basic apps (2) | Optimized for web apps, low latency (1) | High performance, low latency (1) |
| **Cost** | Fixed pricing based on usage (2) | Based on requests and data processed (1) | Based on connections and traffic (1) |
| **Routing Features** | Basic round-robin routing (2) | Host/Path-based routing, content-based (1) | IP-based, supports complex setups (1) |
| **Scalability** | Suitable for legacy apps (3) | Best for microservices & web apps (1) | Best for high-volume or low-latency needs (2) |
| **Sticky Sessions** | Yes, but limited (2) | Yes (1) | No (3) |

### **Even Swaps Method for Choosing AWS Load Balancer: ALB, NLB, ELB**

### **Step 1: Initial Comparison**

We begin by analyzing the three load balancers, AWS ALB, NLB, and ELB, based on critical Technical Requirements (TRs): **latency (TR 5.1)**, **scalability (TR 1.2)**, **monitoring overhead (TR 1.4)**, **cost efficiency (TR 11.1)**, and **data isolation/security (TR 17.1)**.

| **Load Balancer** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Data Isolation/Security (TR 17.1)** |
| --- | --- | --- | --- | --- | --- |
| **ALB** | Medium | High | Medium | Medium | High |
| **NLB** | Low | High | Low | High | Medium |
| **ELB** | Medium | Medium | Low | Medium | Medium |

At this stage, no alternative is conclusively better than the others, so we proceed to eliminate dominated options step by step.

### **Step 2: NLB vs. ALB (Latency and Scalability)**

**Latency (TR 5.1):**NLB operates at Layer 4 (transport layer), making it optimized for low latency as it directly forwards packets without processing HTTP/HTTPS requests. In contrast, ALB functions at Layer 7 (application layer), introducing additional overhead for features like path-based routing and Web Application Firewall (WAF). NLB wins on latency.

**Scalability (TR 1.2):**Both NLB and ALB scale automatically with traffic patterns, ensuring high throughput and resilience. Since they perform equally well in this regard, scalability remains a tie.

| **Load Balancer** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Data Isolation/Security (TR 17.1)** |
| --- | --- | --- | --- | --- | --- |
| **ALB** | Medium | High | Medium | Medium | High |
| **NLB** | **Low** | High | Low | High | Medium |

**Outcome:**NLB outperforms ALB on latency, but ALB remains in the comparison due to its strong security features.

### **Step 3: ALB vs. ELB (Scalability and Monitoring Overhead)**

**Scalability (TR 1.2):**ALB is designed for modern web applications, supporting dynamic host-based and path-based routing. ELB, being a legacy solution, lacks these advanced features and is less suited for high-demand environments. ALB wins.

**Monitoring Overhead (TR 1.4):**ALB integrates seamlessly with AWS CloudWatch, providing detailed metrics for Layer 7 traffic, while ELB has fewer monitoring capabilities and lacks Layer 7 analysis. Again, ALB wins.

| **Load Balancer** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Data Isolation/Security (TR 17.1)** |
| --- | --- | --- | --- | --- | --- |
| **ALB** | Medium | **High** | **Medium** | Medium | High |
| **ELB** | Medium | Medium | Low | Medium | Medium |

**Outcome:**ELB is eliminated due to weaker scalability and monitoring capabilities.

### **Step 4: NLB vs. ELB (Cost Efficiency and Monitoring Overhead)**

**Cost Efficiency (TR 11.1):**NLB is more cost-effective, operating at the transport layer with minimal processing overhead. ELB incurs similar costs but lacks the modern features offered by NLB or ALB, making it less competitive.

**Monitoring Overhead (TR 1.4):**NLB provides basic but sufficient monitoring capabilities, integrating with CloudWatch. ELB’s outdated monitoring capabilities make it less suitable for detailed traffic analysis. NLB wins.

| **Load Balancer** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Data Isolation/Security (TR 17.1)** |
| --- | --- | --- | --- | --- | --- |
| **NLB** | **Low** | High | Low | **High** | Medium |
| **ELB** | Medium | Medium | Low | Medium | Medium |

**Outcome:**ELB is conclusively eliminated. The final comparison is between NLB and ALB.

### **Step 5: NLB vs. ALB (Security vs. Latency/Cost Efficiency)**

**Data Isolation and Security (TR 17.1):**ALB provides advanced security features, including WAF integration, SSL offloading, and tenant data isolation. These features make ALB the preferred option for secure multi-tenant architectures. NLB, while secure, focuses on packet forwarding and lacks these advanced capabilities. ALB wins.

**Latency and Cost Efficiency (TR 5.1, TR 11.1):**NLB excels in both latency and cost efficiency due to its Layer 4 design, which bypasses application-layer processing. It is particularly suited for real-time, high-throughput scenarios where cost and performance are critical. NLB wins.

| **Load Balancer** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Data Isolation/Security (TR 17.1)** |
| --- | --- | --- | --- | --- | --- |
| **ALB** | Medium | High | Medium | Medium | **High** |
| **NLB** | **Low** | High | Low | **High** | Medium |

### **Learning**

1. **Choose NLB** if latency and cost efficiency are the top priorities (e.g., real-time booking systems with high traffic).
2. **Choose ALB** if security and data isolation are more critical (e.g., multi-tenant environments with strict compliance).
3. **ELB is eliminated** as it fails to meet modern scalability, monitoring, and cost-efficiency requirements.

**Final Selection:**

From our comparison we see that AWS ALB is the best suited solution for our application. The nature of hotel booking platforms involves intricate web-based interactions, requiring sophisticated routing capabilities like host and path-based routing to manage different service endpoints such as search, reservation, payment, and customer support. The robust support for HTTP/HTTPS protocols and advanced routing features enable seamless handling of microservices architecture, allowing the booking platform to scale elastically. Low-latency performance ensures quick response times during peak booking periods, such as holiday seasons or flash sales, while its request-based pricing model provides cost-efficiency. Additionally, strong sticky session support is required to ensure consistent user experiences by maintaining client connection states during critical processes like multi-step reservation flows, preventing potential booking interruptions.

**2. Choosing a Load Balancer Algorithm**

**Should we have a Round Robin or LOR Algorithm?**

The load balancing algorithm is a critical component of the hotel booking platform's architecture, balancing competing technical requirements. It optimizes traffic distribution to ensure efficient tenant identification while minimizing impact on microservice efficiency. By dynamically adjusting load distribution, the algorithm enhances system scalability (TR 1.2) and elasticity (TR 1.3) while minimizing monitoring overhead (TR 1.4). It also supports cost-efficient data replication (TR 11.1) and quick data retrieval while maintaining low latency (TR 5.1). Additionally, it balances encryption processing with latency and authentication requirements. Ultimately, the algorithm's ability to distribute workloads across the entire system architecture contributes to the platform's overall performance and user experience.

**Feature Based Comparison:**

[(n) indicates rank for that alternative(top row) against the particular objective mentioned in the left column]

| **Objective** | **Least Outstanding Requests** | **Round Robin** |
| --- | --- | --- |
| **Load Distribution** | Routes to servers with the fewest active requests, balancing real-time load dynamically. (1) | Distributes requests equally without assessing current server conditions. (2) |
| **Performance** | Improves efficiency by minimizing response time under varying workloads. (1) | May lead to delays if some servers face higher processing times or traffic spikes. (2) |
| **Complexity** | Requires tracking active requests per server, making it computationally more complex (2) | Straightforward implementation; minimal computation overhead. (1) |
| **Suitability for Workloads** | Best for unpredictable workloads, such as surges during flash sales or peak travel seasons. (1) | Works for evenly distributed traffic but struggles with sudden demand variations. (2) |
| **Adaptability** | Dynamically reallocates traffic to less-burdened servers for faster responses. (1) | Static allocation fails to adapt to variations in server states or processing times. (2) |

### **Even Swaps Method for Choosing LOR vs. Round Robin as Load Balancing Algorithms**

### **Step 1: Initial Comparison**

We start by comparing the two algorithms based on critical TRs affected by the decision: **latency (TR 5.1)**, **scalability (TR 1.2)**, **monitoring overhead (TR 1.4)**, **cost efficiency (TR 11.1)**, and **elasticity (TR 1.3)**.

| **Algorithm** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** |
| --- | --- | --- | --- | --- | --- |
| **LOR** | Low | High | High | Medium | Medium |
| **Round Robin** | Medium | Medium | Low | High | High |

### **Step 2: LOR vs. RR (Latency and Scalability)**

**Latency (TR 5.1):**LOR minimizes latency by directing requests to the backend server with the fewest outstanding tasks. This ensures that under-utilized servers are prioritized, avoiding bottlenecks. Conversely, RR simply assigns requests cyclically, regardless of server load. **LOR wins.**

