
yourticket

CONTENTS

YourTicket Description

YourTicket is a concert ticketing platform designed to handle high-demand ticket sales. It faces the unique challenge of managing thousands of concurrent users with demand spikes reaching up to 10,000 requests per second during ticket release moments. The system is required to facilitate simultaneous ticket purchases, ensure each seat is sold only once, and provide a real-time overview of available seats.

Core Functions of YourTicket Site

Book Tickets:

1. The system shall allow users to book tickets for events.
2. The system shall ensure that each seat is sold only once and prevent double booking.

View Events:

1. The system shall enable users to view details of events, including date, time, venue, and ticket prices.
2. The system shall display real-time information about the availability of seats for each event.

Search Events:

1. The system shall provide a search feature for users to find events.

Architecturally Interesting aspects of YourTicket

1. The system must be elastically scalable to handle bursts of traffic, accommodating up to 10,000 requests per second when tickets go on sale.
2. The system must be highly available, especially during peak ticket sales periods, to support thousands of concurrent users.
3. The system must ensure strong consistency in ticket sales to prevent selling any seat more than once
4. The system should provide real-time updates of available seats to the users.

Proposed Solution

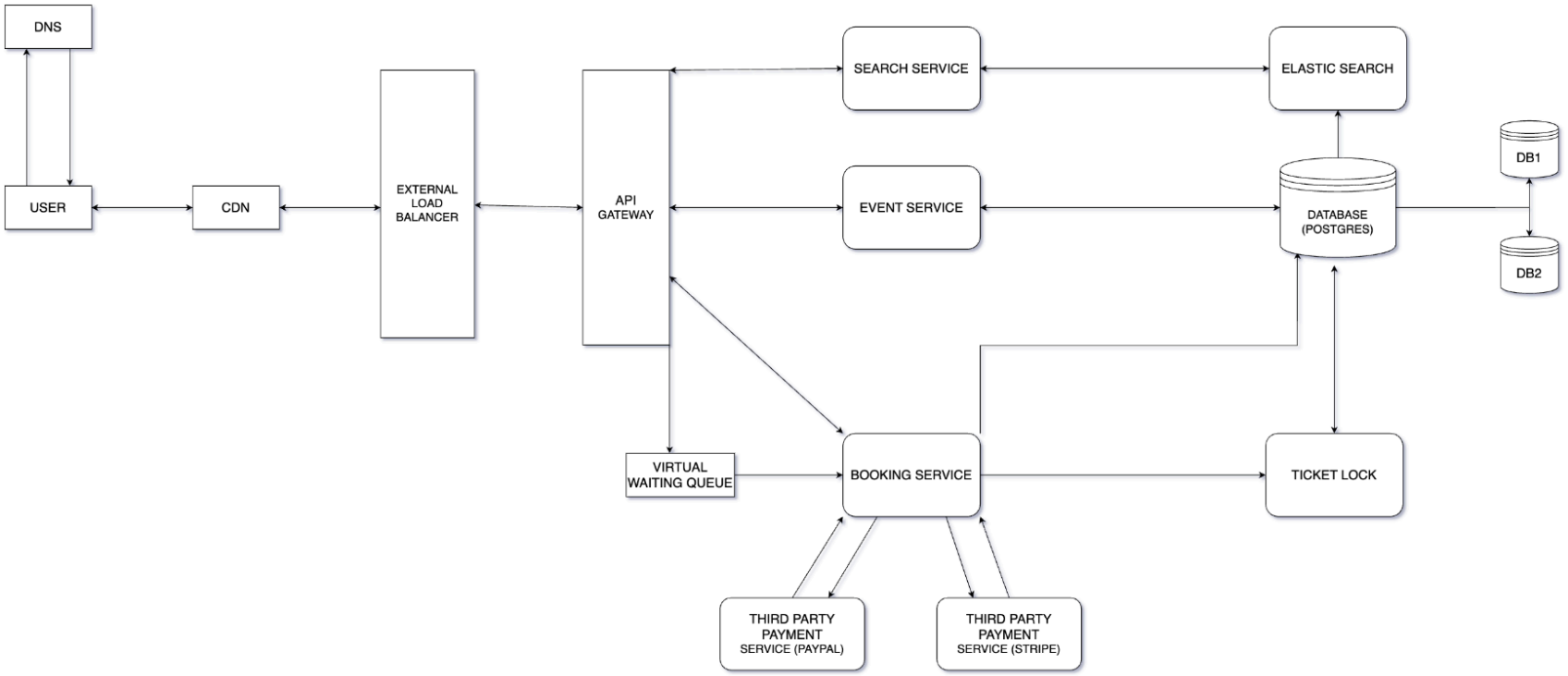


Figure 1 : Architectural solution of the Concert Ticketing system

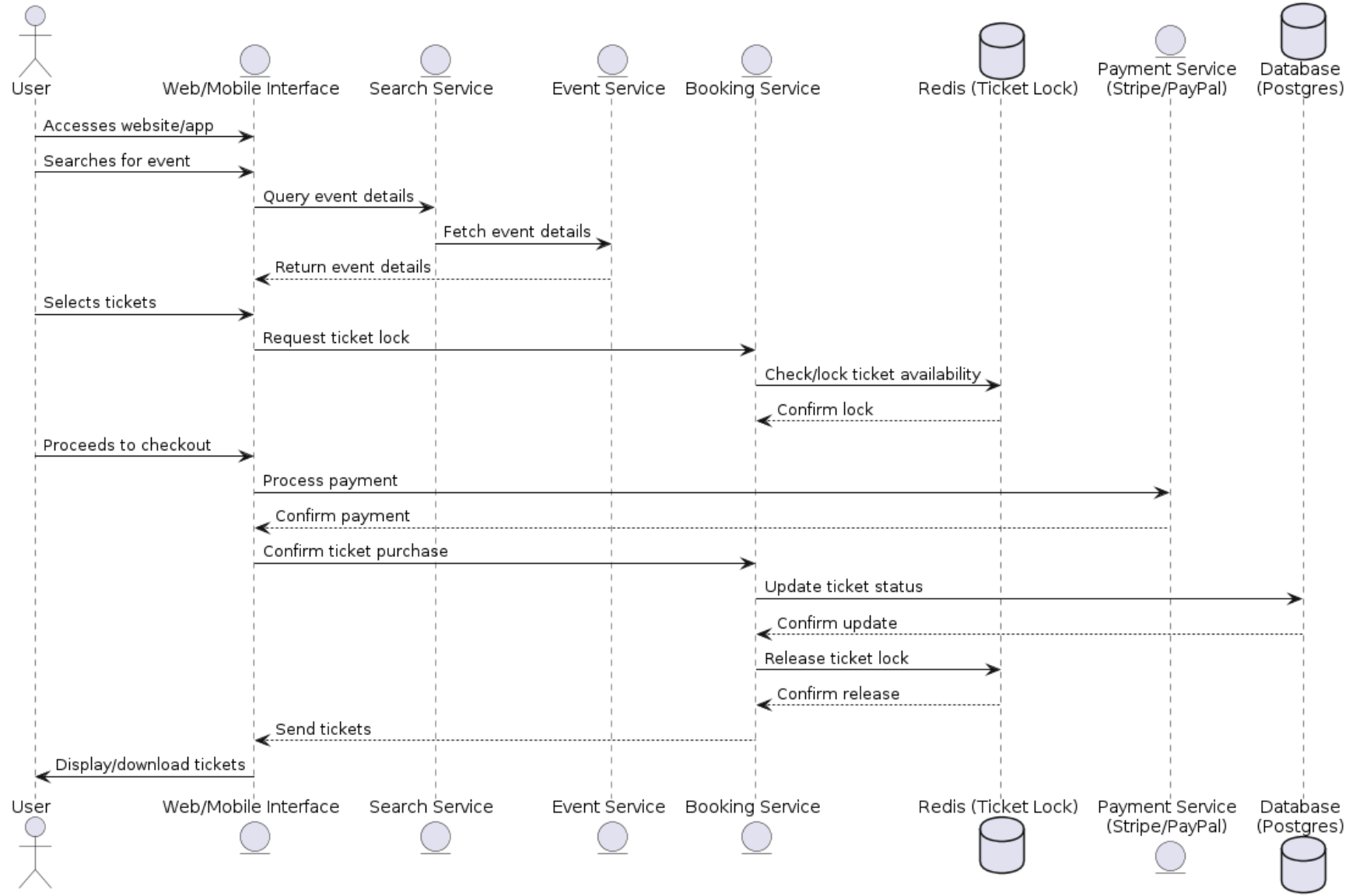


Figure 2: Ticket Purchase Interaction Diagram

Description of proposed solution

The system uses a Microservice Architecture and the explanation for various components of the system are as follows.

