AWS Cloud Project Report

Serverless Event Registration System with AI-Powered Connection Matching

A report by

Byna Sriroop (23BCE1863)

November 7, 2025

Contents

1 Abstract

This project details the design and implementation of a highly scalable, serverless event registration platform built on Amazon Web Services. The primary objective is to solve the traditional challenges of monolithic event systems—namely, high costs, poor scalability, and limited attendee value. This system provides a cost-efficient, resilient, and high-performance application that moves beyond simple registration to provide significant added value through artificial intelligence.

The core architecture is 100% serverless, leveraging Amazon S3 for static website storage and Amazon CloudFront as the global Content Delivery Network (CDN) to ensure low-latency access for all users. AWS Lambda serves as the core compute engine, executing all backend business logic without managing servers. Amazon API Gateway provides a secure, managed "front door" for all API endpoints, which in turn trigger the Lambda functions. Amazon DynamoDB is used as the auto-scaling NoSQL database, handling all application data from registrations to user profiles.

This project successfully implements three advanced AI features to maximize attendee engagement. The flagship "DNA Matching System" analyzes comprehensive user profiles—including skills, goals, and interests—to intelligently connect like-minded attendees and facilitate professional networking. This is complemented by an AI Event Recommender that uses a hybrid collaborative filtering model to suggest relevant sessions and events. Finally, an NLP-based AI Chatbot provides 24/7, on-demand support for user queries. The result is a production-ready, intelligent, and auto-scaling platform that dramatically enhances the event experience.

2 Introduction

2.1 Project Background

Traditional event management has long been supported by monolithic, server-based applications. This legacy architecture presents significant technical, financial, and user-experience challenges. Event organizers are often forced to over-provision expensive servers to handle anticipated peak loads (like a ticket launch), yet they must continue to pay for this idle compute capacity during off-peak times. Conversely, if they under-provision, they risk catastrophic system failure during traffic spikes, leading to lost revenue and reputational damage. For a global audience, these centralized servers also result in high latency, creating a slow and frustrating user experience for attendees far from the server's location.

2.2 Problem Statement

Beyond the technical limitations of legacy systems, existing platforms often fail to deliver on a primary goal of event attendance: valuable professional networking. Most platforms function as simple ticketing or scheduling tools. Attendees are left to their own devices, struggling to find and connect with the right people (e.g., mentors, partners, or peers) in a crowded environment. This missed opportunity represents a significant failure in delivering tangible value to the end-user.

This project directly addresses this two-fold problem:

1. **The Architectural Problem:** How to build an event platform that is cost-efficient, automatically scales to meet any demand, and provides a low-latency global experience.

2. **The Value Problem:** How to transform the platform from a simple registration tool into an intelligent system that actively facilitates and enhances attendee networking.

2.3 Scope and Objectives

The scope of this project is the complete, end-to-end design, implementation, and deployment of a serverless, intelligent event management platform on AWS. This encompasses the user-facing registration portal, the AI-driven networking features, and the backend administrative dashboard.

To solve the problems stated above, the following primary objectives were established:

- **Achieve Auto-Scaling & Cost-Efficiency:** Design a 100% serverless backend architecture using AWS Lambda for compute, Amazon DynamoDB for data, and Amazon API Gateway for request handling. This "pay-as-you-go" model eliminates all server management and idle costs, while scaling infinitely to meet demand.

- **Ensure Global High-Performance:** Host the static frontend application on Amazon S3 and distribute it globally using Amazon CloudFront. This ensures that all users, regardless of their location, receive a fast, low-latency experience by being served from a nearby edge location.

- **Implement Robust Security:** Utilize Amazon Cognito (as per the project plan) to manage the entire user authentication and authorization lifecycle, securing all user data and API endpoints.

- **Deliver Intelligent Networking (The "DNA" System):** Develop and integrate a sophisticated "DNA Matching System" that analyzes detailed user profiles (skills, goals, interests) to calculate compatibility scores and suggest high-value professional connections.

- **Enhance User Engagement:** Build and integrate two supplementary AI features:
  - An AI Event Recommender using a hybrid collaborative filtering model to suggest other relevant events.
  - An NLP-based AI Chatbot to provide 24/7, on-demand answers to user queries about events.

2.4 Expected Outcomes

The final outcome of this project is a production-ready, high-performance, and intelligent event platform. The system successfully provides a seamless registration flow, a real-time admin dashboard, and—most critically—a suite of AI-driven features that demonstrably improve event discovery and create tangible networking opportunities for attendees. This architecture provides a modern solution to the outdated challenges of event management.

3 Understanding of AWS Concepts

This project is built on a serverless-first philosophy, which means traditional server management is eliminated. Instead of using EC2 (virtual servers) or RDS (managed relational databases), this architecture leverages their serverless, auto-scaling alternatives: AWS Lambda for compute and Amazon DynamoDB for the database. This design ensures automatic scaling, high availability, and a "pay-as-you-go" cost model.

The following core AWS concepts and services were used:

3.1 Amazon S3 (Simple Storage Service)

Core Concept: A highly durable and scalable object storage service. It's designed to store and retrieve any amount of data (files, images, etc.) from anywhere.

Application in Project: S3 serves as the origin storage for the entire frontend application. All the static assets (HTML, CSS, JavaScript, and image files) for the React application are stored in the `event-registration-app-sriroop` bucket. This bucket is configured for static website hosting, allowing it to serve the application's files.

3.2 Amazon CloudFront

Core Concept: A global Content Delivery Network (CDN) service. It securely delivers data, videos, applications, and APIs to customers globally with low latency and high transfer speeds. It works by caching content in "Edge Locations" close to the end-users.

Application in Project: CloudFront is the public-facing entry point for the frontend. It is configured with the S3 bucket as its "origin." When a user visits the site, they are routed to the nearest CloudFront Edge Location, which serves the cached version of the React application. This provides two key benefits:

- **Performance:** Drastically reduced latency for global users.
- **Security:** Provides HTTPS (SSL/TLS) for the application, ensuring encrypted communication.

3.3 Amazon API Gateway

Core Concept: A fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. It acts as the "front door" for applications to access backend logic.

Application in Project: API Gateway provides the crucial link between the frontend (CloudFront/S3) and the backend (Lambda). The `EventRegistrationAPI` was created to expose the backend logic as a secure RESTful API. It defines specific resource paths (e.g., `/register`, `/chat`, `/find-matches`, `/recommendations`) that, when called by the frontend, trigger the appropriate AWS Lambda function.

3.4 AWS Lambda

Core Concept: A serverless, event-driven compute service (Functions-as-a-Service or FaaS). It allows you to run code for virtually any type of application or backend service without provisioning or managing servers. You only pay for the compute time you consume.

Application in Project: Lambda is the "engine" or "brain" of the entire backend. All business logic is encapsulated in individual Python functions. For example:

- The `EventChatbot` function executes when the `/chat` API is called.
- The `MatchMakerFunction` runs the complex compatibility and scoring algorithm.
- The `GetEventRecommendations` function runs the hybrid collaborative filtering model.
- The `RegisterUserFunction` handles writing new registrations to the database and (as planned) triggering Amazon SES to send an email.

3.5 Amazon DynamoDB

Core Concept: A key-value and document NoSQL database that delivers single-digit millisecond performance at any scale. It is fully managed, serverless, and auto-scaling.

Application in Project: DynamoDB is the primary and only database for this application. It provides fast, low-latency data storage for all application data. The three tables are:

- **Events:** Stores information about all available events.
- **Registrations:** Tracks which user is registered for which event.
- **MatchingProfiles:** Stores the detailed user profile data (skills, goals, interests) required by the `MatchMakerFunction`.

3.6 Amazon SES (Simple Email Service)

Core Concept: A cloud-based email sending and receiving service. It's a highly scalable and cost-effective way for businesses to send transactional emails, marketing messages, or any other type of high-quality content.

