HealthBooker

A High-Availability Patient Booking and Health Management System on AWS

Team Members

BSSE23012 ---- Jamshaid Ahmed

BSSE23021 ---- Muhammad Sher

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

Traditional healthcare clinics face significant operational inefficiencies due to manual appointment scheduling and fragmented patient health records. This leads to high administrative overhead, increased patient no-shows, poor data accessibility for providers, and a subpar patient experience in a digital-first world. HealthBooker is a secure, scalable, and highly available cloud-native web application designed to solve these challenges. It provides a dual-portal system: a Patient Portal for easy online booking and health record access, and a Provider Portal for managing appointments and patient clinical data.

The solution is architected on AWS for maximum reliability and security. It leverages an Auto Scaling Group of EC2 instances behind an Application Load Balancer for the backend API, Amazon DynamoDB for a serverless, low-latency database, and Amazon Cognito for robust user identity management and role-based access control. The project successfully delivers a production-grade application that reduces administrative tasks, enhances patient engagement, centralizes health data for better clinical decision-making, and provides a resilient platform built on cloud best practices.

PROJECT OVERVIEW

INTRODUCTION

The healthcare industry is undergoing a significant digital transformation, moving away from legacy, on-premises systems towards agile, cloud-based solutions. This shift is driven by the need for greater operational efficiency, enhanced data security, and the rising expectation of patients for modern, digital experiences. Cloud platforms like Amazon Web Services (AWS) provide the foundational building blocks—such as on-demand compute, managed databases, and robust security services—necessary to build compliant, scalable, and resilient healthcare applications. This project, HealthBooker, leverages these AWS capabilities to create a modern solution for patient

booking and health management, demonstrating the practical application of cloud architecture principles to solve a real-world problem.

PROBLEM STATEMENT

The core problem addressed by this project is the operational inefficiency and data fragmentation inherent in traditional clinical management systems. Small to medium-sized clinics are often hindered by:

1. **High Administrative Workload:** Staff spend a significant portion of their day on the phone scheduling appointments, sending manual reminders, and managing paper or siloed digital records.
2. **Fragmented Patient Data:** A patient's medical history is often scattered across various files and systems, making it difficult for providers to get a single, holistic view, which can impact the quality of care.
3. **Poor Patient Experience:** The lack of self-service options, such as online booking and access to personal health records, does not meet the expectations of modern, tech-savvy patients.
4. **Scalability and Reliability Issues:** Legacy systems are expensive to maintain, difficult to scale during peak times, and often lack robust disaster recovery capabilities, posing a risk to business continuity.

AIMS & OBJECTIVES

The primary aim of this project was to design, develop, and deploy a comprehensive, multi-user Patient Booking and Health Management system on AWS, adhering to best practices for security, scalability, and high availability.

To achieve this aim, the following objectives were established:

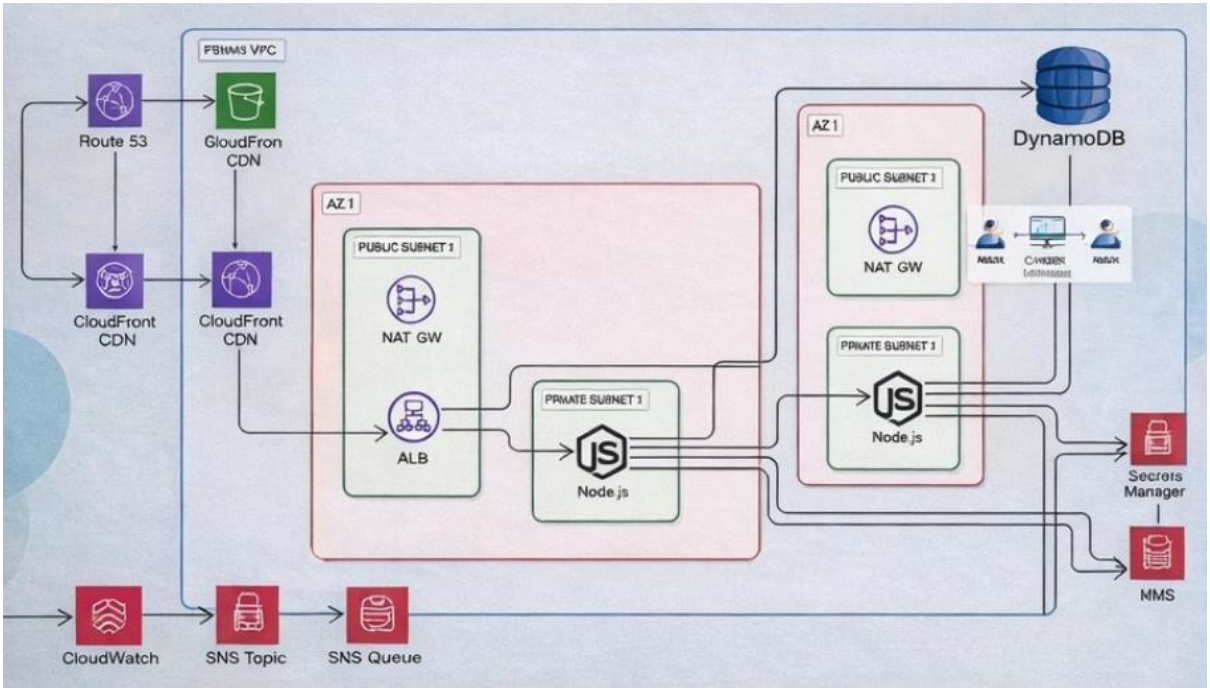
1. To implement secure, role-based authentication for Patients and Providers using Amazon Cognito.
2. To deploy a fault-tolerant Node.js backend using EC2 instances in an Auto Scaling Group across multiple Availability Zones, managed by an Application Load Balancer.
3. To design and implement a scalable single-table data model in Amazon DynamoDB to store all application data, including users, appointments, and health records.
4. To create a secure and isolated network environment using a custom VPC with public and private subnets.

5. To deliver a fully functional React-based web application that provides distinct, intuitive portals for both patients and providers.

SYSTEM ARCHITECTURE DESIGN

ARCHITECTURAL DESIGN

The HealthBooker platform is built on a multi-tier architecture designed for high availability and security, hosted entirely within a custom Amazon Virtual Private Cloud (VPC).



[Figure 1: High-Availability AWS Architecture]

ARCHITECTURAL FLOW

The end-to-end flow of a user request is as follows:

1. A user accesses the React frontend application. API requests are directed to an Application Load Balancer (ALB), which serves as the single, secure entry point to the backend.
2. The ALB routes traffic to a target group containing a fleet of Amazon EC2 instances distributed across multiple Availability Zones (e.g., us-east-1a and us-east-1b).
3. These EC2 instances reside in private subnets, making them inaccessible directly from the public internet.
4. Each EC2 instance runs the Node.js/Express.js backend application, which interacts with the Amazon DynamoDB database to perform CRUD operations.
5. User authentication is managed by Amazon Cognito, and the backend API validates a JSON Web Token (JWT) on every protected request to identify the user and their role.
6. A response is sent back through the same path to the user's browser.

TECHNICAL DEPTH

NETWORK & SECURITY FOUNDATION

Virtual Private Cloud (VPC): A custom VPC was created to establish a logically isolated network. A multi-AZ architecture with public and private subnets was implemented to provide the foundation for high availability and security. The backend EC2 instances are placed in private subnets, shielding them from direct external threats, a core tenet of cloud security.

Security Groups: Acting as stateful virtual firewalls, security groups were configured to enforce the principle of least privilege. The ALB's security group allows inbound HTTP traffic, while the EC2 instances' security group only allows inbound traffic on the application port (5001) from the ALB's security group. This meticulous rule ensures that the application servers can only be reached via the load balancer.

Amazon Cognito: Cognito was chosen as the fully managed Identity Provider to handle all aspects of user management. We implemented Cognito User Groups (Patients and Providers) to enforce Role-Based Access Control (RBAC). The backend middleware inspects the JWT issued by Cognito to authorize or deny access to specific API endpoints, ensuring a patient cannot access provider-only functions.