1. **User Interaction:**
   The journey begins when a user interacts with the system, possibly through a web or mobile interface.

2. **DNS Lookup**:
   The user's request first goes through a DNS resolution process, where the domain name is resolved to an IP address.

3. **Content Delivery Network (CDN)**:
   Static content is served to users via a CDN, which is designed to reduce latency by caching content in multiple geographical locations. This ensures faster load times for static assets.

4. **External Load Balancer**:
   The incoming requests for dynamic content are managed by an External Load Balancer, which ensures that the traffic is distributed evenly across the server infrastructure to maintain performance and prevent any one server from being overloaded.

5. **API Gateway**:
   Once the request passes through the load balancer, it reaches the API Gateway. This component acts as a single entry point for all backend services, providing a layer of abstraction between the client-side and the server-side software. Here the API Gateway handles authentication, rate limiting, and request routing.

6. **Microservices**:
   The architecture leverages a microservices approach, with individual services like Search Service, Event Service, and Booking Service each running in their isolated environments. This separation allows for individual scaling and deployment, which increases the system's resilience and flexibility.

7. **Virtual Waiting Room**:
   To manage high loads during peak times, such as when tickets for a popular event are released, the system can queue requests in a virtual waiting room. This

throttling mechanism prevents the system from being overwhelmed and provides users with a fair and orderly way to access limited resources.

8. **Third-Party Payment Services**:
   For payment processing, the system integrates with external payment services such as Stripe and PayPal. These integrations help to offload the complexity and security concerns associated with financial transactions.

9. **Elasticsearch**:
   The system leverages Elasticsearch for its robust searching capabilities, enabling users to swiftly locate events based on diverse criteria. Additionally, it employs node query caching to efficiently store and retrieve the top 10,000 searches.

10. **Database (Postgres)**:
    The main database used is PostgreSQL, which stores persistent data related to users, events, tickets, etc. It provides strong consistency which is crucial for transactions and Ideal for representing the interrelated data in a ticketing system.

11. **Ticket Lock (Redis)**:
    When booking tickets, the system uses Redis as a fast, in-memory data store to implement ticket locks. This ensures that once a user selects a ticket, it is temporarily reserved, preventing other users from purchasing it simultaneously.

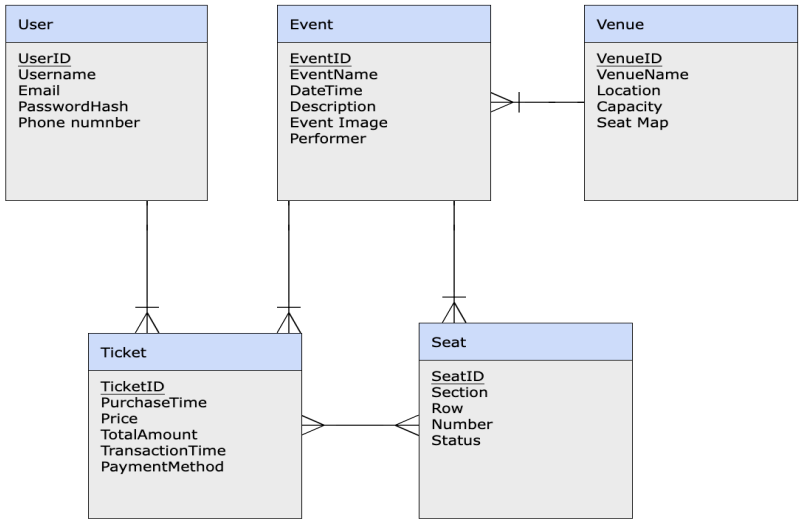
12. **Server-Sent Events (SSE) for real time seat map:**
    It utilizes Server-Sent Events (SSE) for pushing real-time updates from the server to the client. This is particularly crucial for popular events where seat availability can change within seconds.

13. **Change data capture for Elasticsearch Synchronization:**
    CDC is utilized primarily to update Elasticsearch indices with the latest information. Considering that Event and Venue data does not change frequently, CDC offers a streamlined approach to keep Elasticsearch in sync without the need for continuous polling or complex data synchronization routines.