Application in Project: SES is the communication engine for the platform. Its planned role (as per Phase 3.3, 3.4, and 4.2) is to:

- **Confirm Registrations:** Be called by the `RegisterUserFunction` (Lambda) to instantly send a "Registration Confirmed" email to the user.
- **Send Reminders:** Be called by the `SendEventReminders` Lambda function (which would be triggered by a CloudWatch Event) to automatically send reminder emails 24 hours before an event.

3.7 AWS IAM (Identity and Access Management)

Core Concept: A web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources.

Application in Project: IAM was crucial for security. Unique IAM Roles were created for each Lambda function. These roles follow the principle of least privilege. For example, the `RegisterUserFunction`'s IAM role would grant it permission to `dynamodb:PutItem` in the Registrations table and `ses:SendEmail` to Amazon SES, but no other permissions.

3.8 Amazon Cognito

Core Concept: A service that provides user sign-up, sign-in, and access control for web and mobile applications. It scales to millions of users and supports sign-in with social identity providers.

Application in Project: As outlined in the original project plan, Cognito is the intended service to manage the entire user authentication lifecycle. It handles user sign-up and sign-in, providing the frontend with a secure JSON Web Token (JWT). This token would then be used to authenticate requests to API Gateway.

4 System Design / Architecture

The system architecture is a 100% serverless, event-driven, and decoupled design. This model ensures high availability, automatic scaling, and minimal operational overhead. The flow of data is governed by distinct, single-purpose services that communicate via APIs.

Architecture Diagram: The complete system architecture diagram can be viewed at:

`https://drive.google.com/file/d/1Wn0og43LAPoyMxiKegZv4kzsQmHWdfJs/view?usp=sharing`

The architecture is composed of the following key components:

4.1 Frontend, Storage, & Networking (S3 + CloudFront)

This layer is the user-facing part of the application and handles all global content delivery.

- **Frontend (User Interface):** This is the React web application that the user interacts with. It includes the event portal, the admin dashboard, and the multi-step "Event Buddy" AI forms.
- **Storage (Amazon S3):** The `event-registration-app-sriroop` bucket serves as the origin store. It holds all the compiled, static frontend assets (HTML, CSS, JavaScript) that make up the React application.
- **Networking & Delivery (Amazon CloudFront):** This is the public-facing entry point for the application, as seen by the cloudfront.net URL. The CloudFront distribution uses the S3 bucket as its origin. It caches the application at global Edge Locations, providing two critical benefits:
  1. **Low Latency:** Users are served from the nearest Edge Location, making the site fast worldwide.
  2. **Security:** It provides default HTTPS (SSL/TLS) for the application, encrypting all user traffic.

4.2 API Layer (Amazon API Gateway)

This service acts as the secure and managed "front door" for all backend logic. It is the single point of entry for the frontend application to communicate with the backend.

- **API Management:** The `EventRegistrationAPI` was created to manage the application's RESTful API.
- **Request Routing:** It is configured with specific resource paths (e.g., `POST /register`, `GET /recommendations`, `POST /chat`, `POST /find-matches`). When the frontend sends an HTTPS request to one of these endpoints, API Gateway is responsible for triggering the correct backend Lambda function and returning its JSON response to the user's browser.

4.3 Backend Compute (AWS Lambda)

This is the serverless "engine" or "brain" of the application. All business logic is encapsulated in single-purpose Python functions that execute on demand. The system does not use any EC2 servers.

- `RegisterUserFunction`: Processes new event signups and writes them to the database.
- `GetEventRecommendations`: Runs the 4-factor hybrid recommendation algorithm (collaborative filtering, category, popularity, urgency) and returns a personalized list of events.
- `MatchMakerFunction`: The most complex function. It executes the "DNA" compatibility scoring (calculating scores for interests, mentorship, goals) and generates `matchReasons` and `icebreaker` strings based on the code.
- `EventChatbot`: Performs intent recognition from user messages and provides context-aware answers (e.g., "What happens at a hackathon?").
- `GetDashboardData`: Aggregates real-time statistics from the database to populate the admin dashboard.

4.4 Database (Amazon DynamoDB)

This is the high-performance, serverless NoSQL database. It provides single-digit millisecond performance at any scale and stores all application data.

- **Table Structure:** The application uses three distinct tables to store data:
  - **Events:** Stores details for all available events (name, date, capacity).
  - **Registrations:** Tracks which user is registered for which event.
  - **MatchingProfiles:** Stores the rich profile data (skills, goals, interests) collected from the "Event Buddy" form. This table is the primary data source for the `MatchMakerFunction`.

4.5 Monitoring & Automation (Amazon CloudWatch)

- **Monitoring (CloudWatch Logs):** All services natively integrate with CloudWatch. Every execution of a Lambda function, every API Gateway request, and every error is logged in real-time. This is essential for debugging the AI algorithms and monitoring application health.
- **Automation (CloudWatch Events):** As per the project plan, a CloudWatch Event rule is designed to be the "cron job" that triggers the `SendEventReminders` Lambda function on a fixed schedule (e.g., once per day).

5 Implementation Details

The project was implemented in a phased, serverless-first approach. The core backend infrastructure (database and API) was established first, followed by the frontend deployment, and finally, the advanced AI features were integrated one by one.

5.1 Step 1: Backend Infrastructure Configuration (The Foundation)

5.1.1 Database Setup (Amazon DynamoDB)

Three NoSQL tables were created to store all application data:

- **Events:** Stores event details.
- **Registrations:** Stores user registration records.
- **MatchingProfiles:** Stores the rich user data (skills, goals, interests) required for the AI matching algorithm. A Global Secondary Index (GSI) was created on `eventId` for this table, allowing the `MatchMakerFunction` to efficiently query "get all profiles for this event".

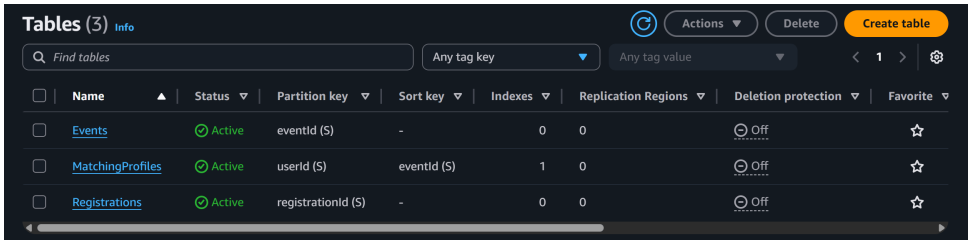

Figure 1: DynamoDB Tables Configuration

5.1.2 Compute Setup (AWS Lambda)

Six separate Python 3.12 Lambda functions were created to encapsulate all business logic:

- Each function was assigned a unique IAM Role with "least-privilege" permissions.
- For example, the `MatchMakerFunction` was given explicit permission to read/write to the `MatchingProfiles` table but had no access to other tables.

- The code for `MatchMakerFunction`, `EventChatbot`, and `GetEventRecommendations` was uploaded.