**Scalability (TR 1.2):**LOR dynamically adapts to uneven loads, making it highly scalable for workloads with unpredictable spikes. RR, while simple, does not account for varying server capacities or response times, leading to suboptimal performance in scaling scenarios. **LOR wins.**

| **Algorithm** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** |
| --- | --- | --- | --- | --- | --- |
| **LOR** | **Low** | **High** | High | Medium | Medium |
| **Round Robin** | Medium | Medium | Low | High | High |

**Outcome:**RR remains in consideration due to its lower monitoring overhead and cost efficiency.

### **Step 3: Monitoring Overhead and Cost Efficiency**

**Monitoring Overhead (TR 1.4):**LOR requires monitoring server states to evaluate outstanding requests, increasing resource overhead. RR does not monitor server states, significantly reducing overhead. **RR wins.**

**Cost Efficiency (TR 11.1):**The simplicity of RR minimizes operational costs, as it does not require additional computational resources for monitoring. LOR incurs higher costs due to its complex monitoring logic. **RR wins.**

| **Algorithm** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** |
| --- | --- | --- | --- | --- | --- |
| **LOR** | **Low** | **High** | High | Medium | Medium |
| **Round Robin** | Medium | Medium | **Low** | **High** | High |

**Outcome:**RR is competitive due to its simplicity and lower resource requirements. LOR still leads on performance-oriented metrics (latency and scalability).

### **Step 4: Elasticity (TR 1.3)**

**Elasticity (TR 1.3):**RR’s simplicity allows it to adapt more predictably during rapid scaling events. LOR, while adaptable, may experience delays in recalculating outstanding requests during high-traffic spikes, potentially affecting elasticity. **RR wins.**

| **Algorithm** | **Latency (TR 5.1)** | **Scalability (TR 1.2)** | **Monitoring Overhead (TR 1.4)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** |
| --- | --- | --- | --- | --- | --- |
| **LOR** | **Low** | **High** | High | Medium | Medium |
| **Round Robin** | Medium | Medium | **Low** | **High** | **High** |

**Outcome:**RR edges out LOR in elasticity, making it a strong choice for workloads requiring rapid scaling.

### **Step 5: Resolving Dominance**

LOR and RR exhibit clear strengths in different areas:

* **LOR** is optimal for latency-sensitive applications where performance consistency is critical.
* **RR** is preferable for cost-sensitive environments with predictable traffic patterns.

| **Metric** | **Winner** | **Reason** |
| --- | --- | --- |
| **Latency (TR 5.1)** | LOR | Minimizes latency by prioritizing servers with fewer outstanding requests. |
| **Scalability (TR 1.2)** | LOR | Adapts dynamically to varying server loads. |
| **Monitoring Overhead (TR 1.4)** | RR | No server-state monitoring reduces overhead. |
| **Cost Efficiency (TR 11.1)** | RR | Simplicity minimizes operational costs. |
| **Elasticity (TR 1.3)** | RR | Rapid scaling without delays caused by monitoring computations. |

**Final Selection:**

From our comparison we see that Least Outstanding Requests (LOR) is the best suited solution for our application. Using the Least Outstanding Requests (LOR) load balancing algorithm is crucial for a hotel booking system due to its ability to handle dynamic and unpredictable workloads effectively. Unlike Round Robin, which evenly distributes traffic without considering server conditions, LOR routes requests to servers with the fewest active requests, ensuring real-time load balancing. This dynamic approach minimizes response times and enhances performance, particularly during flash sales or peak travel seasons when traffic surges are common. While LOR requires tracking active requests, adding computational complexity, it is well worth the trade-off for its adaptability and efficiency under varying workloads. By reallocating traffic to less-burdened servers, LOR ensures consistent user experience, which is vital for maintaining customer satisfaction in high-demand scenarios.

**3. Microservice vs Monolith Architecture**

**Which is the best for our performance needs?**

The choice of architecture between microservices and monolithic is paramount for a highly scalable hotel booking system. A well-considered architectural decision directly impacts the system's ability to meet critical technical requirements such as scalability (TR 1.2, 1.3), latency (TR 5.1), data isolation (TR 17.1), cost efficiency (TR 11.1), and monitoring overhead (TR 1.4). A microservices architecture, with its fine-grained scalability, independent deployment, and modular design, can offer significant advantages in these areas. However, it requires careful consideration of factors like increased complexity and operational overhead. A monolithic architecture, while simpler to develop and deploy initially, may struggle to meet the demands of a growing and evolving hotel booking system. Ultimately, the optimal architectural choice depends on the specific needs, constraints, and long-term vision of the organization.

**Feature Based Comparison:**

[(n) indicates rank for that alternative(top row) against the particular objective mentioned in the left column]

| **Objective** | **Microservices Architecture** | **Monolithic Architecture** |
| --- | --- | --- |
| **Scalability** | Independent scaling of services (e.g., payment, booking, support)Independent scaling of services (e.g., payment, booking, support). (1) | Entire system must scale together, increasing costs (2) |
| **Deployment** | Independent deployment of individual services (1) | Single codebase, requires redeployment of the entire system (2) |
| **Fault Isolation** | Issues in one service (e.g., payments) don't affect others (1) | A failure in one component can impact the entire system (2) |
| **Complexity** | Requires advanced orchestration and management tools (e.g., Kubernetes). (2) | Simpler to develop and maintain for smaller teams. (1) |
| **Performance** | Optimized for specific services, enabling faster response times (1) | Performance may degrade as the application grows (2) |

### **Even Swaps Method for Choosing Microservices vs. Monolith Architecture for a Scalable Hotel Booking System**

### **Step 1: Initial Comparison**

| **Architecture** | **Scalability (TR 1.2)** | **Latency (TR 5.1)** | **Data Isolation (TR 17.1)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** | **Monitoring Overhead (TR 1.4)** |
| --- | --- | --- | --- | --- | --- | --- |
| Microservices | High | Medium | High | Medium | High | High |
| Monolith | Medium | High | Medium | High | Medium | Low |

### **Step 2: Scalability vs. Latency**

**Scalability (TR 1.2):**Microservices excel in scalability due to independent deployment and scaling of services, allowing the system to handle fluctuating workloads efficiently. Monoliths, being a single codebase, scale less efficiently because scaling often involves duplicating the entire application. **Microservices win.**

**Latency (TR 5.1):**Monoliths offer lower latency due to direct communication between components within a single process. In contrast, microservices rely on network communication, which increases latency. **Monolith wins.**

| **Architecture** | **Scalability (TR 1.2)** | **Latency (TR 5.1)** | **Data Isolation (TR 17.1)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** | **Monitoring Overhead (TR 1.4)** |
| --- | --- | --- | --- | --- | --- | --- |
| **Microservices** | **High** | **Medium** | **High** | **Medium** | **High** | **High** |
| **Monolith** | **Medium** | **High** | **Medium** | **High** | **Medium** | **Low** |

### **Step 3: Data Isolation vs. Cost Efficiency**

**Data Isolation (TR 17.1):**Microservices inherently support data isolation, as each service manages its own database. This architecture aligns with regulatory and business requirements for data separation. Monoliths, with shared databases, are less suited for strict data isolation requirements. **Microservices win.**

**Cost Efficiency (TR 11.1):**Monoliths are cost-effective due to simpler infrastructure and lower operational overhead. Microservices require additional resources for orchestration, inter-service communication, and monitoring, increasing costs. **Monolith wins.**

| **Architecture** | **Scalability (TR 1.2)** | **Latency (TR 5.1)** | **Data Isolation (TR 17.1)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** | **Monitoring Overhead (TR 1.4)** |
| --- | --- | --- | --- | --- | --- | --- |
| **Microservices** | **High** | **Medium** | **High** | **Medium** | **High** | **High** |
| **Monolith** | **Medium** | **High** | **Medium** | **High** | **Medium** | **Low** |

### **Step 4: Elasticity vs. Monitoring Overhead**

**Elasticity (TR 1.3):**Microservices excel in elasticity due to their ability to scale specific services independently, enabling fine-grained resource allocation. Monoliths, while scalable, cannot scale individual components separately, reducing elasticity. **Microservices win.**

**Monitoring Overhead (TR 1.4):**Microservices introduce significant monitoring overhead, as each service must be tracked independently for health, performance, and reliability. Monoliths, with fewer components, have simpler monitoring needs. **Monolith wins.**

| **Architecture** | **Scalability (TR 1.2)** | **Latency (TR 5.1)** | **Data Isolation (TR 17.1)** | **Cost Efficiency (TR 11.1)** | **Elasticity (TR 1.3)** | **Monitoring Overhead (TR 1.4)** |
| --- | --- | --- | --- | --- | --- | --- |
| **Microservices** | **High** | **Medium** | **High** | **Medium** | **High** | **High** |
| **Monolith** | **Medium** | **High** | **Medium** | **High** | **Medium** | **Low** |