IAM Role for EC2: To avoid hardcoding credentials, an already provided IAM Role (LabRole) was attached to the EC2 instances. This role granted the application code the specific permissions needed to communicate with DynamoDB securely and automatically.

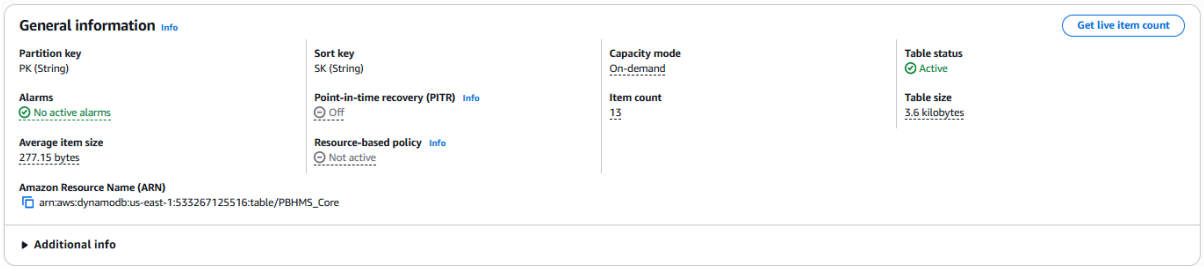
HIGH AVAILABILITY COMPUTE LAYER

Application Load Balancer (ALB): The ALB is the cornerstone of the application's reliability. It distributes incoming requests evenly across all healthy EC2 instances and performs constant health checks. If an instance becomes unresponsive, the ALB immediately stops routing traffic to it, ensuring seamless service continuity.

Auto Scaling Group (ASG) & Launch Template: This pairing provides automation, resilience, and scalability. The ASG's primary role is to maintain the desired capacity of two instances. If an instance fails, the ASG automatically provisions a new one, making the application self-healing. The Launch Template serves as a master blueprint, containing a user data script that fully automates the setup of a new instance, guaranteeing consistency and enabling rapid recovery.

SERVERLESS DATA LAYER

Amazon DynamoDB (Single-Design): DynamoDB was selected for its consistent low-latency performance and serverless nature. A single-table design was implemented to optimize performance by reducing the number of database requests needed to render complex views.



[Figure 2: DynamoDB Configuration]

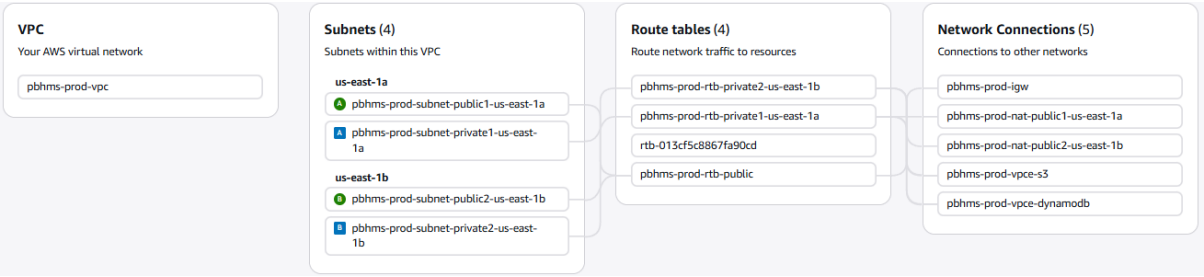
Key Strategy: The Partition Key (PK) is the user's ID, which co-locates all data for a user. The Sort Key (SK) defines the data type (e.g., METADATA, APPOINTMENT#...). This allows us to fetch a patient's profile and all their appointments in a single, efficient Query operation.

Global Secondary Index (GSI): To allow providers to view their schedules (an access pattern not possible on the main table), a GSI named **ProviderScheduleIndex** was created with providerId as its partition key. This provides a secondary, highly efficient query path tailored specifically to this business requirement.

IMPLEMENTATION SUMMARY & VERIFICATION

VPC & NETWORKING SETUP

The foundational network was established using the AWS VPC Wizard to create a multi-AZ VPC, including public/private subnets, an Internet Gateway, and NAT Gateways. Security Groups were then manually configured as described in section 3.1.