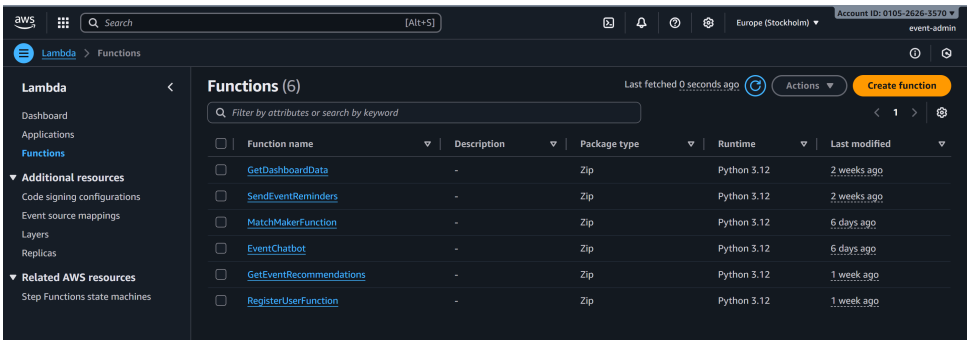

Figure 2: AWS Lambda Functions List

5.1.3 API Setup (Amazon API Gateway)

- An HTTP API named `EventRegistrationAPI` was created.
- Resource paths (e.g., `/register`, `/chat`, `/find-matches`, `/recommendations`) were created to align with the application's needs.
- Integrations were configured to link each path to its corresponding Lambda function.
- CORS (Cross-Origin Resource Sharing) was enabled on all endpoints to allow the frontend application (on a different domain) to make requests to the API.

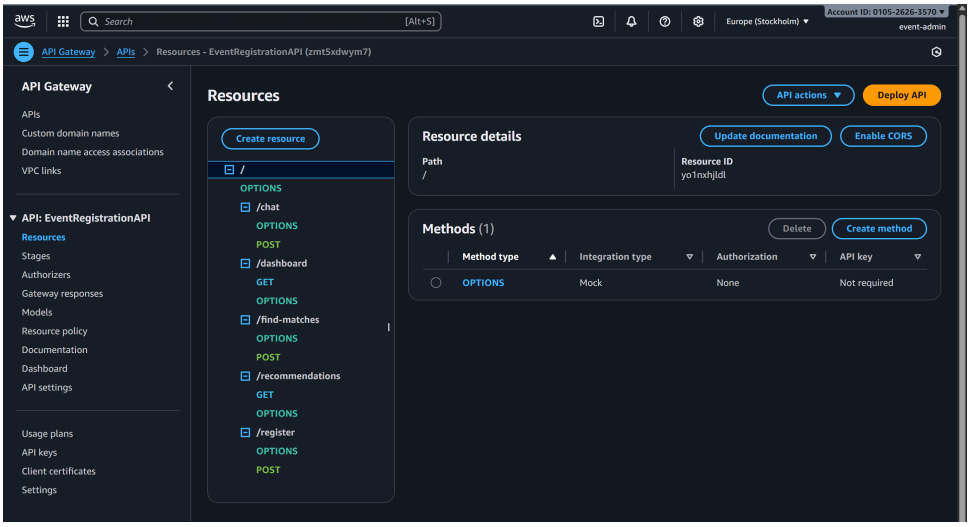

Figure 3: API Gateway Resource Configuration

5.2 Step 2: Frontend Deployment (S3 + CloudFront)

5.2.1 Storage Setup (Amazon S3)

- An S3 bucket named `event-registration-app-sriroop` was created.

- All public access was blocked at the bucket level, a security best practice.
- The compiled React application (build files: index.html, main.js, style.css, etc.) was uploaded to this bucket.

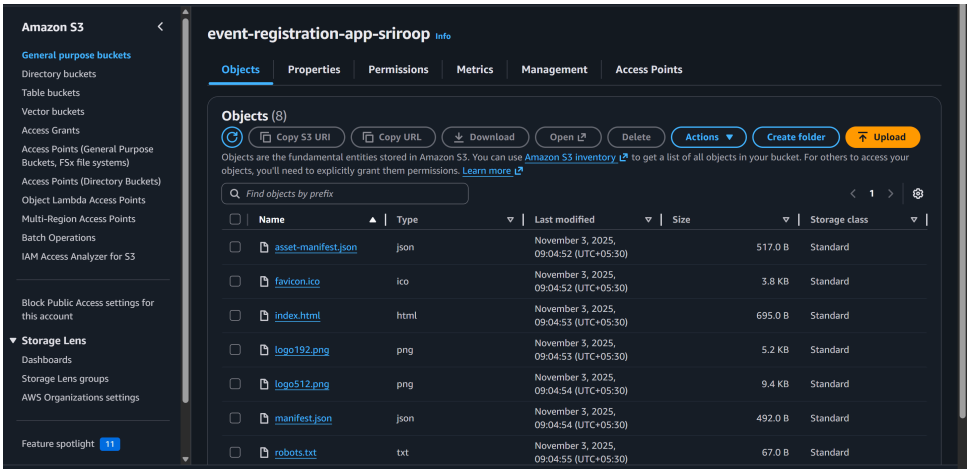

Figure 4: S3 Bucket Configuration

5.2.2 CDN & Deployment (Amazon CloudFront)

- A new CloudFront distribution was created.
- The Origin was set to the S3 bucket (`event-registration-app-sriroop`). An Origin Access Identity (OAI) was used to ensure that S3 only allows access from CloudFront, not a direct public URL.
- The distribution was configured to serve the application over HTTPS, providing security.
- The final cloudfront.net URL became the single, public-facing entry point for the entire application, serving the frontend globally with low latency.

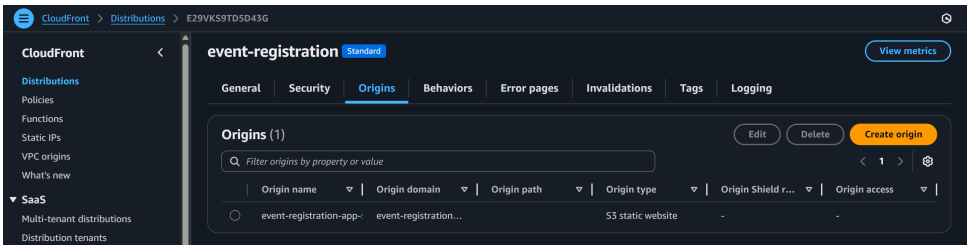

Figure 5: CloudFront Distribution and Origins

5.3 Step 3: AI Feature Implementation & Integration

This phase involved connecting the frontend UI to the backend Lambda functions and implementing the core AI algorithms.

5.3.1 AI Chatbot (EventChatbot Lambda Function)

The `EventChatbot` Lambda function was deployed with advanced NLP capabilities. It uses keyword matching (`detect_intent`) and fuzzy string matching (`extract_event_name`) to understand user queries. Its `get_event_activities` function provides rich, context-aware responses based on event category.

Core Algorithm - Intent Detection:

```
def detect_intent(message):
    """
    Detect user intent from message using keyword matching
    """
    intents = []

    # Greeting patterns
    if any(word in message for word in ['hi', 'hello', 'hey']):
        intents.append('greeting')

    # List events patterns
    if any(phrase in message for phrase in ['list events', 'show events
    ', 'what events']):
        intents.append('list_events')

    # Explain/What happens patterns
    if any(phrase in message for phrase in ['explain', 'what happens',
    'activities']):
        intents.append('explain')
        intents.append('what_happens')

    # Event details patterns
    if any(word in message for word in ['details', 'about', 'info']):
        intents.append('event_details')

    return intents
```

Listing 1: Intent Detection Algorithm

Context-Aware Response Generation:

```
def get_event_activities(event):
    """
    Generate detailed activities based on event type
    """
    event_name = event.get('name', '').lower()
    category = event.get('category', '').lower()

    # Workshop activities
    if 'workshop' in event_name or 'workshop' in category:
        return """Hands-on coding sessions with industry experts
Build real-world projects using latest technologies
Interactive Q&A and live demonstrations
Networking with fellow developers
Certificate of completion"""

    # Hackathon activities
    elif 'hackathon' in event_name:
        return """Form teams and brainstorm innovative ideas
24-hour intensive coding marathon
Mentor support throughout the event
```

```
Pitch your project to judges
Win exciting prizes and recognition"""

    # Default activities
    return """Interactive sessions with industry experts
Networking opportunities with peers
Hands-on activities and demonstrations"""
```