### **Step 5: Resolving Dominance**

The even swaps reveal clear trade-offs between the two architectures:

* **Microservices** dominate on scalability, data isolation, and elasticity, making them ideal for systems requiring high adaptability, compliance, and granular resource allocation.
* **Monoliths** dominate on latency, cost efficiency, and monitoring simplicity, making them suitable for environments with predictable workloads and budget constraints.

| **Metric** | **Winner** | **Reason** |
| --- | --- | --- |
| Scalability (TR 1.2) | Microservices | Independent scaling of services enhances adaptability. |
| Latency (TR 5.1) | Monolith | Direct in-process communication minimizes latency. |
| Data Isolation (TR 17.1) | Microservices | Service-specific databases ensure better data isolation. |
| Cost Efficiency (TR 11.1) | Monolith | Lower infrastructure and operational costs. |
| Elasticity (TR 1.3) | Microservices | Fine-grained scaling improves resource utilization during demand surges. |
| Monitoring Overhead (TR 1.4) | Monolith | Simpler architecture reduces monitoring complexity. |

**Final Selection:**

A microservices architecture is better suited for a hotel booking system because it enables independent scaling and optimization of critical workloads like room availability searches, payment processing, and customer support. Each service can use its own technology stack and scale based on traffic demands, improving system efficiency during peak periods like holidays. Fault isolation ensures that a failure in one service does not disrupt others, preserving functionality. Additionally, microservices facilitate quicker updates and integrations, essential for a dynamic system handling frequent promotions or third-party integrations. Containers and services like Amazon EKS (Elastic Kubernetes Service) further enhance this flexibility by enabling the rapid deployment and scaling of microservices in response to traffic spikes.

For further discussions we stick to only a feature based comparison for our design choices:

4. AWS RDS vs Aurora DB

Which DB exactly supports our needs?

The strategic selection of a database is essential to meet the demands of the hotel booking application, considering factors such as faster access, performance, reliability, and cost efficiency. For **performance**, a database must support low-latency queries (TR 5.1) and handle high transactional workloads, particularly for real-time booking and payment processing. Reliability (TR 2.1, TR 11.1) requires features like data replication and fault tolerance to ensure availability during peak loads.

A database should support scalable architectures (TR 1.2, TR 13.1), allowing dynamic resource allocation based on demand, and provide efficient indexing for faster access. Cost optimization (TR 4.2, TR 11.2) is critical, particularly for archiving inactive tenant data and minimizing storage costs. Security (TR 3.1-3.4, TR 8.1) is also a priority, requiring encryption and compliance with regional laws to protect tenant information. Choosing a suitable database is fundamental to balancing these requirements while maintaining application efficiency and reliability.

[(n) indicates rank for that alternative(top row) against the particular objective mentioned in the left column]

| **Objective** | **AWS RDS** | **Aurora DB** |
| --- | --- | --- |
| **Performance** | Suitable for general-purpose applications with moderate read/write workloads. (2) | Optimized for high-performance, low-latency workloads, handling millions of transactions per second. (1) |
| **Cost** | Lower cost; billed per instance and storage usage. (1) | Higher cost but more cost-efficient for high-performance workloads. (2) |
| **Startup Time** | Faster deployment for simpler setups. (1) | Slightly more time-consuming due to its advanced configuration options. (2) |
| **Engine Support** | Supports multiple engines (MySQL, PostgreSQL, Oracle, SQL Server). (1)   |  | | --- | | Supports MySQL and PostgreSQL-compatible engines with Aurora-specific optimizations. (2) |
| **High Availability** | Multi-AZ deployments for failover support. (Tie) | Designed for high availability with up to 15 read replicas and automatic failover. (Tie) |

In a hotel booking system, fine-grained control and cost considerations take precedence because workloads can often be estimated based on previous usage patterns, such as seasonal peaks or booking trends. With predictable traffic, RDS offers sufficient performance without the added expense of Aurora DB, which is optimized for unpredictable and high-volume workloads. Fine-grained control allows for tailored configurations to meet specific operational needs in a hotel booking system. For example, during predictable seasonal peaks like holidays, database administrators can adjust RDS configurations to allocate more resources for query caching and optimize connection pooling. Similarly, if the booking system requires compliance with specific data retention laws in different regions, administrators can customize backup schedules and retention policies accordingly.

5. Is Encryption at rest really required?

The need for encryption at rest in the hotel booking application is paramount to ensuring data security and compliance. For tenant data isolation (TR 25.1, TR 17.1), encryption at rest safeguards customer details and sensitive information stored in the database, preventing unauthorized access even if the storage medium is compromised. It supports regional compliance requirements (TR 3.4) and aligns with global standards for data protection (TR 3.1, TR 3.2, TR 3.3). Encryption at rest secures archived tenant data (TR 4.2, TR 11.2), balancing cost-efficient storage with robust security measures. By encrypting all data at rest, the platform ensures that tenant trust, legal compliance, and the overall security posture of the application are maintained.

| **Objective** | **Encryption at Rest** | **No Encryption** |
| --- | --- | --- |
| **Data Security** | Ensures that sensitive data (e.g., guest information, payment details) is encrypted when stored, reducing the risk of data breaches. (1) | Vulnerable to data theft if unauthorized access occurs to the storage. (2) |
| **Compliance** | Meets industry standards and regulations (e.g., PCI DSS for payments). Essential for handling personal and payment data. (1) | May not meet compliance requirements for data protection laws (e.g., GDPR). (2) |
| **Performance** | Slightly impacts performance due to the overhead of encryption and decryption processes. (2) | Faster read/write performance as no encryption is applied. (1) |
| **Scalability** | Suitable for large-scale systems where sensitive data is stored across multiple databases or services, ensuring consistent encryption across all data. (1) | May be easier to scale due to less processing overhead, but risks increase. (2) |
| **Cost** | Typically incurs additional costs for encryption management, key management, and storage overhead. (2) | Lower operational costs as no encryption or key management is needed (1) |

Encryption at rest is crucial for a hotel booking system because it safeguards sensitive guest information, such as personal details, payment data, and booking histories, which are stored across various databases. This encryption helps mitigate the risk of data breaches, ensuring compliance with regulatory requirements like PCI DSS and GDPR. Given the system's reliance on secure transactions and customer trust, encryption at rest provides essential data protection while maintaining confidentiality. Although it may slightly impact performance and incur extra costs, the benefits in security, compliance, and customer confidence far outweigh these drawbacks.

6. Which Deployment Strategy should be used?

The strategic selection of a deployment strategy is vital for ensuring the architectural, operational, and business needs of the hotel booking application are met. We want to select a strategy that is effective, optimizes cost and also does not affect the user experience while using the application.

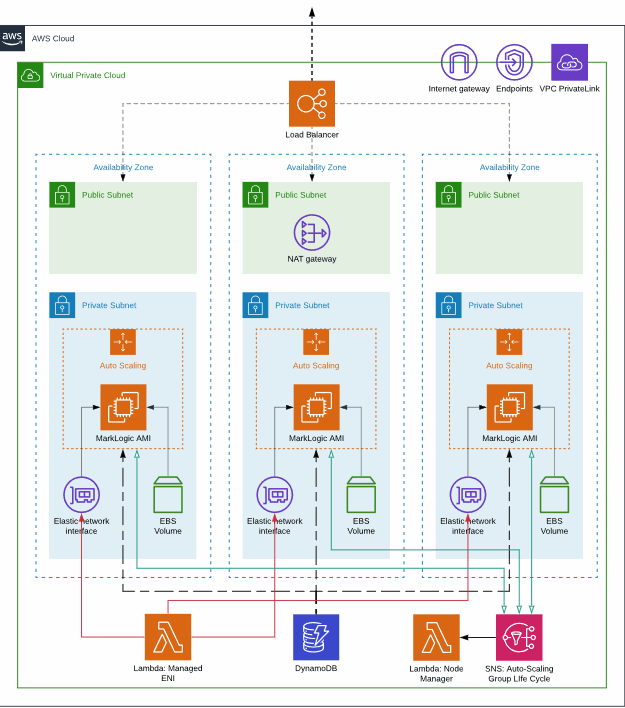
[(n) indicates rank for that alternative(top row) against the particular objective mentioned in the left column]

