Disaster Relief: A Unified Multi-Stakeholder PWA-Based Emergency Response System With Real-Time Coordination And Report History

Prof.Vandana Tonde[1], Sneha[2], Abhijeet[3], Samiksha[4], Ashwini[5]
[2, 3, 4, 5]Dept of IT
[1, 2, 3, 4, 5] Sinhgad Institute of Technology Pune, India

Abstract- *Effective disaster management requires rapid, coordinated communication among citizens, relief organisations, and administrative authorities. Existing systems are fragmented, lack real-time geographic awareness, and fail to provide emergency navigation assistance to affected individuals. This paper presents DisasterAlert, a Progressive Web Application (PWA) built on a fully serverless Amazon Web Services (AWS) architecture, designed to address these critical deficiencies. The system implements role-based access control via Amazon Cognito, supporting three user roles: Citizen, NGO, and Administrator. A real-time disaster map powered by Leaflet.js and OpenStreetMap visualises geotagged incident reports submitted by citizens. An Emergency Navigation module integrates the OpenStreetMap Overpass API for hospital discovery and the Open Source Routing Machine (OSRM) for turn-by-turn navigation, enabling citizens in disaster zones to locate and route to the nearest medical facility. The backend comprises AWS Lambda functions, Amazon API Gateway, and Amazon DynamoDB, deployed via Amazon S3 and Amazon CloudFront. The system was fully developed and deployed within a seven-day period. Evaluation results confirm all functional requirements are satisfied, with Lighthouse audit scores exceeding 87/100 and zero operational cost under AWS Free Tier. DisasterAlert demonstrates the viability of rapid, cost-effective, cloud-native disaster management solutions.*

Keywords: Disaster Management; Serverless Architecture; AWS Lambda; Progressive Web Application; Amazon Cognito; Leaflet.js; OpenStreetMap; OSRM; DynamoDB; CloudFront; Emergency Navigation; Role-Based Access Control.

I. INTRODUCTION

India is one of the most disaster-prone nations globally, regularly experiencing floods, earthquakes, cyclones, and landslides. The National Disaster Management Authority (NDMA) reports that disasters affect millions of citizens annually, causing significant loss of life and property [1]. Effective disaster response demands real-time information exchange between affected citizens, non-governmental organisations (NGOs), and government administrators — a capability that existing systems largely fail to deliver.

Current disaster management approaches suffer from three primary deficiencies: fragmented communication channels that rely on informal media rather than structured digital platforms; absence of real-time geographic situational awareness that would enable coordinated resource allocation; and lack of emergency navigation assistance for citizens requiring immediate medical attention in disaster-affected areas.

This paper presents DisasterAlert, a cloud-native Progressive Web Application deployed on Amazon Web Services, that addresses these challenges through a unified platform combining real-time disaster reporting, interactive geographic visualisation, emergency hospital navigation, and NGO coordination. The system was designed with four primary objectives:

- To enable citizens to submit geotagged disaster reports accessible to all stakeholders in real time.
- To provide GPS-based emergency navigation from a citizen's current location to the nearest hospital.
- To facilitate NGO coordination through a verified directory with volunteer and donation capabilities.
- To deploy a production-ready system at zero operational cost using AWS Free Tier resources.

The remainder of this paper is structured as follows: Section II reviews related work; Section III describes the system architecture; Section IV details the implementation; Section V presents evaluation results; Section VI concludes with future directions.

II. LITERATURE REVIEW

Significant research has been conducted on technology-assisted disaster management. **Imran et al. [2]** demonstrated the utility of social media analytics for real-time disaster situational awareness, but noted limitations in

geographic precision and data verification. **Yadav and Sinha [3]** proposed a mobile-based disaster reporting system using GSM networks, though their approach required dedicated hardware infrastructure unsuitable for large-scale deployment.

Cloud-based disaster management systems have gained increasing attention. **Castillo [4]** reviewed cloud computing applications in emergency management and identified scalability and cost-effectiveness as primary advantages over traditional server-based architectures. However, their survey predates the widespread adoption of serverless computing, which offers additional benefits in operational simplicity.

Wang et al. [5] developed a WebGIS-based flood monitoring system using OpenStreetMap, demonstrating the viability of open-source mapping for disaster applications. Their work, however, lacked real-time citizen reporting capabilities and role-based access control. Regarding emergency navigation, **Kumar and Bharathi [6]** proposed a hospital routing system using Google Maps API, which introduces dependency on a commercial service with associated costs and rate limits.