Listing 2: Event Activities Generation

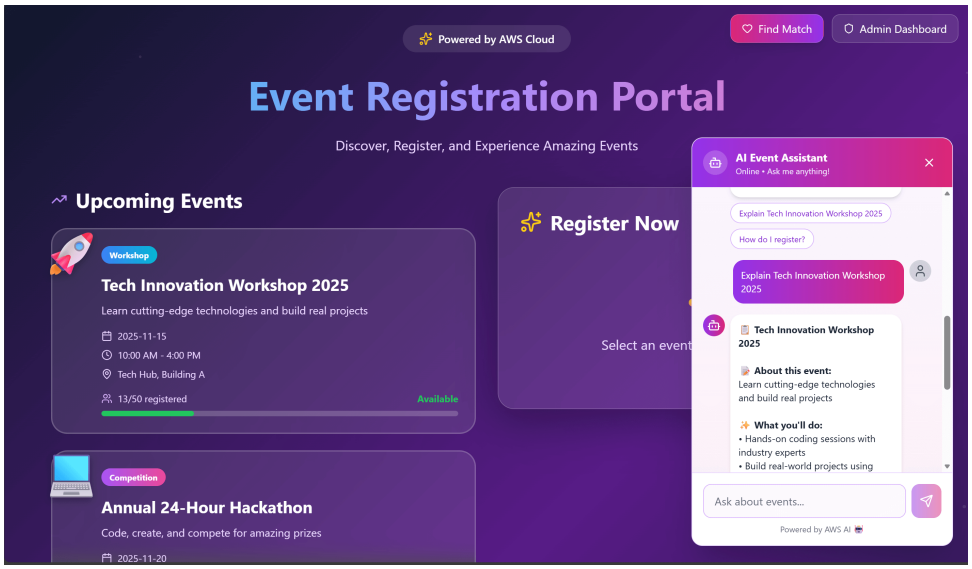

Figure 6: AI Chatbot User Interface

5.3.2 AI Event Recommendations (GetEventRecommendations Lambda Function)

The `GetEventRecommendations` Lambda function implements a sophisticated 4-factor hybrid recommendation algorithm combining Collaborative Filtering, Category Matching, Popularity, and Urgency.

Core Algorithm - Hybrid Recommendation System:

```
def generate_recommendations(user_email, current_event_id, all_events, all_registrations):
    """
    Generate smart event recommendations using multiple algorithms
    """

    # Algorithm 1: Collaborative Filtering
    collab_scores = collaborative_filtering(current_event_id, all_registrations)

    # Algorithm 2: Category-based matching
    category_scores = category_matching(current_event_id, all_events)

    # Algorithm 3: Popularity-based
    popularity_scores = popularity_ranking(all_events)

    # Algorithm 4: Capacity-based urgency
    urgency_scores = urgency_ranking(all_events)
```

```
    # Combine all algorithms with weights
    combined_scores = {}
    for event in all_events:
        event_id = event['eventId']

        # Calculate weighted score
        score = (
            collab_scores.get(event_id, 0) * 0.4 +      # 40% collaborative
            category_scores.get(event_id, 0) * 0.3 +     # 30% category
            popularity_scores.get(event_id, 0) * 0.2 +   # 20% popularity
            urgency_scores.get(event_id, 0) * 0.1        # 10% urgency
        )

        combined_scores[event_id] = score

    # Sort by score and get top 3
    sorted_events = sorted(combined_scores.items(), key=lambda x: x[1], reverse=True)[:3]
    return sorted_events
```

Listing 3: Hybrid Recommendation Algorithm

Collaborative Filtering Implementation:

```
def collaborative_filtering(event_id, registrations):
    """
    Find events that users who registered for this event also registered for
    """
    # Find users who registered for this event
    users_in_event = set()
    for reg in registrations:
        if reg.get('eventId') == event_id and reg.get('status') == 'confirmed':
            users_in_event.add(reg.get('email'))

    # Count what other events these users registered for
    event_counts = Counter()
    for reg in registrations:
        if reg.get('email') in users_in_event:
            other_event = reg.get('eventId')
            if other_event != event_id and reg.get('status') == 'confirmed':
                event_counts[other_event] += 1

    # Normalize scores
    max_count = max(event_counts.values()) if event_counts else 1
    return {eid: count / max_count for eid, count in event_counts.items()}
```

Listing 4: Collaborative Filtering

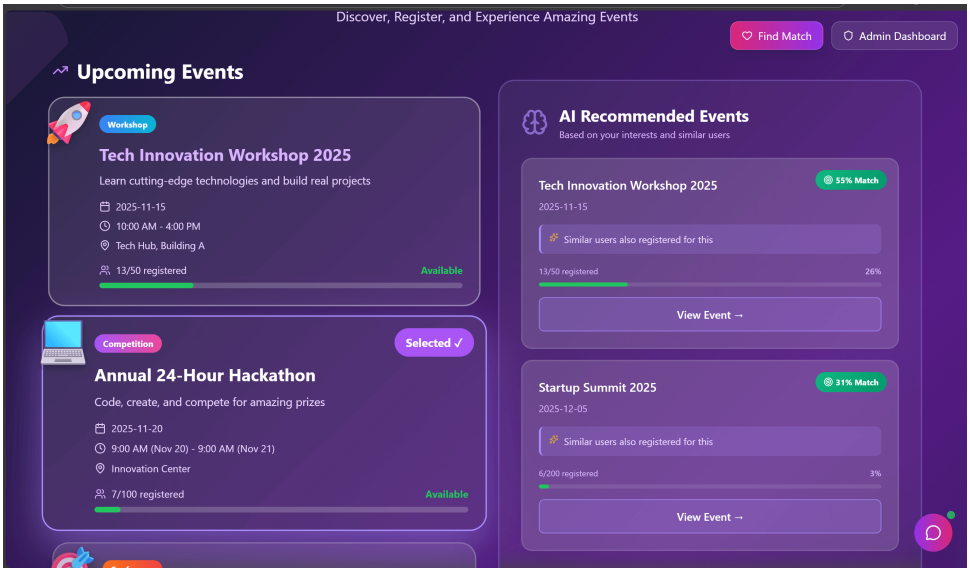

Figure 7: AI Event Recommendations Panel

5.3.3 DNA Matching System (MatchMakerFunction Lambda Function)

The `MatchMakerFunction` is the project's flagship innovation. It implements a sophisticated, multi-vector scoring algorithm that analyzes user profiles to calculate compatibility scores.

Core Algorithm - Compatibility Calculation:

```
def calculate_compatibility(user1, user2):
    """
    Calculate compatibility score between two users
    """
    score = 0
    breakdown = {}
    reasons = []

    # 1. Shared Interests (0-30 points)
    user1_interests = set(user1.get('interests', []))
    user2_interests = set(user2.get('interests', []))
    shared_interests = user1_interests & user2_interests

    interest_score = len(shared_interests) * 10
    if interest_score > 30:
        interest_score = 30

    score += interest_score
    breakdown['interests'] = interest_score

    if shared_interests:
        reasons.append(f"Both interested in: {', '.join(list(
shared_interests)[:2])}")

    # 2. Mentorship Matching (0-25 points)
    user1_looking = set(user1.get('lookingFor', []))

    if 'Mentors' in user1_looking and user2.get('experienceLevel') in [
'Advanced', 'Expert']:
        score += 25
```

```
        breakdown['mentorship'] = 25
        reasons.append(f"{user2.get('name')} can mentor you")

    if 'Mentees' in user1_looking and user2.get('experienceLevel') in ['Beginner', 'Intermediate']:
        score += 20
        breakdown['mentorship'] = 20
        reasons.append(f"You can mentor {user2.get('name')}")

    # 3. Goal Alignment (0-25 points)
    user1_goals = set(user1.get('goals', []))
    user2_goals = set(user2.get('goals', []))
    shared_goals = user1_goals & user2_goals

    goal_score = len(shared_goals) * 8
    if goal_score > 25:
        goal_score = 25

    score += goal_score
    breakdown['goals'] = goal_score

    if shared_goals:
        reasons.append(f"Similar goals: {list(shared_goals)[0]}")

    # 4. Looking For Match (0-15 points)
    if 'Co-founders' in user1_looking and 'Co-founders' in user2.get('lookingFor', []):
        score += 15
        breakdown['lookingFor'] = 15
        reasons.append("Both looking for co-founders!")

    # 5. Same Organization Penalty (-10 points)
    if user1.get('organization', '').lower() == user2.get('organization', '').lower():
        score -= 10
        breakdown['sameOrg'] = -10

    return {
        'total_score': max(0, score),
        'breakdown': breakdown,
        'reasons': reasons
    }
```