| **Objective** | **Blue Green Deployment** | **Canary Deployment** | **Multiservice Deployment** |
| --- | --- | --- | --- |
| **Use case** | Suitable for critical systems requiring minimal downtime, such as the booking flow and payment systems. (Tie) | Ideal for testing new features or updates on live traffic without impacting the entire user base. (Tie) | Ideal for testing new features or updates on live traffic without impacting the entire user base. (Tie) |
| **Downtime** | Near-zero downtime, as the traffic is shifted instantly from blue to green. (1) | Negligible downtime, as only a subset of users experiences the changes initially. (2) | Service-level downtime, but the rest of the system remains unaffected. (3) |
| **Risk Mitigation** | Full rollback to the blue environment is quick and straightforward if issues are detected. (1) | Issues are contained within the canary audience, minimizing impact on the majority of users. (2) | Failures in one microservice don't affect others, but rollback or fixes can require service-specific handling. (3) |
| **Resource utilization** | High resource usage during deployment as both environments (blue and green) run simultaneously.(3) | Moderate resource usage, as only a small percentage of traffic is redirected to the canary version. (2) | Minimal additional resources are required; focuses on the service being updated.(1) |
| **Impact on User Experience** | Minimal, as users experience an immediate switch with no interruptions. (1) | Users may see different versions depending on the rollout phase, but the majority remain unaffected initially. (2) | Isolated to users interacting with the updated service; the rest of the application is unaffected. (3) |
| **Cost** | Higher cost due to duplication of environments during deployment. (3) | Moderate cost, as resources are only required for the canary deployment fraction. (2) | Cost-effective for microservice architectures but may involve overhead for managing dependencies. (1) |
| **Rollback Complexity** | Simple rollback by reverting traffic to the blue environment. (1) | Incremental rollback involves reducing canary traffic gradually. (1) | Rollback is service-specific and may impact inter-service dependencies if not handled carefully. (2) |  |

Canary deployment can be highly suitable for a hotel booking system using a microservice architecture due to its gradual, controlled rollout of new updates. By deploying changes incrementally to a small subset of users, the system minimizes risks of service disruption or booking errors, which are critical for maintaining customer trust and ensuring uninterrupted operations. The ability to monitor and validate updates on a small scale allows quick detection of potential issues, such as bugs in the booking, payment, or notification services. Additionally, the strategy helps balance performance and cost, as only a portion of traffic is redirected to the new version, requiring fewer resources than maintaining parallel environments in Blue-Green deployment. This makes Canary deployment particularly effective for dynamic systems like hotel booking platforms.

## 5.7 Discussion of an alternate design

The alternate design [24]. we explored for our hotel booking application leverages AWS tools, as depicted in the CloudFormation diagram given below:



This design includes the following **flipped** choices:

1. AWS NLB as the load balancer
2. Monolithic Implementation
3. Round Robin as the LB Algorithm

**Where would this architecture work?**

This architecture is well-suited for a hotel booking application due to its simplicity and cost-effectiveness. The use of Auto Scaling Groups ensures elasticity, dynamically adjusting resources based on varying traffic, which is crucial for handling seasonal or peak booking periods. The deployment across multiple Availability Zones (AZs), combined with an Elastic Load Balancer, provides high availability and fault tolerance, ensuring uninterrupted service during hardware or network failures. Cost optimization is achieved through the efficient use of NAT Gateways and DynamoDB for handling transient data like session management and real-time booking updates. Additionally, the reliance on managed services like Lambda for workflows and SNS for notifications eliminates the need for dedicated servers, further minimizing infrastructure expenses.

**Why not pursued:**

Replacing AWS ALB with AWS NLB and adopting a monolithic architecture introduces significant challenges across performance, security, scalability, and global reach. AWS NLB, operating at Layer 4, lacks ALB’s advanced routing capabilities, limiting traffic management and making the system inefficient at handling dynamic workloads. Coupled with a monolithic design, scaling becomes cumbersome as the entire stack must be replicated, violating elasticity requirements and increasing response times under high traffic. The round-robin load-balancing algorithm does not account for instance health, leading to overloaded or degraded instances, further impacting performance.

Security is notably compromised due to the absence of AWS WAF, leaving the system vulnerable to common web attacks. The lack of explicit IAM configurations and encryption measures increases the risk of data breaches and non-compliance with regional security standards. For a booking system handling sensitive customer data, such vulnerabilities are unacceptable. Monitoring is another critical gap, as the absence of tools like CloudWatch limits the ability to track performance and detect failures, reducing reliability and availability.

The reliance on AMIs instead of modern containerized solutions like ECS or EKS hinders scalability and flexibility, increasing operational costs and violating cost optimization goals. Monolithic architecture also limits modularity, making it difficult to optimize workload-specific performance. Additionally, deploying the system in a single region without leveraging AWS Global Accelerator or CloudFront results in higher latency for international users, degrading the experience for a global audience.

Cost inefficiencies further arise from scaling entire applications during peak loads and storing inactive data without archiving mechanisms like S3 Glacier. The centralized nature of the monolithic architecture also increases the risk of cascading failures, reducing reliability and making data replication more challenging. These limitations collectively hinder the system’s ability to meet critical technical requirements, significantly impacting its effectiveness as a scalable, secure, and user-friendly hotel booking platform.

# 6 Kubernetes experimentation

In this section, you’ll validate some aspects of your design experimentally. Which ones are your choice, but you must discuss with the instructors first. The idea is to limit the scope of the experiment runs and required analysis.

## 6.1 Experiment Design

**Technical Requirements (TR)**

Validate the Horizontal Pod Autoscaler's (HPA) ability to dynamically scale pods based on CPU utilization and confirm the system can handle sudden increases in concurrent user load

The primary objective of this experiment is to validate the dynamic scaling capabilities of a Kubernetes Horizontal Pod Autoscaler (HPA) for a PHP Apache web server under simulated load conditions.

Describe in detail the experiment that will validate your design. More specifically, define the configuration of the environment, the inputs and the expected outputs.

**Experiment Objectives**

Demonstrate HPA's ability to scale pods based on CPU utilization

Verify the responsiveness of the autoscaling mechanism

Analyze the system's behavior under concurrent user load

**Environment Configuration**

**Kubernetes Deployment:**

Platform: Kubernetes Cluster

Application: PHP Apache Web Server

**Deployment Configuration:**

apiVersion: apps/v1

kind: Deployment

metadata:

name: php-apache

spec:

selector:

matchLabels:

run: php-apache

template:

metadata:

labels:

run: php-apache

spec:

containers:

- name: php-apache

image: registry.k8s.io/hpa-example

ports:

- containerPort: 80

resources:

limits:

cpu: 500m

requests:

cpu: 200m

**HPA Configuration:**

kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10

CPU Utilization Threshold: 50%

Minimum Pods: 1

Maximum Pods: 10

### **Input Parameters**

**Load Testing Tool:** Locust

* Concurrent Users: 500
* User Spawn Rate: 20 users/second
* Test Duration: Continuous load
* Request Type: GET requests to root endpoint

## **Locust Script:**

from locust import HttpUser, task, between

class PhpApacheUser(HttpUser):

wait\_time = between(1, 2)

@task

def load\_test(self):

self.client.get("/")

### **Expected Outcomes**

## Dynamic pod scaling in response to increased load

## Maintaining CPU utilization near the 50% threshold

## Scaling within the defined minimum and maximum pod limits

## 6.2 Workload generation with Locust

Describe how you intend to use Locust to create (all or some of) the inputs. Discuss with the instructors use of other workload generators.

The load testing strategy is designed to create a realistic and nuanced simulation of user interactions with the web application. By implementing a gradual ramp-up of concurrent users, the test avoids abrupt spikes in traffic that might not reflect real-world scenarios. The approach incorporates a consistent request pattern to ensure uniform load distribution, while introducing randomized think times between requests to authentically mimic human browsing behavior. This methodology allows for a more accurate representation of how the system might perform under actual user conditions, providing insights into the application's scalability, performance, and responsiveness across different load intensities. The strategy ultimately aims to stress test the infrastructure in a way that closely parallels genuine user engagement, enabling more meaningful performance analysis and capacity planning.

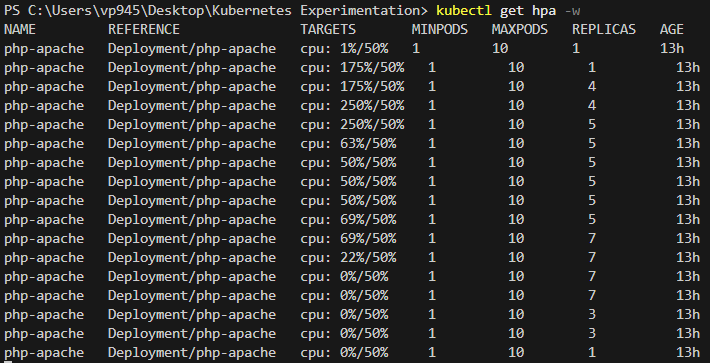
## 6.3 Analysis of the results

Add the screenshots with the explanations.

Analyze the outputs you observed. Why do they validate the design?