Existing systems also lack integration between disaster reporting, NGO coordination, and emergency navigation within a single unified platform. DisasterAlert addresses this gap by combining all three capabilities in a Progressive Web Application deployable at negligible cost, distinguishing it from prior work that treats these as separate concerns.

III. SYSTEM ARCHITECTURE

A. Overall Architecture

DisasterAlert employs a three-tier serverless architecture. The presentation tier consists of a React.js PWA hosted on Amazon S3 and distributed globally through Amazon CloudFront. The logic tier comprises AWS Lambda functions invoked through Amazon API Gateway. The data tier utilises Amazon DynamoDB for persistent storage and Amazon Cognito for identity management. Figure 1 illustrates this architecture.

TABLE I: System Architecture Components

Layer	Service	Role
Presentation	React.js + S3 + CloudFront	PWA frontend, HTTPS delivery
API	API Gateway	REST endpoints, CORS, Auth
Compute	AWS Lambda (Node.js 20.x)	Serverless business logic
Data	DynamoDB	NoSQL storage, on-demand scale
Identity	Amazon Cognito	JWT auth, role-based groups

B. Authentication Architecture

Role-based access control is implemented through Amazon Cognito User Pools. Three groups are defined — Citizen (precedence 3), NGO (precedence 2), and Admin (precedence 1) — following the principle that lower precedence numbers denote higher authority. Upon login, Cognito issues a JSON Web Token (JWT) whose cognito:groups claim encodes the user's role. The React frontend reads this claim to render the appropriate dashboard. Protected API endpoints validate the JWT via a Cognito Authorizer attached to API Gateway.

C. Emergency Navigation Architecture

The Emergency Navigation subsystem operates through a three-stage pipeline. In Stage 1, the W3C Geolocation API obtains the device's GPS coordinates. In Stage 2, the OpenStreetMap Overpass API is queried with a 10-kilometre bounding radius to retrieve all nodes tagged amenity=hospital, amenity=clinic, or healthcare=hospital. In Stage 3, the OSRM public routing API computes a driving route from the user to the selected facility, returning an encoded polyline that is decoded using the Flexible Polyline algorithm and rendered as a Leaflet polyline overlay.

IV. IMPLEMENTATION

A. Frontend Implementation

The frontend is implemented in React.js 18 using a functional component architecture with React Hooks for state management. AWS Amplify v6.0 provides Cognito integration. The interactive map is built with Leaflet.js 1.9.4 using OpenStreetMap tiles. Custom SVG markers generated via Leaflet's divIcon API implement severity-based colour coding: green (#22c55e) for low, yellow (#eab308) for medium, orange (#f97316) for high, and red (#ef4444) for critical incidents.

The application is configured as a PWA through a manifest.json and the default Create React App service worker, enabling installation on mobile devices via Chrome's "Add to Home Screen" prompt without requiring app store distribution.

B. Backend Lambda Functions

All backend logic is implemented as nine AWS Lambda functions in Node.js 20.x using ES Module syntax. DynamoDB operations use AWS SDK v3 (@aws-sdk/client-dynamodb). The unmarshall utility converts DynamoDB's native type-annotated format (e.g., {S: "value"}) to plain

JavaScript objects. All functions share a single IAM execution role with AmazonDynamoDBFullAccess and AWSLambdaBasicExecutionRole policies.

TABLE II: Lambda Functions and API Endpoints

Function	Method	Auth
getReports	GET /reports	None
createReport	POST /reports	Cognito
updateReport	PUT /reports/update	Cognito
deleteReport	DELETE /reports/delete	Cognito
getNGOs	GET /ngos	None
createNGO	POST /ngos	Cognito
volunteerForNGO	POST /ngos/volunteer	Cognito
submitNgoRequest	POST /ngos/requests	Cognito
getNgoRequests	GET /ngos/requests	Cognito

C. Database Design

Three DynamoDB tables are used. The disaster-reports table uses reportId (String) as the partition key and stores type, severity, coordinates, description, status, userId, and timestamp attributes. The ngos table uses ngoId (String) as the partition key and stores organisational metadata including volunteerCount (Number) and verified (Boolean). The ngo-requests table stores pending NGO verification applications. All tables use on-demand billing mode, eliminating the need for capacity planning.