14. **Data Sharding**:
    To enhance performance and scalability, the database is likely sharded, with references to multiple databases like DB, DB2 etc. Sharding the database allows "YourTicket" to distribute the load across multiple databases, which can help manage large datasets and high throughput.

    PostgreSQL seemed to be a good choice for this kind of relational data and ensured strong consistency.

High Level Datamodel of the Concert Ticketing System



Other Solution Considered and Discarded

Space Based Architecture style was considered and discarded because of the following reasons.

Data Consistency Challenges: Space Based Architecture relies on eventual consistency, ensuring data consistency across distributed nodes can be more challenging, particularly under high load or in the presence of network partitions.

Technology Ecosystem and Support: Space Based Architecture is less common than microservices, which means there are fewer resources, less community support, and a smaller talent pool familiar with this architecture. This can make it harder to implement and maintain.

Cost Considerations: The economies of scale provided by microservices running on containerized infrastructure might offer more cost-effective scaling.

Flexibility and Adaptability: Space Based Architecture may limit the flexibility to use different technologies and platforms for different parts of the application, as the entire space usually relies on a uniform technology stack.

Architectural Choice Made and Why was it made

Microservices architecture was used for the Concert Ticketing Site. It is a style of software design where applications are structured as a collection of loosely coupled services.

Reasons for the choosing Microservices architecture

Aspect	Description
Scalability	The platform demands the capability to handle sudden and significant surges in traffic, especially during popular event ticket releases. Microservices allow for scaling individual components independently to meet varying load requirements.
Resilience	By decomposing the application into microservices, the system becomes more resilient. If one service fails, it doesn't bring down the entire application, which is crucial for a high-availability ticketing platform.
Technological Flexibility	This architecture allows for using the most suitable technology stack for each service based on its specific requirements, like using Redis for rapid, temporary data storage in the Ticket Lock system.
Easier Maintenance and Debugging	Smaller, well-defined services are easier to maintain and debug compared to a large monolithic application.

Deployment Environment

AWS has been chosen to deploy YourTicket. AWS is the perfect choice for YourTicket for the following reasons.

1. AWS provides a vast array of services that cater to different aspects of computing, such as computing power, storage, databases, machine learning, and Internet of Things (IoT), to name a few.

2. AWS has a wide global network of data centers spread across multiple geographical regions and availability zones and also comparatively more than others.

3. AWS operates on a pay-as-you-go pricing model, which means you pay only for the services you use without upfront expenses or long-term contracts and also competitive pricing when compared with others.

AWS Services used in YourTicket

Service	Description
Route 53	AWS's DNS service translates domain names like www.yourticket.com into IP addresses, routing user requests and improving availability with health checks and routing policies.
AWS WAF (Web Application Firewall)	Guards against web exploits and attacks by defining customizable security rules for the application.
CloudFront	AWS's CDN service caches static content near users, reducing latency and load times for assets such as images and stylesheets.
Elastic Load Balancing (ELB)	Distributes incoming traffic across multiple targets, enhancing fault tolerance and scalability.

Amazon API Gateway	Front door for "YourTicket" APIs, managing request routing, SSL termination, authentication, and authorization.
Amazon EKS (Elastic Kubernetes Service)	Orchestrates Kubernetes containerized applications, simplifying deployment, scaling, and operations.
AWS Fargate	Serverless compute engine for containers, enabling automatic scaling without managing servers or clusters.
Elasticsearch	Provides advanced search capabilities for "YourTicket," making event discovery efficient for users.
Amazon Aurora Serverless	On-demand, auto-scaling configuration for Amazon Aurora, offering cost-effective database solutions.
Redis	Fast, in-memory data store for ticket locking in "YourTicket," ensuring each seat is sold only once.

Table: Various components of the proposed solution and corresponding AWS Service.

Why EKS + FARGATE chosen ?

Combining Amazon EKS (Elastic Kubernetes Service) with AWS Fargate offers a potent mix of Kubernetes' powerful orchestration capabilities with the operational simplicity of serverless computing. This integration allows developers to deploy containerized applications without managing underlying servers, focusing instead on application development while AWS handles scalability, security, and infrastructure management. This setup enhances cost efficiency by eliminating over-provisioning and paying only for the compute resources used, offering robust application isolation and automatic scaling to efficiently handle varying workloads.