[Figure 3: VPC Resource Map]

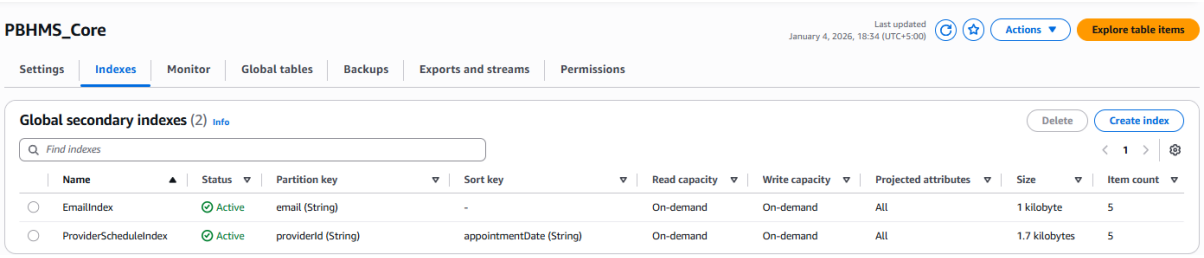
IDENTITY & ACCESS MANAGEMENT

A Cognito User Pool was created with two User Groups: Patients and Providers. The IAM Role for the EC2 instances was configured with **LabRole** policy granting access to DynamoDB.

[Figure 4: Cognito User Groups]

DATABASE CONFIGURATION

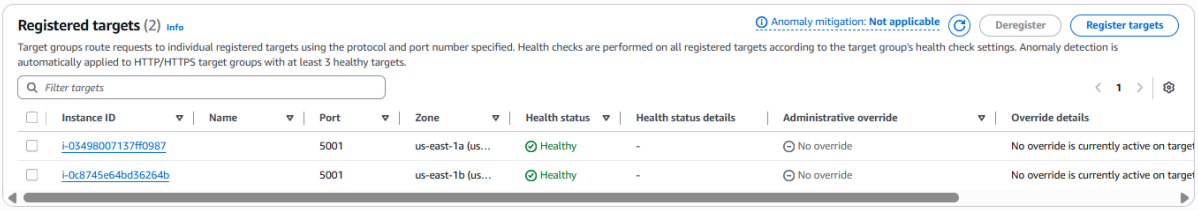
The PBHMS_Core table was created in DynamoDB with PK and SK as its composite primary key. The ProviderScheduleIndex GSI was subsequently added to the table to support the provider schedule query pattern.



[Figure 5: DynamoDB Table PBHMS_Core's Global Secondary Indices]

COMPUTE LAYER DEPLOYMENT

A Launch Template was created, including the final user data script for automated application deployment. An Auto Scaling Group was then configured to use this template, launching a minimum of two **t3.micro** instances across the private subnets. Finally, an Application Load Balancer was created and configured to route traffic to a target group associated with the ASG.



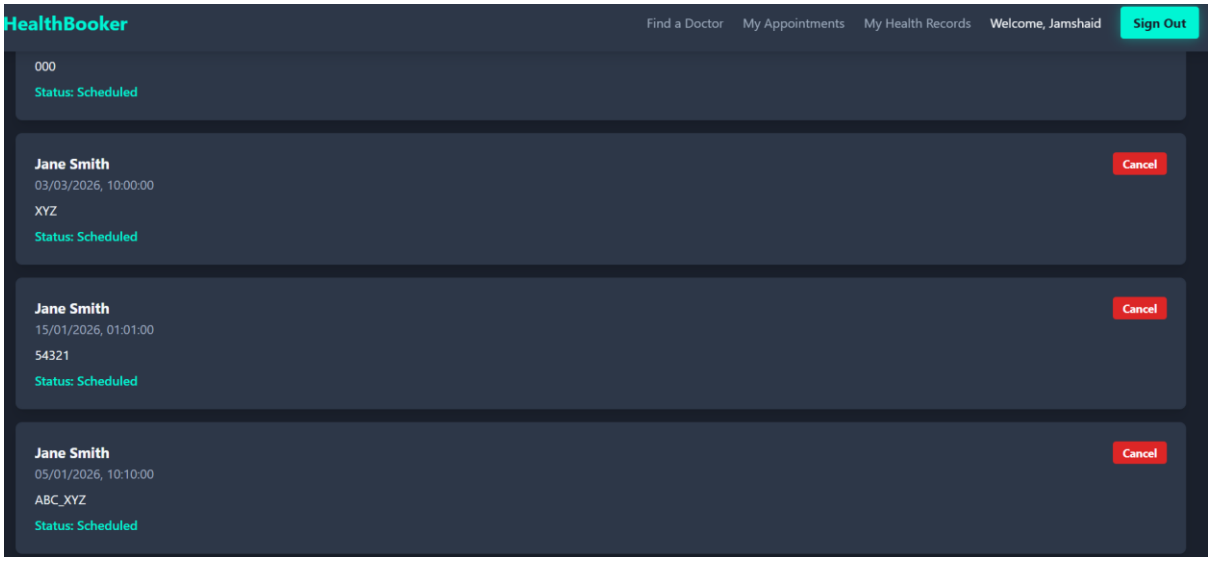
[Figure 6: Healthy EC2 Instance in the ASG associated TG]