**HPA Scaling Observations**

**HPA Output Analysis:**

****

**Scaling Mechanism Insights**

**Initial State:**

Started with 1 replica

Low initial CPU utilization

**Load Impact:**

CPU utilization rapidly increased to 175-250%

Triggered immediate scaling response

Scaled from 1 to 5 replicas

**Stabilization:**

Converged around 50-63% CPU utilization

Maintained 5 replicas

Demonstrated effective load distribution

**Design Validation**

**Key Validation Points:**

HPA successfully detected increased load

Dynamically scaled pods within configured limits

Maintained performance under concurrent user load

Achieved near-target CPU utilization

**Limitations and Potential Improvements:**

Consider more granular resource requests

Implement more diverse load testing scenarios

Monitor network and memory metrics alongside CPU

**Conclusion**

The experiment successfully demonstrated Kubernetes HPA's ability to dynamically scale a PHP Apache deployment under simulated user load, validating the auto-scaling design and its responsiveness.

# Section 7

skipped

# Section 8

skipped

# 9 Comparisons

In this section, you will provide a comparison between two solutions, approaches, tools, current trends, etc. The comparison can be theoretical or experimental.

Here are some suggestions:

1. Facebook’s Katran and another LB algorithm

**Overview of Katran and AWS NLB**

Facebook’s **Katran** [26] is an open-source, high-performance load balancing framework that operates at Layer 4 (Transport Layer) using eBPF (Extended Berkeley Packet Filter) [27] technology. It is designed for environments where low latency, high throughput, and deep customizability are required. On the other hand, **AWS Network Load Balancer (NLB)** [25] is a fully managed Layer 4 load balancer provided by AWS. It is built for cloud-native applications, offering automatic scaling, seamless integration with AWS services, and robust performance out of the box.

**Scalability**

Katran provides high scalability through its efficient use of eBPF and distributed architecture. However, scaling with Katran requires manual intervention and tuning, as there are no native automated scaling capabilities. In contrast, AWS NLB scales automatically based on traffic patterns, making it highly suitable for dynamic workloads like a hotel booking system, where traffic can spike unpredictably during events like promotional sales or peak holiday seasons.

**Performance**

Both Katran and AWS NLB offer excellent performance at Layer 4, with low latency and high throughput. Katran achieves this through its lightweight packet filtering capabilities via eBPF, while NLB is optimized for cloud environments and can handle millions of requests per second. For the hotel booking system, where seamless user interactions are critical, AWS NLB’s managed performance capabilities provide an edge, as it ensures consistency without additional configuration overhead.

**Ease of Management**

Katran, being an open-source tool, requires skilled personnel for deployment, configuration, and ongoing maintenance. This includes kernel-level tuning and managing the infrastructure. In contrast, AWS NLB is fully managed, requiring minimal effort to set up and maintain. For the hotel booking system, where operational simplicity and efficiency are paramount, AWS NLB is the more practical choice as it offloads the burden of infrastructure management.

**Integration with the Ecosystem**

Katran is best suited for custom-built environments where deep integration with proprietary infrastructure is required. However, it lacks native integration with third-party tools and platforms. AWS NLB, on the other hand, seamlessly integrates with the AWS ecosystem, including services like Auto Scaling, Route 53 for DNS resolution, and CloudWatch for monitoring. Given the hotel booking system’s reliance on these AWS services (as seen in the architecture), AWS NLB fits naturally into the existing workflow.

**Monitoring and Debugging**

Katran requires custom tools and setups for monitoring traffic patterns and diagnosing issues. AWS NLB provides built-in monitoring capabilities through CloudWatch, offering detailed metrics and logs for debugging. For the hotel booking system, where monitoring microservices and ensuring traffic flow consistency are essential, NLB simplifies the process with its native tools.

**Cost**

Katran is free and open-source, making it attractive from a licensing perspective. However, the operational costs of maintaining and scaling the infrastructure can be significant. AWS NLB operates on a pay-as-you-go pricing model, allowing for predictable cost management based on traffic. This aligns with the hotel booking system’s need for elasticity and cost-efficiency in handling variable traffic loads.

**Resiliency and High Availability**

Katran’s resilience depends on custom failover mechanisms, which require significant effort to implement and maintain. AWS NLB offers built-in redundancy across multiple availability zones (AZs), ensuring high availability without additional configuration. For the hotel booking system, which demands uninterrupted service and fault tolerance, AWS NLB’s multi-AZ failover capabilities are indispensable.

**Security**

Katran does not offer built-in security features and relies on external tools for DDoS mitigation and traffic encryption. AWS NLB is integrated with AWS Shield for DDoS protection and supports TLS termination, ensuring secure traffic handling. Additionally, NLB complements the architecture’s existing security components like WAF and KMS, further enhancing the overall security posture of the system.

**Customizability**

Katran is highly customizable, making it suitable for organizations like Facebook that require unique solutions tailored to their specific infrastructure. However, this level of customizability is unnecessary for the hotel booking system, which benefits more from AWS NLB’s standardized and managed approach.

**Current Trends**

Katran is popular in specialized environments like Facebook, where massive scale and unique requirements justify the investment in custom tools. AWS NLB, on the other hand, is widely adopted in cloud-native applications due to its simplicity, scalability, and robust integration with AWS services.

### **Choosing for our Hotel Booking System**

For our Hotel Booking System’s reliance on a cloud-native architecture with AWS services like CloudWatch (**TR1.4**), WAF (**TR3.6**), and Auto Scaling (**TR1.2**, **TR1.5**), **AWS NLB is the better choice**. It provides the scalability, reliability (**TR2.1**), and ease of management required for a dynamic workload, while also ensuring high availability and robust security (**TR3.1**, **TR3.2**, **TR3.3**, **TR8.1**). AWS NLB’s seamless integration with global routing policies (**TR2.2**) and multi-region deployment (**TR5.1**) further ensures latency reduction and performance efficiency. Katran, while powerful, would introduce unnecessary complexity and operational overhead for a system that can efficiently leverage AWS’s managed services. By aligning with AWS’s scaling algorithms (**TR20.1**) and tenant data isolation strategies (**TR17.1**), AWS NLB ensures that the system achieves its goals of simplicity, cost-efficiency, and seamless integration, making it the optimal solution for the hotel booking system.

However, Katran may be more suitable in cases where ultra-low latency and highly customized traffic management are critical, such as handling specific routing rules for VIP customers or regions, or during high-demand events like flash sales. Its use of eBPF allows for fine-grained control over packet filtering and high throughput, which could outperform AWS NLB in certain scenarios. Additionally, for businesses with in-house networking expertise, Katran can offer cost savings by avoiding AWS NLB’s pay-per-request pricing model, especially in private cloud environments. However, for the hotel booking system’s global scale, operational simplicity, and fast deployment, AWS NLB remains the better choice due to its seamless integration with AWS services, automatic scalability, and minimal maintenance overhead.

2. CloudFormation and Terraform

### **Overview of CloudFormation and Terraform**

AWS **CloudFormation** [31] is a native Infrastructure as Code (IaC) service provided by AWS to define, provision, and manage AWS resources using JSON or YAML templates. It integrates deeply with AWS services and provides managed capabilities for resource orchestration. **Terraform** [32], developed by HashiCorp, is a widely used open-source IaC tool that allows users to provision and manage infrastructure across multiple cloud providers (AWS, Azure, Google Cloud, and more) using its declarative HashiCorp Configuration Language (HCL).

### **Ease of Use**

CloudFormation offers tight integration with AWS, making it highly intuitive for users who are familiar with AWS services. Its templates are straightforward for AWS-specific use cases, as seen in the hotel booking system, which relies heavily on services like Route 53, WAF, RDS, and S3. On the other hand, Terraform provides greater flexibility but comes with a learning curve due to its proprietary HCL syntax. While Terraform is slightly more complex to start with, it excels when managing hybrid or multi-cloud environments, which is unnecessary for this single-cloud architecture.

### **Support for AWS Services**

CloudFormation has 100% coverage of AWS services and features, including new releases, ensuring that the hotel booking system can take full advantage of the latest AWS functionalities as they become available. In contrast, Terraform’s AWS provider lags behind AWS’s rapid updates, though the community quickly updates its modules to bridge the gap. For the hotel booking system, which is entirely hosted within AWS, CloudFormation’s native service coverage offers a significant advantage.

### **Portability**

Terraform is cloud-agnostic and supports multiple providers, enabling infrastructure provisioning across diverse environments. This is ideal for organizations considering multi-cloud or on-premises deployments. However, for the hotel booking system, which is solely built on AWS, portability is not a critical factor. CloudFormation, while AWS-specific, aligns perfectly with the architecture and eliminates unnecessary complexity.