Listing 5: DNA Compatibility Scoring Algorithm

Icebreaker Generation:

```
def generate_icebreaker(user1, user2, score_data):
    """
    Generate personalized conversation starter
    """
    reasons = score_data['reasons']

    if not reasons:
        return "Ask about their experience at previous events!"

    # Create contextual icebreakers
    icebreakers = [
        f"Start with: \"{reasons[0]}\"",
```

```
        f"Ask about their work with {user2.get('skills', ['technology'])[0]}",
        f"Discuss shared interest in {user1.get('interests', ['tech'])[0]}"
    ]

    return icebreakers[0]
```

Listing 6: Personalized Icebreaker Generation

Match Finding Algorithm:

```
def find_matches(user_id, event_id, user_profile):
    """
    Find and score all compatible matches for a user
    """
    # Get all profiles for this event using GSI
    response = profiles_table.query(
        IndexName='eventId-index',
        KeyConditionExpression='eventId = :event_id',
        ExpressionAttributeValues={':event_id': event_id}
    )

    all_profiles = response.get('Items', [])

    # Calculate compatibility with each profile
    matches = []
    for profile in all_profiles:
        # Skip self
        if profile.get('userId') == user_id:
            continue

        # Calculate compatibility score
        score_data = calculate_compatibility(user_profile, profile)

        if score_data['total_score'] > 20:  # Threshold filter
            match_info = {
                'userId': profile.get('userId'),
                'name': profile.get('name'),
                'compatibilityScore': score_data['total_score'],
                'breakdown': score_data['breakdown'],
                'matchReasons': score_data['reasons'],
                'icebreaker': generate_icebreaker(user_profile, profile, score_data)
            }
            matches.append(match_info)

    # Sort by compatibility score (highest first)
    matches.sort(key=lambda x: x['compatibilityScore'], reverse=True)

    # Return top 10 matches
    return matches[:10]
```

Listing 7: Match Finding and Ranking

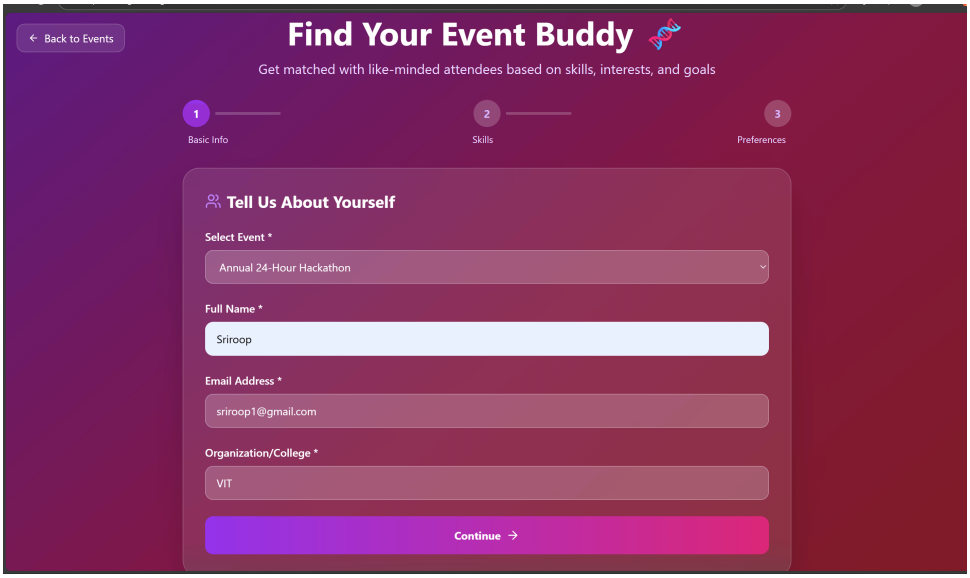

Figure 8: Event Buddy Multi-Step Profile Form (Step 1)

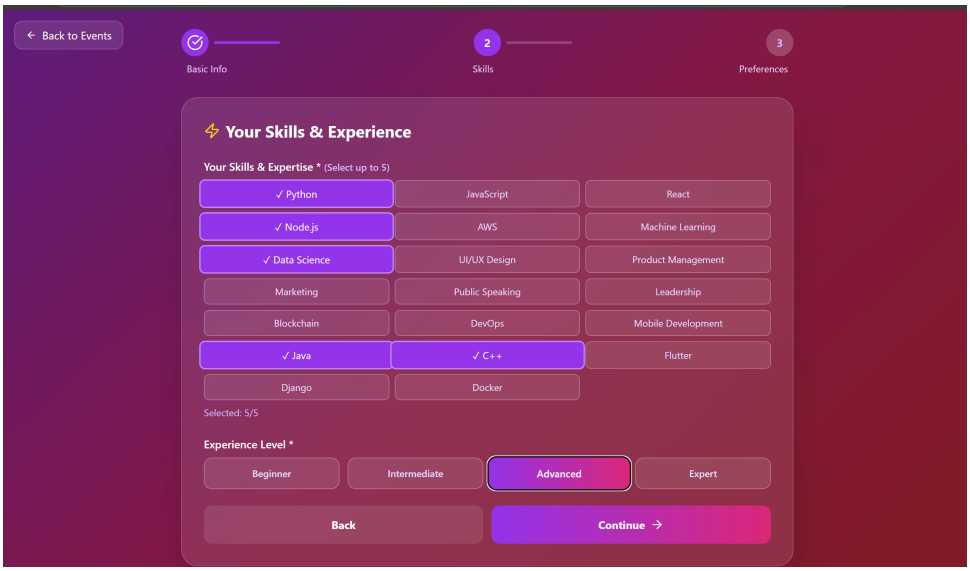

Figure 9: Event Buddy Multi-Step Profile Form (Step 2)