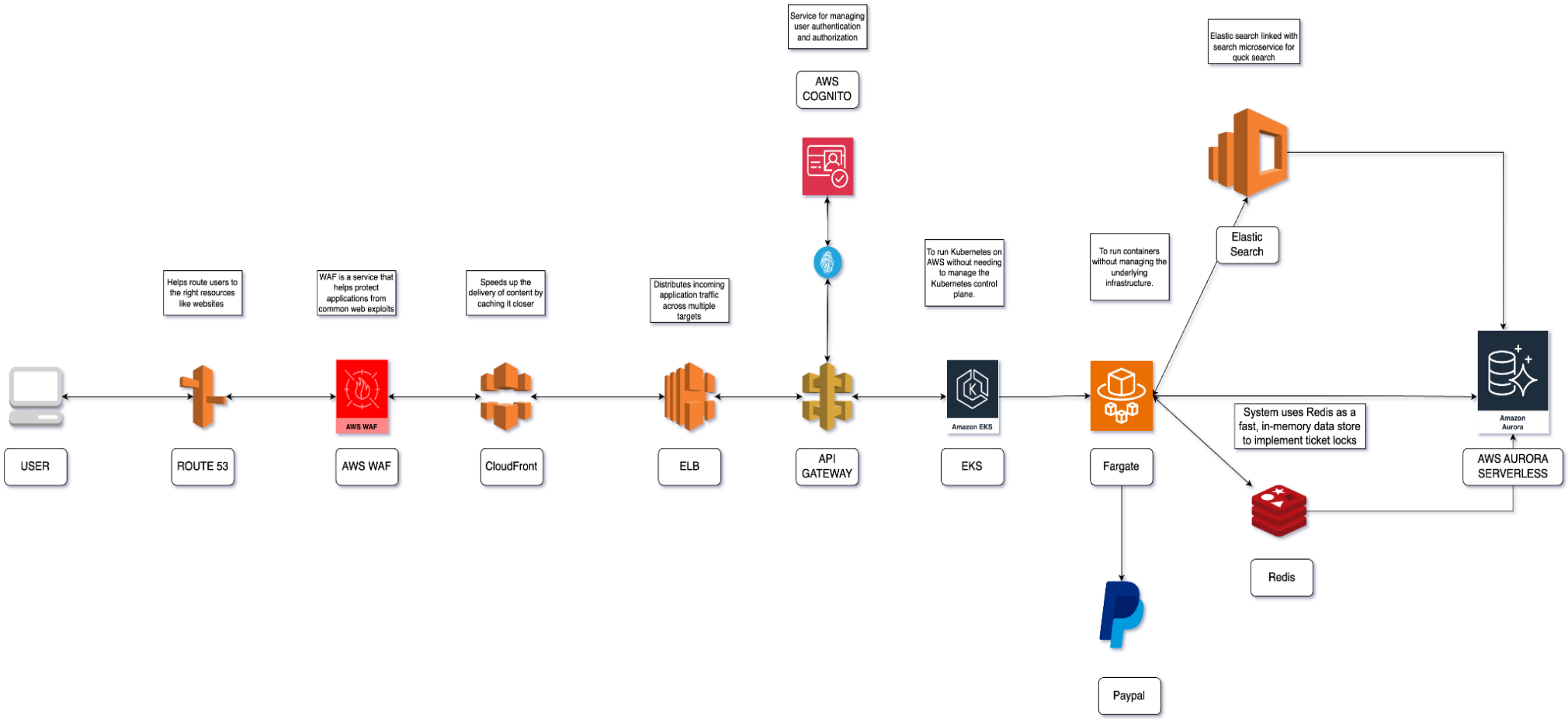


Figure: Describing the flow and various components of the system when deployed in AWS environment

Fitness Functions

1. Scalability Testing:

Function: Automatically test the system's ability to handle a specific number of concurrent users or requests, particularly during peak load times like major event ticket releases.

Validation: The system successfully handles up to 10,000 requests per second without significant degradation in response times.

2. Data Consistency and Integrity:

Function: Validate data consistency and integrity across different system components, particularly in Booking services.

Validation: Data is consistently and accurately reflected across all system components after updates, with no anomalies or integrity issues.