### **State Management**

CloudFormation automatically handles state management, storing the current state of the infrastructure directly within the AWS ecosystem. This reduces operational overhead and ensures synchronization with deployed resources. In contrast, Terraform requires external state management, typically stored in an S3 bucket or other secure storage. While Terraform’s approach is powerful and customizable, CloudFormation’s managed state is better suited for the hotel booking system, simplifying operations and reducing the risk of state drift.

### **Modularity and Reusability**

Terraform excels in modularity and code reuse. It allows the creation of reusable modules that can be shared across teams or projects, offering excellent maintainability for large infrastructures. While CloudFormation supports modularization through nested stacks, it lacks Terraform’s flexibility in this regard. For the hotel booking system, where the infrastructure is AWS-specific and relatively contained, CloudFormation’s nested stacks and standard templates provide sufficient modularity without the complexity of Terraform.

### **Community and Ecosystem**

Terraform boasts a large and active community, offering pre built modules for common use cases. This can reduce development time for teams unfamiliar with specific resources. CloudFormation’s ecosystem is less vibrant in terms of community-contributed templates but is supplemented by AWS-managed features like the AWS CloudFormation Registry. For the hotel booking system, the reliance on standardized AWS services reduces the need for external modules, making CloudFormation a more seamless choice.

### **Error Handling and Debugging**

Terraform provides a more transparent debugging process, offering detailed logs and output during the plan and application phases. This makes it easier to identify and resolve configuration issues. CloudFormation, while improving over time, has more opaque error messages, which can sometimes slow down troubleshooting. However, AWS-specific error handling tools like AWS CloudFormation Drift Detection mitigate this limitation and work well for systems like the hotel booking platform.

### **Automation and CI/CD Integration**

Both CloudFormation and Terraform integrate well with CI/CD pipelines. CloudFormation works natively with AWS CodePipeline and CodeBuild, which are already part of the architecture shown in the hotel booking system. Terraform, while requiring third-party tools or additional configuration, supports integration with platforms like Jenkins, GitLab CI/CD, and GitHub Actions. Given the AWS-centric design of the hotel booking system, CloudFormation provides a more straightforward integration path.

### **Security**

CloudFormation is inherently secure within AWS, leveraging IAM roles and policies to manage access and operations. Additionally, it integrates seamlessly with AWS KMS for encrypting sensitive data. Terraform, while also secure, relies on external configurations for IAM and encryption, requiring more setup effort. For the hotel booking system, where security is critical (e.g., compliance with TR3.3 encryption and TR3.4 regional compliance), CloudFormation simplifies security configuration by staying native to AWS.

### **Cost**

CloudFormation is free to use; users only pay for the underlying AWS resources they provision. Terraform is also free as an open-source tool, but operational costs can arise from managing its state files (e.g., storing them in S3) and configuring its environment. For the hotel booking system, where cost management is a key consideration, CloudFormation’s built-in capabilities minimize operational overhead.

### **Resilience and Change Management**

CloudFormation handles changes using Change Sets, allowing users to preview infrastructure modifications before applying them. This reduces the risk of unintentional disruptions in production environments, such as the hotel booking system’s live user-facing services. Terraform’s plan and apply workflow offers similar functionality but is more manual compared to CloudFormation’s AWS-native automation.

### **Choosing for our Hotel Booking System**

In the hotel booking system and its reliance on AWS-native services, **AWS CloudFormation** is the recommended IaC tool as it aligns closely with the outlined TRs. For **tenant isolation (TR25.1, TR17.1)**, CloudFormation can provision isolated VPCs, IAM roles, and Kubernetes clusters to securely separate tenant environments. It ensures robust **monitoring (TR1.4)** by automating CloudWatch metrics and alarms with optimal frequencies and thresholds. To enhance **performance efficiency (TR5.1, TR2.2)**, CloudFormation supports deploying global infrastructure components like Route 53, CloudFront, and Global Accelerator to reduce latency and manage global traffic. Kubernetes clusters provisioned through CloudFormation ensure elasticity (TR1.2, TR1.3), scaling algorithms (TR20.1), and microservices (TR13.1), adhering to workload requirements while optimizing resource usage and costs. **Data archival (TR4.2, TR11.2)** is streamlined through integrations with S3 Glacier, while **security (TR3.6, TR3.1-3.4, TR8.1)** is enforced via AWS WAF, encryption, and IAM policies to protect user data and meet regional compliance standards. For **reliability (TR2.1, TR11.1)**, CloudFormation provides multi-AZ architectures with Auto Scaling and data replication for high availability and fault tolerance. Its tight integration with AWS-native services makes CloudFormation an optimal choice, simplifying infrastructure management while meeting the TRs for a scalable, secure, and cost-effective hotel booking system.

# 10 Conclusion - Optional

# 10.1 The lessons learned

Designing cloud solutions is challenging but rewarding. Writing business and technical requirements allowed us to deeply understand the problem and refine the scope of the solution. Iterating on designs helped create a more compact and fitting approach. During this entire procedure, we remembered one thing that the professor told us in our first project meeting -**”Always remember which hat you are wearing.”** By this he implied, whenever taking a decision while designing or documenting we should think from the perspective of a Cloud Architect. This advice helped us in every section of the project. Exploring trade-offs through frameworks like the Well-Architected Framework and methods like even-swaps helped in decision-making. Despite limitations in testing environments, we learned the value of experimenting with finalized requirements before finalizing designs. Overall, this experience was insightful and iterative improvements remain key.

# 11 References

[1] Instructor Notes- <https://moodle-courses2425.wolfware.ncsu.edu/pluginfile.php/505696/course/section/85216/CloudArchitectureNotes2023.pdf?time=1687793847340>

[2] <https://aws.amazon.com/architecture/well-architected/>

[3] <https://aws.amazon.com/blogs/apn/the-6-pillars-of-the-aws-well-architected-framework/>

[4] <https://aws.amazon.com/cloudwatch/>

[5] <https://aws.amazon.com/eks/>

[6] <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>

[7] <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

[8] <https://aws.amazon.com/global-accelerator/>

[9] <https://aws.amazon.com/route53/>

[10] <https://aws.amazon.com/rds/aurora/>

[11] <https://aws.amazon.com/s3/>

[12] <https://aws.amazon.com/free/security/>

[13] <https://aws.amazon.com/ec2/>

[14] <https://aws.amazon.com/step-functions/>

[15] [https://aws.amazon.com/config/](https://aws.amazon.com/step-functions/)

[16] <https://www.nasdaq.com/articles/the-big-3-in-cloud-computing>

[17] <https://docs.aws.amazon.com/wellarchitected/latest/framework/rel-dp.html>

[18] <https://docs.aws.amazon.com/wellarchitected/latest/framework/sec-design.html>

[19] <https://docs.aws.amazon.com/wellarchitected/latest/framework/cost-dp.html>

[20] <https://docs.aws.amazon.com/wellarchitected/latest/framework/perf-dp.html>

[21] <https://medium.com/dazn-tech/aws-application-load-balancer-algorithms-765be2eca158>

[22] <https://hbr.org/1998/03/even-swaps-a-rational-method-for-making-trade-offs>

[23] <https://iamondemand.com/blog/elb-vs-alb-vs-nlb-choosing-the-best-aws-load-balancer-for-your-needs/>

[24] <https://docs.marklogic.com/guide/ec2/CloudFormation>

[25] <https://aws.amazon.com/elasticloadbalancing/network-load-balancer/>

[26] <https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/>

[27] <https://en.wikipedia.org/wiki/EBPF>

[28] <https://docs.aws.amazon.com/whitepapers/latest/aws-overview/six-advantages-of-cloud-computing.html>

[29] <https://www.romexsoft.com/blog/aws-vs-azure-vs-gcp-comparison/>

[30] [Handout H4.1 - AWS-wellarchitected-framework.pdf](https://moodle-courses2425.wolfware.ncsu.edu/pluginfile.php/506115/mod_folder/content/0/Handout%20H4.1%20-%20AWS-wellarchitected-framework.pdf?forcedownload=1) Page 4

[31] <https://aws.amazon.com/cloudformation/features/>

[32] <https://www.terraform.io/>

## 

Our architectural decision-making will follow a two-step approach:

Feature Comparison Evaluate potential solutions by analyzing their features, considering AWS capabilities and our application's specific requirements.

Even Swaps Method systematically resolves technical trade-offs by comparing and balancing competing architectural attributes.