TESTING & VERIFICATION WORKFLOW

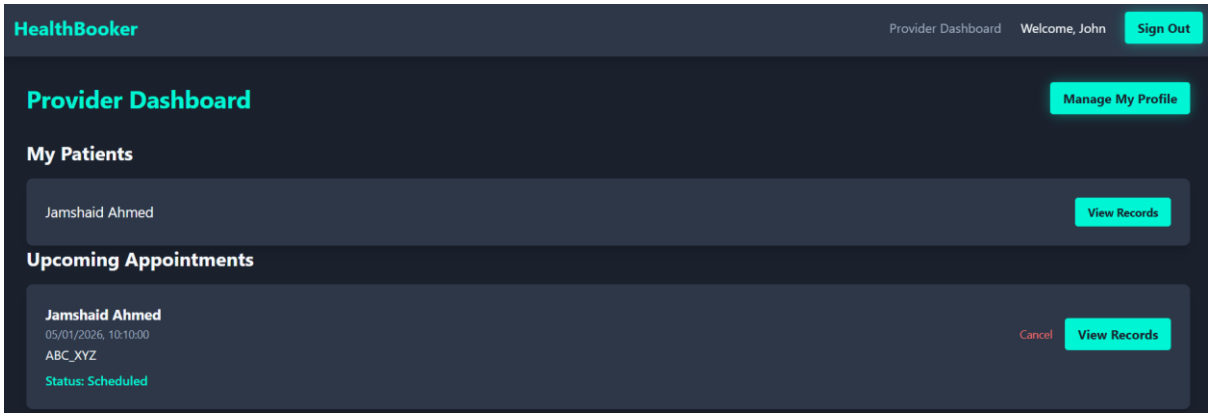
End-to-end testing was performed to validate the entire system. A typical test involved:

1. Signing up and confirming a new Patient user.
2. Logging in as the Patient and booking an appointment with a Provider.
3. Verifying the new appointment item was created correctly in the DynamoDB table.

4. Logging in as the Provider and seeing the new appointment on their dashboard.
5. The Provider then added a health record for the patient, which was verified in DynamoDB.
6. Finally, logging back in as the Patient to confirm they could view the newly added health record. This successful workflow validated the functionality of the entire stack, from the frontend UI to the backend logic and database interactions.



[Figure 7: Patient Booked Appointments]



[Figure 8: Doctor Upcoming Appointments]

CONCLUSION

The HealthBooker project successfully demonstrates the design and implementation of a secure, scalable, and highly available web application using core AWS services. By combining a multi-AZ VPC architecture with a self-healing compute layer (ALB/ASG) and a high-performance serverless database (DynamoDB), the application meets the stringent requirements of a modern healthcare platform. The use of managed services like Cognito significantly reduced development complexity while enhancing the security and functionality of the system. This project serves as a practical blueprint for building robust, real-world applications on the AWS cloud.

REFERENCES & PROMPT LINKS

AWS WELL ARCHITECTED FRAMEWORK

AISTUDIO PROMPT LINK

DYNAMODB PARTITIONING GUIDE