Failure Cases and Mitigations

In the "YourTicket" system, several failure scenarios are possible given its complexity and high-demand nature. Identifying these likely failure cases and implementing mitigation strategies is crucial for maintaining system integrity and performance. Here are some common failure scenarios and their respective mitigations:

1. **High Traffic Overload**

   Failure Case: The system could become overwhelmed during peak ticket release times, leading to slow response times or system crashes.

   Mitigation: Implement auto-scaling for critical services and load balancers in AWS. Utilize caching mechanisms (like Redis) to reduce database load and use a Content Delivery Network (CDN) to offload static content delivery.

2. **Third-Party Service Failures (e.g., Payment Gateway)**

   Failure Case: Dependency on external services like payment gateways could lead to failures in processing transactions.

Mitigation: Integrate multiple payment gateways to provide alternatives in case one fails. Implement graceful degradation where the system continues to operate in a limited mode when a non-critical service fails.

Normal, Warning and Alarm state

State	Indicators	Monitoring Actions
Normal	- Response times within 200 ms - CPU usage under 70% - Memory usage under 70% - Database queries executing under 100 ms - Error rates below 0.1% - Successful user transactions over 99.5% - Performance metrics within expected range	- Regular health checks every 5 minutes - Continuous performance monitoring - Log analysis for unusual activities
Warning	- Response times between 200 ms and 500 ms - CPU or memory usage between 70% and 90% - Database queries executing between 100 ms and 500 ms - Error rates between 0.1% and 0.5% - Minor service slowdowns - Delays in third-party services - Approaching alert thresholds	- Increase monitoring frequency to every minute - Prepare scaling procedures - Analyze logs for error spikes - Alert technical team to check for potential issues - Initiate performance tuning - Review third-party service SLAs - Early warning alerts to stakeholders
Alarm	- Response times over 500 ms - CPU or memory usage over 90% - Critical service downtimes	- Immediate response from technical team - Initiate scaling/failover procedures - Engage recovery protocols

	- Security breaches or critical failures - Critical alerts - Scaling/failover initiated	- Notify security team and initiate incident response - Emergency alerts to all stakeholders - Monitor systems for stabilization post-failover

Data Growth and Impacts

For the "YourTicket" concert ticketing platform, expected data growth and the factors influencing it depend on various aspects of the system's usage, market trends, and operational scope. Here's an analysis:

Expected DATA Growth

1. **Increasing User Base**: As the popularity of "YourTicket" grows, more users will register and use the platform, leading to a steady increase in user-related data.
2. **Number of Events**: As "YourTicket" expands its offerings to include more events, the data associated with each event (like event details, artist information, venue data) will grow proportionally.
3. **Ticket Sales and Bookings**: Each ticket sale or booking generates transactional data. High-demand events can cause significant spikes in data generation.
4. **User Interactions**: User searches, browsing history, and interaction data can grow rapidly, especially as the platform scales and adds more features.

Impacts of DATA Growth

1. **Increased Storage Requirements**: As the user base expands and more events are added, the amount of data stored will increase significantly. This requires scalable storage solutions to handle both the steady growth and sudden spikes associated with high-demand events.
2. **Performance and Scalability Challenges**: With the increase in user interactions, ticket sales, and event data, the platform will need to maintain high performance levels. This may involve scaling up database and application server capacities, as well as optimizing database queries and indexing to handle larger datasets efficiently.
3. **Cost Implications**: Data growth directly impacts costs related to data storage, processing, and management

Business Drivers

1. Market Demand for Reliable and Fast Ticketing

Impact on Services: To meet this demand, the architecture must support high concurrency and quick response times during peak sales periods. This leads to the implementation of scalable microservices, load balancers, and a robust CDN to manage traffic effectively and distribute it across multiple servers.

2. Customer Satisfaction and Retention

Impact on Services: The need for a user-friendly interface that provides quick access to event information and seamless ticket purchasing requires a front-end that is responsive and backed by efficient backend services. This involves utilizing a user-centric design and ensuring that backend services like the search and booking microservices are optimized for performance.

Business Continuity Needs

The two most important business continuity needs can be categorized and addressed as follows:

1. High Availability: Ensuring that the platform remains operational and accessible at all times, especially during high-traffic events like ticket releases. (Up to 99.99% availability)

2. Disaster Recovery: Ability to quickly recover from catastrophic events like data center failures, natural disasters, or major cyber attacks.