The final recommendation will synthesize insights from both analyses, ensuring a comprehensive, data-driven architectural selection.

\begin{enumerate}

\item \textbf{Choosing the right load balancer: }

The strategic selection of a Load Balancer is crucial for resolving key technical conflicts in our hotel booking platform, particularly addressing challenges related to latency (TR 5.1), scalability (TR 1.2), monitoring overhead (TR 1.4), cost efficiency (TR 11.1), and data isolation/security (TR 17.1). By implementing an advanced Load Balancer with intelligent routing capabilities, multi-tenant support, and low-latency performance, we can effectively mitigate the tensions between data isolation, system responsiveness, and scaling requirements that would otherwise compromise the platform's architectural integrity.\\\textbf{Feature based comparison:}\\ \\

[(n) indicates rank for that alternative(top row) against the particular objective mentioned in the left column]

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\setlength{\tabcolsep}{12pt}

\begin{tabular}{p{4cm}p{3cm}p{3cm}p{3cm}}

\toprule

\textbf{Aspect} & \textbf{AWS ELB} & \textbf{AWS ALB} & \textbf{AWS NLB} \\

\midrule

\textbf{Protocol Support} & TCP/HTTP/HTTPS (1) & HTTP/HTTPS (1) & TCP/UDP (2) \\

\textbf{Performance} & Moderate, suitable for basic apps (2) & Optimized for web apps, low latency (1) & High performance, low latency (1) \\

\textbf{Cost} & Fixed pricing based on usage (2) & Based on requests and data processed (1) & Based on connections and traffic (1) \\

\textbf{Routing Features} & Basic round-robin routing (2) & Host/Path-based routing, content-based (1) & IP-based, supports complex setups (1) \\

\textbf{Scalability} & Suitable for legacy apps (3) & Best for microservices \& web apps (1) & Best for high-volume or low-latency needs (2) \\

\textbf{Sticky Sessions} & Yes, but limited (2) & Yes (1) & No (3) \\

\bottomrule

\end{tabular}

\caption{Comparison of AWS Load Balancing Options}

\label{tab:aws-load-balancers}

\end{table}

\textbf{Even Swaps Method for Choosing AWS Load Balancer: ALB, NLB, ELB}\\ \\

\textbf{Step 1: Initial Comparison}\\

We begin by analyzing the three load balancers, AWS ALB, NLB, and ELB, based on critical Technical Requirements (TRs): latency (TR 5.1), scalability (TR 1.2), monitoring overhead (TR 1.4), cost efficiency (TR 11.1), and data isolation/security (TR 17.1).

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{2.0}

\begin{tabular}{p{2cm}p{2.5cm}p{2.5cm}p{2.5cm}p{2.5cm}p{2.5cm}}

\toprule

\textbf{Load Balancer} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Data Isolation (TR 17.1)} \\

\midrule

\textbf{ALB} & Medium & High & Medium & Medium & High \\

\textbf{NLB} & Low & High & Low & High & Medium \\

\textbf{ELB} & Medium & Medium & Low & Medium & Medium \\

\bottomrule

\end{tabular}

\caption{Comparison of AWS Load Balancers Based on Key Technical Requirements}

\label{tab:aws-load-balancers-tr}

\end{table} \\ \\

Refer to table 5.2.

At this stage, no alternative is conclusively better than the others, so we proceed to eliminate dominated options step by step.\\ \\

\textbf{Step 2: NLB vs. ALB (Latency and Scalability)} \\ \\

\textbf{Latency (TR 5.1):}\\

NLB operates at Layer 4 (transport layer), making it optimized for low latency as it directly forwards packets without processing HTTP/HTTPS requests. In contrast, ALB functions at Layer 7 (application layer), introducing additional overhead for features like path-based routing and Web Application Firewall (WAF). NLB wins on latency. \\ \\

\textbf{Scalability (TR 1.2):} \\

Both NLB and ALB scale automatically with traffic patterns, ensuring high throughput and resilience. Since they perform equally well in this regard, scalability remains a tie.

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{2.0}

\setlength{\tabcolsep}{10pt}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Load Balancer} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Data Isolation (TR 17.1)} \\

\midrule

\textbf{ALB} & Medium & High & Medium & Medium & High \\

\textbf{NLB} & Low & High & Low & High & Medium \\

\textbf{ELB} & Medium & Medium & Low & Medium & Medium \\

\bottomrule

\end{tabular}

\caption{Comparison of AWS Load Balancers Based on Key Technical Requirements}

\label{tab:aws-load-balancers-tr}

\end{table}

\textbf{Outcome:} \\ \\

NLB outperforms ALB on latency, but ALB remains in the comparison due to its strong security features.\\ \\

\textbf{Step 3: ALB vs. ELB (Scalability and Monitoring Overhead)}\\ \\

\textbf{Scalability (TR 1.2):} \\ \\

ALB is designed for modern web applications, supporting dynamic host-based and path-based routing. ELB, being a legacy solution, lacks these advanced features and is less suited for high-demand environments. ALB wins. \\ \\

\textbf{Monitoring Overhead (TR 1.4):}

ALB integrates seamlessly with AWS CloudWatch, providing detailed metrics for Layer 7 traffic, while ELB has fewer monitoring capabilities and lacks Layer 7 analysis. Again, ALB wins.

\setlength{\tabcolsep}{8pt} % Adjust space between columns

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Load Balancer} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Data Isolation/Security (TR 17.1)} \\

\midrule

\textbf{ALB} & Medium & High & Medium & Medium & High \\

\textbf{ELB} & Medium & Medium & Low & Medium & Medium \\

\bottomrule

\end{tabular}

\caption{Comparison of AWS Load Balancers Based on Key Technical Requirements}

\label{tab:aws-load-balancers-tr}

\end{table}

\textbf{Outcome:} \\ \\

ELB is eliminated due to weaker scalability and monitoring capabilities. \\ \\

\textbf{Step 4: NLB vs. ELB (Cost Efficiency and Monitoring Overhead)}\\ \\

\textbf{Cost Efficiency (TR 11.1):} \\ \\

NLB is more cost-effective, operating at the transport layer with minimal processing overhead. ELB incurs similar costs but lacks the modern features offered by NLB or ALB, making it less competitive. \\ \\

\textbf{Monitoring Overhead (TR 1.4):} \\ \\

NLB provides basic but sufficient monitoring capabilities, integrating with CloudWatch. ELB’s outdated monitoring capabilities make it less suitable for detailed traffic analysis. NLB wins. \\ \\

\setlength{\tabcolsep}{10pt}

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Load Balancer} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Data Isolation/Security (TR 17.1)} \\

\midrule

\textbf{NLB} & Low & High & Low & High & Medium \\

\textbf{ELB} & Medium & Medium & Low & Medium & Medium \\

\bottomrule

\end{tabular}

\caption{Comparison of AWS Load Balancers Based on Key Technical Requirements}

\label{tab:aws-load-balancers-tr}

\end{table}

\textbf{Outcome:} \\ \\

ELB is conclusively eliminated. The final comparison is between NLB and ALB. \\ \\

\textbf{Step 5: NLB vs. ALB (Security vs. Latency/Cost Efficiency)} \\ \\

\textbf{Data Isolation and Security (TR 17.1):} \\ \\

ALB provides advanced security features, including WAF integration, SSL offloading, and tenant data isolation. These features make ALB the preferred option for secure multi-tenant architectures. NLB, while secure, focuses on packet forwarding and lacks these advanced capabilities. ALB wins.

\textbf{Latency and Cost Efficiency (TR 5.1, TR 11.1):} \\ \\

NLB excels in both latency and cost efficiency due to its Layer 4 design, which bypasses application-layer processing. It is particularly suited for real-time, high-throughput scenarios where cost and performance are critical. NLB wins. \\ \\

\setlength{\tabcolsep}{10pt} % Adjust space between columns

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Load Balancer} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Data Isolation/Security (TR 17.1)} \\

\midrule

\textbf{ALB} & Medium & High & Medium & Medium & High \\

\textbf{NLB} & Low & High & Low & High & Medium \\

\bottomrule

\end{tabular}

\caption{Comparison of AWS Load Balancers Based on Key Technical Requirements}

\label{tab:aws-load-balancers-tr}

\end{table}

\textbf{Learning}

\begin{enumerate}

\item \textbf{Choose NLB} if latency and cost efficiency are the top priorities (e.g., real-time booking systems with high traffic).

\item \textbf{Choose ALB} if security and data isolation are more critical (e.g., multi-tenant environments with strict compliance).

\item \textbf{ELB is eliminated} as it fails to meet modern scalability, monitoring, and cost-efficiency requirements.