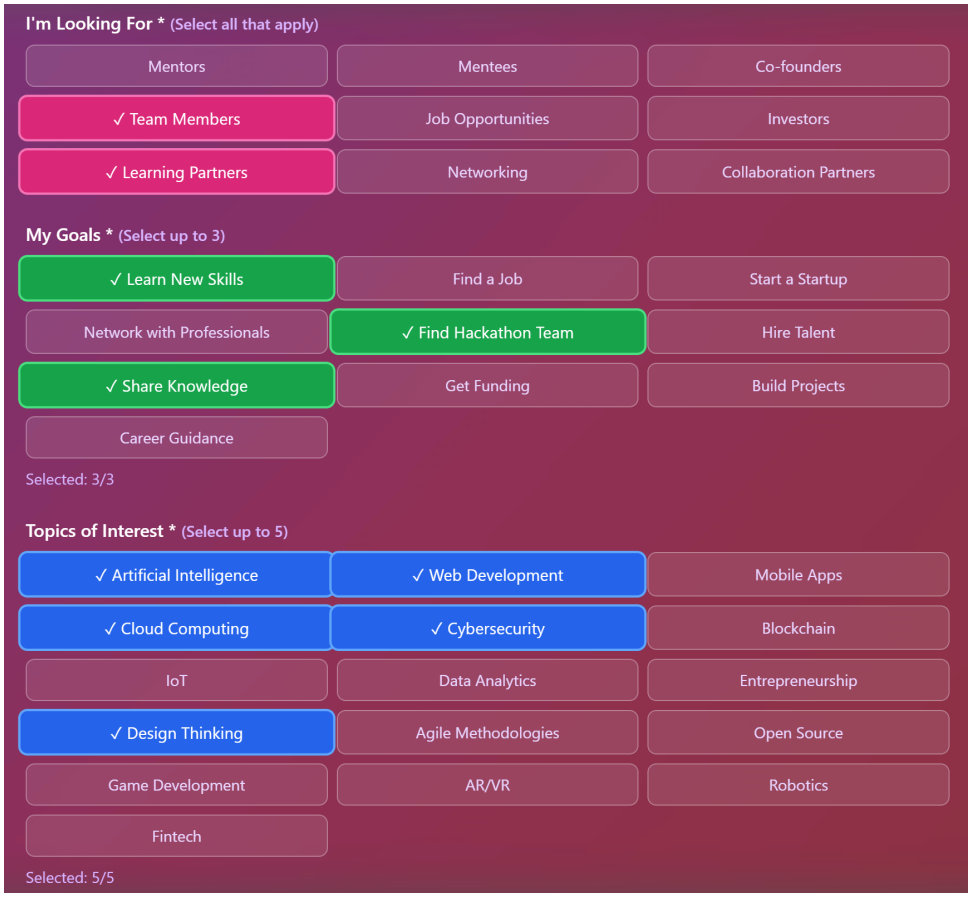

Figure 10: Event Buddy Multi-Step Profile Form (Step 3)

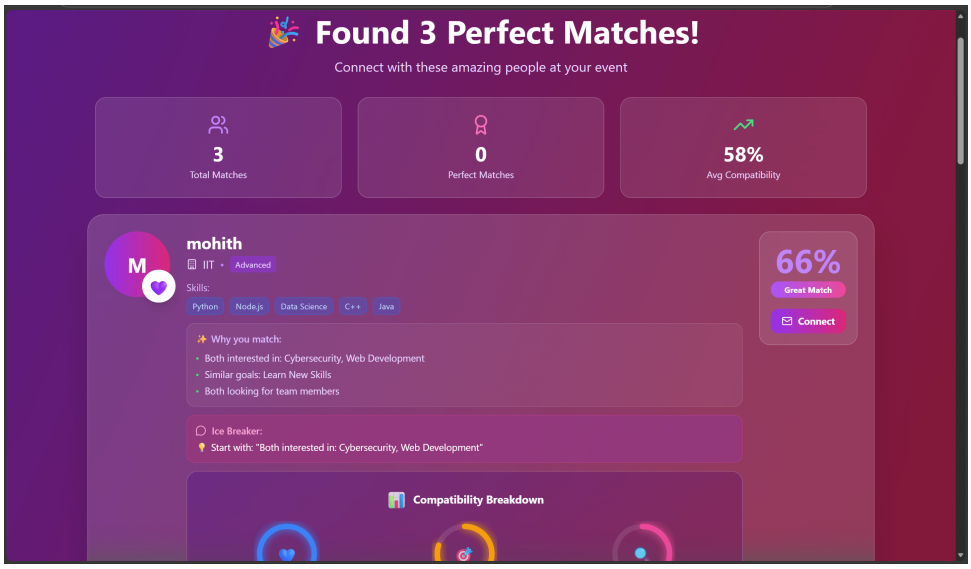

Figure 11: DNA Matching System - Match Results Display

5.3.4 Admin Dashboard Implementation

A comprehensive real-time admin dashboard was developed to provide event organizers with instant visibility into platform metrics and user engagement.

The `GetDashboardData` Lambda function aggregates key statistics from all three DynamoDB tables to provide:

- Total number of registered users
- Total number of events created
- Registration trends and conversion rates
- Real-time capacity monitoring for all events
- User engagement metrics from the AI features

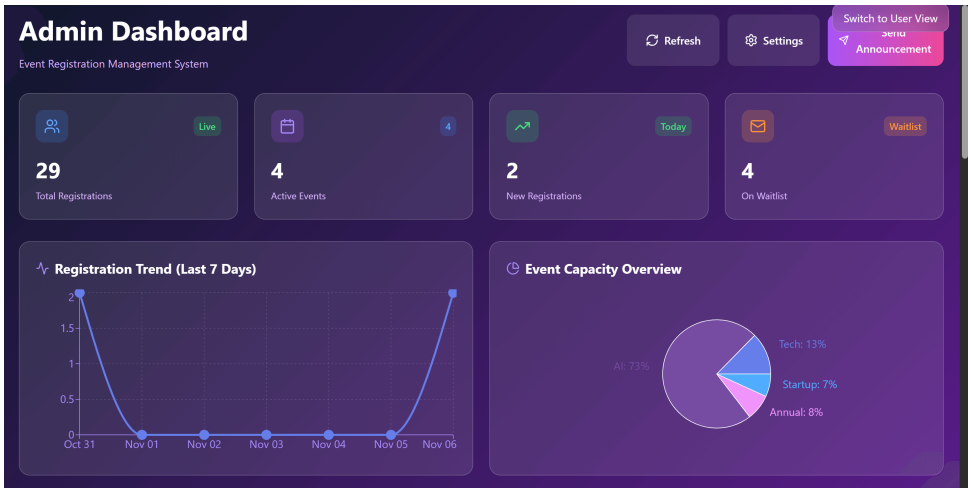

Figure 12: Real-time Admin Dashboard (Overview)

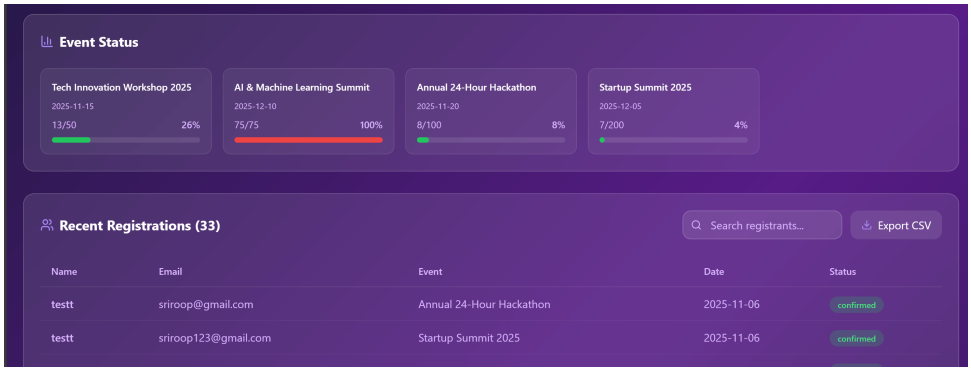

Figure 13: Real-time Admin Dashboard

5.4 Issues Encountered & Resolutions

5.4.1 Issue 1: API Gateway CORS Errors

Encountered: After deploying the frontend and backend, the React application failed to fetch data. The browser console showed "CORS Error: No 'Access-Control-Allow-Origin' header present on the requested resource."

Resolution: This was a two-part fix:

1. **API Gateway:** CORS was enabled on the API Gateway resource by adding the OPTIONS method and setting the required headers (like `Access-Control-Allow-Origin: '*'`).