\end{enumerate}

\textbf{Final Selection:} \\

From our comparison we see that AWS ALB is the best suited solution for our application. The nature of hotel booking platforms involves intricate web-based interactions, requiring sophisticated routing capabilities like host and path-based routing to manage different service endpoints such as search, reservation, payment, and customer support. The robust support for HTTP/HTTPS protocols and advanced routing features enable seamless handling of microservices architecture, allowing the booking platform to scale elastically. Low-latency performance ensures quick response times during peak booking periods, such as holiday seasons or flash sales, while its request-based pricing model provides cost-efficiency. Additionally, strong sticky session support is required to ensure consistent user experiences by maintaining client connection states during critical processes like multi-step reservation flows, preventing potential booking interruptions. \\ \\

\item \textbf{Choosing a Load Balancer Algorithm}

\setlength{\tabcolsep}{10pt} % Adjust space between columns

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{4cm}p{5cm}p{5cm}}

\toprule

\textbf{Objective} & \textbf{Least Outstanding Requests} & \textbf{Round Robin} \\

\midrule

\textbf{Load Distribution} & Routes to servers with the fewest active requests, balancing real-time load dynamically. (1) & Distributes requests equally without assessing current server conditions. (2) \\

\textbf{Performance} & Improves efficiency by minimizing response time under varying workloads. (1) & May lead to delays if some servers face higher processing times or traffic spikes. (2) \\

\textbf{Complexity} & Requires tracking active requests per server, making it computationally more complex (2) & Straightforward implementation; minimal computation overhead. (1) \\

\textbf{Suitability for Workloads} & Best for unpredictable workloads, such as surges during flash sales or peak travel seasons. (1) & Works for evenly distributed traffic but struggles with sudden demand variations. (2) \\

\textbf{Adaptability} & Dynamically reallocates traffic to less-burdened servers for faster responses. (1) & Static allocation fails to adapt to variations in server states or processing times. (2) \\

\bottomrule

\end{tabular}

\caption{Comparison of Least Outstanding Requests and Round Robin Based on Key Factors}

\label{tab:algorithm-comparison}

\end{table}

\setlength{\tabcolsep}{10pt} % Adjust space between columns

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Algorithm} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Elasticity (TR 1.3)} \\

\midrule

\textbf{LOR} & Low & High & High & Medium & Medium \\

\textbf{Round Robin} & Medium & Medium & Low & High & High \\

\bottomrule

\end{tabular}

\caption{Comparison of LOR and Round Robin Algorithms Based on Key Technical Requirements}

\label{tab:algorithm-comparison}

\end{table}

\setlength{\tabcolsep}{10pt} % Adjust space between columns

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Algorithm} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Elasticity (TR 1.3)} \\

\midrule

\textbf{LOR} & \textbf{Low} & \textbf{High} & High & Medium & Medium \\

\textbf{Round Robin} & Medium & Medium & Low & High & High \\

\bottomrule

\end{tabular}

\caption{Comparison of LOR and Round Robin Algorithms Based on Key Technical Requirements}

\label{tab:algorithm-comparison}

\end{table}

\setlength{\tabcolsep}{10pt} % Adjust space between columns

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Algorithm} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Elasticity (TR 1.3)} \\

\midrule

\textbf{LOR} & \textbf{Low} & \textbf{High} & High & Medium & Medium \\

\textbf{Round Robin} & Medium & Medium & \textbf{Low} & \textbf{High} & High \\

\bottomrule

\end{tabular}

\caption{Comparison of Least Outstanding Requests (LOR) and Round Robin Algorithms Based on Key Technical Requirements}

\label{tab:algorithm-comparison}

\end{table}

\setlength{\tabcolsep}{10pt} % Adjust space between columns

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\begin{tabular}{p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}p{2cm}}

\toprule

\textbf{Algorithm} & \textbf{Latency (TR 5.1)} & \textbf{Scalability (TR 1.2)} & \textbf{Monitoring Overhead (TR 1.4)} & \textbf{Cost Efficiency (TR 11.1)} & \textbf{Elasticity (TR 1.3)} \\

\midrule

\textbf{LOR} & \textbf{Low} & \textbf{High} & High & Medium & Medium \\

\textbf{Round Robin} & Medium & Medium & \textbf{Low} & \textbf{High} &\textbf{ High} \\

\bottomrule

\end{tabular}

\caption{Comparison of LOR and Round Robin Algorithms Based on Key Technical Requirements}

\label{tab:algorithm-comparison}

\end{table}

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\setlength{\tabcolsep}{12pt}

\begin{tabular}{p{4cm} p{5cm} p{5cm}}

\toprule

\textbf{Objective} & \textbf{LOR} & \textbf{Round Robin} \\

\midrule

\textbf{Latency (TR 5.1)} & Low; minimizes latency by prioritizing servers with fewer outstanding requests. (1) & Medium; may result in delays during high traffic due to lack of load consideration. (2) \\

\textbf{Scalability (TR 1.2)} & High; dynamically adapts to server load, ensuring scalability under unpredictable conditions. (1) & Medium; does not account for dynamic server load changes, which may affect scalability. (2) \\

\textbf{Monitoring Overhead (TR 1.4)} & High; requires monitoring server states to track outstanding requests. (2) & Low; no monitoring required, reducing computational overhead. (1) \\

\textbf{Cost Efficiency (TR 11.1)} & Medium; additional complexity in tracking requests can increase operational costs. (2) & High; simple implementation reduces operational costs. (1) \\

\textbf{Elasticity (TR 1.3)} & Medium; although adaptable, it may incur slight delays during high-traffic events. (2) & High; simple design allows rapid scaling without delays. (1) \\

\bottomrule

\end{tabular}

\caption{Comparison of LOR vs Round Robin Based on Key Objectives}

\label{tab:lor-vs-rr}

\end{table}

\textbf{Final Selection:}

From our comparison we see that Least Outstanding Requests (LOR) is the best suited solution for our application. Using the Least Outstanding Requests (LOR) load balancing algorithm is crucial for a hotel booking system due to its ability to handle dynamic and unpredictable workloads effectively. Unlike Round Robin, which evenly distributes traffic without considering server conditions, LOR routes requests to servers with the fewest active requests, ensuring real-time load balancing. This dynamic approach minimizes response times and enhances performance, particularly during flash sales or peak travel seasons when traffic surges are common. While LOR requires tracking active requests, adding computational complexity, it is well worth the trade-off for its adaptability and efficiency under varying workloads. By reallocating traffic to less-burdened servers, LOR ensures consistent user experience, which is vital for maintaining customer satisfaction in high-demand scenarios.

-----------------------------------

\item \textbf{Which DB exactly supports our needs?}

The strategic selection of a database is essential to meet the demands of the hotel booking application, considering factors such as faster access, performance, reliability, and cost efficiency. For performance, a database must support low-latency queries (TR 5.1) and handle high transactional workloads, particularly for real-time booking and payment processing. Reliability (TR 2.1, TR 11.1) requires features like data replication and fault tolerance to ensure availability during peak loads.

A database should support scalable architectures (TR 1.2, TR 13.1), allowing dynamic resource allocation based on demand, and provide efficient indexing for faster access. Cost optimization (TR 4.2, TR 11.2) is critical, particularly for archiving inactive tenant data and minimizing storage costs. Security (TR 3.1-3.4, TR 8.1) is also a priority, requiring encryption and compliance with regional laws to protect tenant information. Choosing a suitable database is fundamental to balancing these requirements while maintaining application efficiency and reliability.

\begin{table}[h!]

\centering

\renewcommand{\arraystretch}{1.5}

\setlength{\tabcolsep}{12pt}

\begin{tabular}{p{4cm} p{5cm} p{5cm}}

\toprule

\textbf{Objective} & \textbf{AWS RDS} & \textbf{Aurora DB} \\

\midrule

\textbf{Performance} & Suitable for general-purpose applications with moderate read/write workloads. (2) & Optimized for high-performance, low-latency workloads, handling millions of transactions per second. (1) \\

\textbf{Cost} & Lower cost; billed per instance and storage usage. (1) & Higher cost but more cost-efficient for high-performance workloads. (2) \\

\textbf{Startup Time} & Faster deployment for simpler setups. (1) & Slightly more time-consuming due to its advanced configuration options. (2) \\

\textbf{Engine Support} & Supports multiple engines (MySQL, PostgreSQL, Oracle, SQL Server). (1) & Supports MySQL and PostgreSQL-compatible engines with Aurora-specific optimizations. (2) \\

\textbf{High Availability} & Multi-AZ deployments for failover support. (Tie) & Designed for high availability with up to 15 read replicas and automatic failover. (Tie) \\

\bottomrule

\end{tabular}

\caption{Comparison of AWS RDS vs Aurora DB Based on Key Objectives}

\label{tab:rds-vs-aurora}

\end{table}

In a hotel booking system, fine-grained control and cost considerations take precedence because workloads can often be estimated based on previous usage patterns, such as seasonal peaks or booking trends. With predictable traffic, RDS offers sufficient performance without the added expense of Aurora DB, which is optimized for unpredictable and high-volume workloads. Fine-grained control allows for tailored configurations to meet specific operational needs in a hotel booking system. For example, during predictable seasonal peaks like holidays, database administrators can adjust RDS configurations to allocate more resources for query caching and optimize connection pooling. Similarly, if the booking system requires compliance with specific data retention laws in different regions, administrators can customize backup schedules and retention policies accordingly.