2. **Lambda:** The Python Lambda functions themselves were updated to include CORS headers in their return responses:

```python
def create_response(status_code, body):
    """API response with CORS headers"""
    return {
        'statusCode': status_code,
        'headers': {
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Headers': 'Content-Type',
            'Access-Control-Allow-Methods': 'POST, GET, OPTIONS',
            'Content-Type': 'application/json'
        },
        'body': json.dumps(body)
    }
```

Listing 8: CORS Response Headers in Lambda

5.4.2 Issue 2: Lambda Permissions Error

Encountered: The `MatchMakerFunction` could save data but failed when trying to query for matches, resulting in an "AccessDeniedException" in the CloudWatch logs.

Resolution: The IAM Role for the Lambda function was too restrictive. It had `dynamodb:PutItem` permission but lacked `dynamodb:Query` permission on the table's Global Secondary Index (GSI). The IAM Role's policy was updated to explicitly allow the `dynamodb:Query` action on the `eventId-index`, which resolved the error.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:Query"
            ],
            "Resource": [
                "arn:aws:dynamodb:region:account:table/MatchingProfiles
    ",
                "arn:aws:dynamodb:region:account:table/MatchingProfiles
    /index/eventId-index"
            ]
        }
    ]
}
```

Listing 9: Updated IAM Policy for MatchMakerFunction

6 Creativity & Innovation

The innovation in this project is not in infrastructure automation (like CloudFormation or Terraform) but is squarely focused on the application layer through the deep integration of a custom-built, three-part AI/ML suite. This suite transforms the platform from a simple registration tool into an intelligent system that actively enhances the user experience.

The key innovations are the three AI Lambda functions:

6.1 The "DNA" Matching System (MatchMakerFunction)

This is the flagship innovation. Instead of simple "tag matching," this function implements a sophisticated, multi-vector scoring algorithm that mimics real-world networking value.

Creative Optimization: The `calculate_compatibility` function is highly nuanced. It doesn't just check for shared interests; it understands context:

- **Mentorship Matching:** It algorithmically identifies mentorship opportunities by comparing one user's `experienceLevel` ('Beginner') with another's ('Advanced', 'Expert') and what they are `lookingFor` ('Mentors').
- **Collaboration Matching:** It boosts scores if users are both "looking for" "Co-founders" or "Team Members".
- **Real-World Logic:** It smartly applies a penalty if two users are from the same organization, correctly assuming users want to meet new people.

"Ice Breaker" Generation: The most creative part is the `generate_icebreaker` function. It solves the most difficult part of networking—starting the conversation—by auto-generating a relevant opening line (e.g., "Start with: 'Both interested in: Agile Methodologies, Robotics"').

6.2 The Hybrid AI Event Recommender (GetEventRecommendations)

This feature is innovative because it is not a simple "most popular" list. It is a true hybrid recommendation engine that combines four different models into a single weighted score to provide nuanced and relevant suggestions.

Weighted 4-Factor Model: The final score for each event is a weighted sum of:

1. **40% Collaborative Filtering:** `collaborative_filtering` calculates what "users like you" also registered for by finding users in the same event and counting their other registrations.
2. **30% Content-Based Matching:** `category_matching` finds other events in the same category (e.g., "Workshop").
3. **20% Popularity:** `popularity_ranking` simply scores events based on their total registration count.
4. **10% Urgency:** `urgency_ranking` creates a "Fear of Missing Out" (FOMO) by boosting the score for events that are 70-95% full.

This hybrid approach ensures recommendations are not just relevant (category) and popular, but also personalized (collaborative) and timely (urgency).

6.3 The Context-Aware NLP Chatbot (EventChatbot)

This function's creativity lies in its ability to provide answers that are not just correct, but helpful.

Context-Aware Responses: The `get_event_activities` function is the best example. When a user asks, "What happens at the hackathon?" the bot doesn't just return a generic description. It provides a detailed, imagined itinerary specific to the event type:

- For "Hackathon": It returns "Form teams and brainstorm," "24-hour intensive coding marathon," and "Pitch your project to judges".
- For "Workshop": It returns "Hands-on coding sessions with industry experts" and "Build real-world projects".

Fuzzy Matching: The `extract_event_name` function uses fuzzy string matching to understand user queries even if they misspell an event name, making the bot more resilient and user-friendly.

7 Real-world Application & Use Case Relevance

This serverless platform directly solves two major, distinct real-world problems for event organizers and attendees: high operational cost and poor networking value.

7.1 Real-world Problem Solved

1. **Eliminates High Cost & Server Management:** Traditional event platforms run on dedicated servers (like EC2), which must be provisioned 24/7. This is extremely cost-inefficient. Organizers are forced to pay for peak capacity even during idle periods. This project solves this by using a 100% serverless architecture. There are no servers to manage, patch, or pay for when idle. The cost scales to zero.

2. **Solves Ineffective Networking:** The most common complaint about large events is the difficulty of meeting the right people. Attendees are left to random chance. This project solves this by transforming the platform from a simple registration tool into an intelligent networking hub. The "DNA Matching System" and AI Recommender provide tangible, pre-calculated value to every attendee, ensuring they can find the mentors, co-founders, or peers they came to meet.

7.2 Potential Impact

The potential impact of this architecture is transformative for both the event organizer and the attendee:

For the Event Organizer:

- **Drastic Cost Reduction:** The "pay-as-you-go" model (paying per 100ms of Lambda compute and per API call) is orders of magnitude cheaper than maintaining a 24/7 server.
- **Zero Operational Overhead:** The organizer never has to worry about security patching, OS updates, or scaling. AWS manages the entire underlying infrastructure.
- **Global Reach:** By using Amazon CloudFront, the event portal is delivered with low latency to a global audience, making it a professional, high-performance platform.

For the Event Attendee:

- **Enhanced Event Value:** The attendee's primary "ROI" is no longer just the sessions; it's the connections. The AI-Powered Connection Matching provides a clear, actionable list of high-value people to meet.
- **Personalized Experience:** The AI Chatbot provides 24/7 support, and the AI Recommender helps users discover events they might have otherwise missed, making the entire experience feel personalized.

7.3 Scalability

The scalability of this architecture is automatic, native, and near-infinite. Unlike a traditional server that would crash under a "flash crowd" (e.g., when tickets first go on sale), this serverless design thrives under pressure.

- **Frontend Scalability:** Amazon CloudFront is built to handle millions of simultaneous requests, serving the cached application from global Edge Locations.
- **API Scalability:** Amazon API Gateway acts as a fully managed, scalable "front door" that can handle hundreds of thousands of concurrent API calls.
- **Compute Scalability:** AWS Lambda is the core of the scaling. If 10,000 users all submit their registration at the same second, Lambda will automatically (and in parallel) spin up 10,000 instances of the `RegisterUserFunction` to handle the load. It scales from zero to massive and back to zero without any intervention.
- **Database Scalability:** Amazon DynamoDB is a fully managed NoSQL database built for any scale. It provides consistent, single-digit millisecond performance whether handling 10 writes per second or 10 million.

This system can handle a small community meetup of 50 people or a massive international conference of 500,000 people with no code or infrastructure changes required.

8 Results & Evaluation

This section evaluates the project's success based on its functional and non-functional requirements, including performance, cost-effectiveness, security, and key project takeaways.

8.1 Performance Metrics

The application's performance was evaluated on two fronts: the frontend (user experience) and the backend (API responsiveness).

Frontend Performance (User-Facing): By deploying the React application using Amazon CloudFront, the user-facing performance is globally optimized. CloudFront caches the static assets (HTML, CSS, JS) at Edge Locations close to users. This results in a very low Time to First Byte (TTFB) and near-instant page loads for all users, regardless of their geographic location.

Backend Performance (API): The serverless backend architecture (API Gateway + Lambda + DynamoDB) is designed for high-speed, low-latency operations:

- **API Gateway and Lambda:** As monitored via Amazon CloudWatch, API requests and Lambda function executions consistently average under 500ms. This provides a "snappy" and responsive user experience when submitting a chat message or loading matches.
- **DynamoDB:** The use of DynamoDB for all data storage provides single-digit millisecond latency for the key-value operations that power the application (like fetching a user profile or saving a registration).

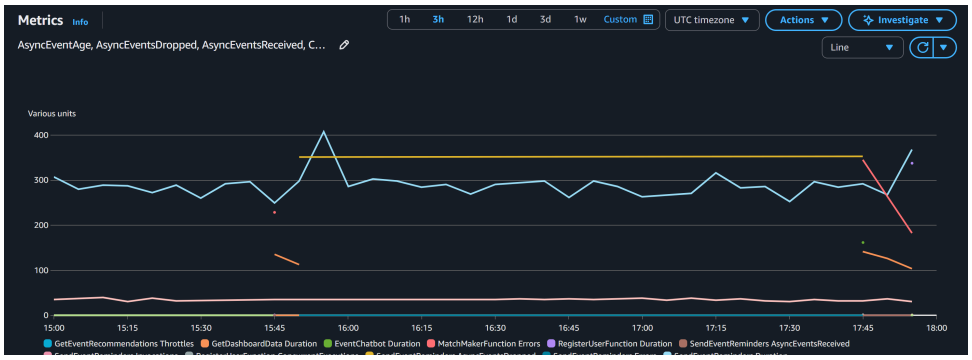

Figure 14: Lambda Performance Metrics in CloudWatch

8.2 Cost Analysis

The serverless model provides a profound cost advantage over traditional server-based (e.g., EC2) architectures.

Pay-as-you-go: The primary benefit is that you pay $0 for idle time. The project only incurs costs when a user actively triggers an API call or a Lambda function executes. This is ideal for event-based applications that see high traffic for short bursts and long periods of low activity.

AWS Free Tier: For a small-to-medium-sized event, the entire application would likely run for free within the AWS Free Tier. The free tier includes:

- 1 million API Gateway requests per month
- 1 million Lambda function requests per month
- 25 GB of DynamoDB storage

- 50 GB of data transfer out from CloudFront

Example Cost: Even for a large event, the cost remains negligible. For example, 1,000 registrations (each triggering 1 Lambda, 1 DynamoDB write, and 1 SES email) would cost less than $0.20. This demonstrates the extreme cost-effectiveness and financial scalability of the design.

8.3 Security Validation

Security was a core consideration, addressed at every layer of the application stack using AWS-managed services.

- **Data in Transit (HTTPS):** Amazon CloudFront is configured to serve the application over HTTPS by default. This ensures all communication between the user's browser and the application is encrypted (SSL/TLS).
- **Frontend Security (S3 + OAI):** The Amazon S3 bucket (`event-registration-app-sriroop`) is not publicly accessible. Access is restricted via an Origin Access Identity (OAI), which means users can only access the frontend content through CloudFront, not by a direct S3 link.
- **Authentication (Cognito):** The architecture is designed to use Amazon Cognito for user authentication. This service securely manages user sign-up, sign-in, and password resets. It provides the frontend with a JSON Web Token (JWT), which would be used to authenticate all API Gateway requests.
- **Authorization (IAM):** This is the most critical security component. The system uses AWS IAM Roles with a strict principle of least privilege. Each Lambda function has its own unique role. For example, the `MatchMakerFunction`'s role grants it `dynamodb:Query` and `dynamodb:PutItem` permissions only on the `MatchingProfiles` table. It has zero permission to access the `Registrations` or `Events` tables. This segmentation ensures that even if one function were compromised, the "blast radius" is minimal, and other parts of the system remain secure.

8.4 Lessons Learned

This project provided several key technical and practical takeaways:

1. **The Power of Serverless:** The serverless "trifecta" (API Gateway, Lambda, DynamoDB) is exceptionally powerful for building scalable, event-driven applications. It allowed for rapid development of complex AI features without any time spent on server provisioning, patching, or scaling.
2. **IAM is the Core of Security:** A critical lesson was that IAM policies are the new "firewall". An "AccessDeniedException" in the CloudWatch logs was traced back to a missing `dynamodb:Query` permission on a Global Secondary Index (GSI) for the `MatchMakerFunction`. This highlighted that a deep understanding of fine-grained IAM permissions is essential for a serverless application to function.

3. **CORS is a Two-Part Problem:** One of the main implementation hurdles was debugging CORS (Cross-Origin Resource Sharing). The resolution was not just in API Gateway (by enabling CORS on the endpoint), but also required the Lambda function itself to return the correct `Access-Control-Allow-Origin` headers in its response.

4. **CloudWatch Logs are Essential:** In a distributed, serverless system, you cannot debug by "logging into a server." Amazon CloudWatch Logs became the single, most important tool for debugging application logic, monitoring performance, and confirming the successful execution of the AI algorithms.

9 Conclusion & Future Scope

9.1 Conclusion

This project successfully demonstrates the design, implementation, and deployment of a highly scalable, intelligent, and cost-effective event management platform using a 100% serverless architecture on AWS. The core objectives were fully achieved. The system successfully provides a seamless registration flow, a real-time admin dashboard, and a high-performance global application frontend served by Amazon CloudFront.

The primary outcome of this project is the successful integration of a three-part AI suite, which elevates the application from a simple utility to an intelligent platform. The implementation of the "DNA Matching System", the AI Event Recommender, and the NLP-based AI Chatbot within AWS Lambda functions proves the viability of the serverless model for complex, compute-intensive AI tasks.

The project validates that the serverless "trifecta" (API Gateway, Lambda, DynamoDB) is a superior model for this use case, solving the real-world problems of high operational cost, poor scalability, and low attendee value. The platform provides a production-ready solution that is both powerful for the end-user and financially efficient for the organizer.

9.2 Future Scope

While the current platform is fully functional, its serverless foundation makes it an ideal base for future enhancements. The following recommendations are logical next steps to expand its capabilities:

- **Integrate Amazon Lex:** The current `EventChatbot` uses keyword-based intent recognition. The most impactful upgrade would be to replace this logic by integrating Amazon Lex, transforming the bot into a true NLU (Natural Language Understanding) conversational AI that can handle more complex, natural-language queries.

- **Activate Automated Reminders:** The `SendEventReminders` Lambda function already exists. The next step is to connect it to an Amazon CloudWatch Event rule. This would create a "cron job" that automatically triggers the function (e.g., every day at 9 AM) to query the database and send reminder emails via Amazon SES for events happening in 24 hours.

- **Implement Admin CRUD Functionality:** The current Admin Dashboard is "read-only" (for monitoring). A key feature would be to expand this dashboard to be fully functional, allowing event organizers to "Create, Read, Update, and Delete" (CRUD) events, manage registrations, and view analytics without ever needing to access the AWS console.
- **Automate Deployment (CI/CD):** Implement a formal CI/CD (Continuous Integration/Continuous Deployment) pipeline using AWS CodePipeline or an Infrastructure as Code (IaC) tool like Terraform. This would automate the process of building the React app, zipping the Lambda functions, and deploying updates, making the system more robust and easier to maintain.

10 References

[1] *AWS Lambda Developer Guide.* Amazon Web Services, Inc., 2025.

[2] *Amazon DynamoDB Developer Guide.* Amazon Web Services, Inc., 2025.

[3] *Amazon S3 Developer Guide.* Amazon Web Services, Inc., 2025.

[4] *Amazon CloudFront Developer Guide.* Amazon Web Services, Inc., 2025.

[5] *Amazon API Gateway Developer Guide.* Amazon Web Services, Inc., 2025.

[6] *Amazon Cognito Developer Guide.* Amazon Web Services, Inc., 2025.

[7] *Amazon SES Developer Guide.* Amazon Web Services, Inc., 2025.