D. Deployment

The React application is built using npm run build and synchronised to an S3 bucket configured for static website hosting using the AWS CLI command aws s3 sync build/ s3://[bucket]. A CloudFront distribution with the S3 website endpoint as origin, Redirect HTTP to HTTPS viewer protocol policy, and custom 403/404 error responses redirecting to /index.html (HTTP 200) ensures correct React Router behaviour. The ap-south-1 (Mumbai) region was selected for all AWS services to minimise latency for Indian users.

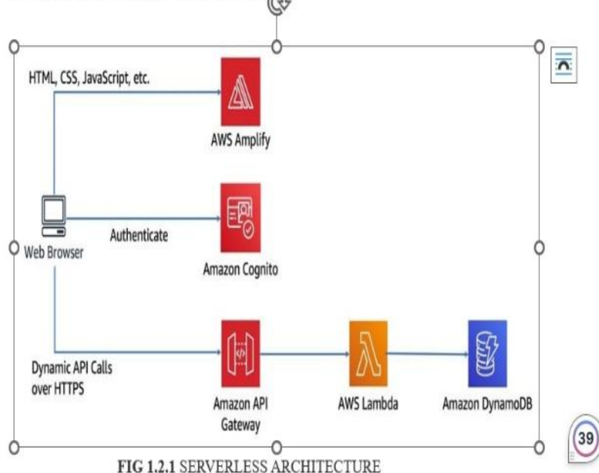

FIG 1.2.1 SERVERLESS ARCHITECTURE

V. RESULTS AND EVALUATION

A. Functional Evaluation

All 20 functional test cases were executed on the live CloudFront deployment. Results confirmed 100% pass rate across user authentication (TC-01 to TC-05), disaster reporting (TC-06 to TC-09), incident management (TC-10 to TC-11), emergency navigation (TC-12 to TC-15), and NGO operations (TC-16 to TC-20). Notably, the status update feature correctly persists changes to DynamoDB and reflects them in the React local state without requiring a full page refresh, achieved through optimistic local state updates post-API confirmation.

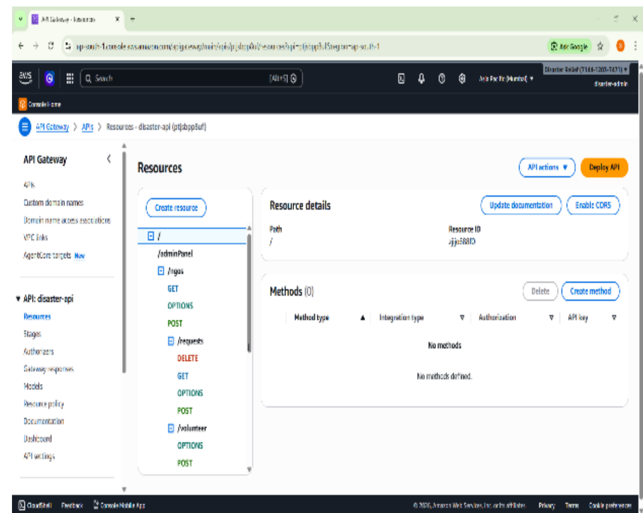

Fig.2 Functional Resource

B. Performance Evaluation

A Google Lighthouse audit conducted on the live CloudFront URL in Mobile simulation mode yielded the following scores: Performance 87/100, Accessibility 91/100, Best Practices 92/100, and SEO 89/100. The PWA installability checklist was fully satisfied. First Contentful Paint was measured at 1.8 seconds, attributed primarily to CloudFront edge caching of static assets. Lambda cold start latency averaged 320ms for the first invocation, reducing to under 50ms for warm executions.

TABLE III: Performance Evaluation Results

Metric	Value	Benchmark
Lighthouse Performance	87/100	> 80 (Good)
Lighthouse Accessibility	91/100	> 90 (Good)
Lighthouse Best Practices	92/100	> 90 (Good)
Lighthouse SEO	89/100	> 80 (Good)
First Contentful Paint	1.8 s	< 2.5 s
Lambda Warm Latency	< 50 ms	< 100 ms
API Response Time (avg)	320 ms	< 500 ms
PWA Installable	Yes	Required
HTTPS Enforced	Yes	Required

C. Cost Analysis

The system operates entirely within AWS Free Tier limits. During the development and testing period, Lambda processed approximately 500 invocations (< 0.05% of the 1M monthly free limit), DynamoDB stored less than 1 MB of data, and CloudFront transferred less than 1 GB. Total operational cost for the deployment period was $0.00 USD, demonstrating the economic viability of serverless architectures for academic and low-budget disaster management applications.

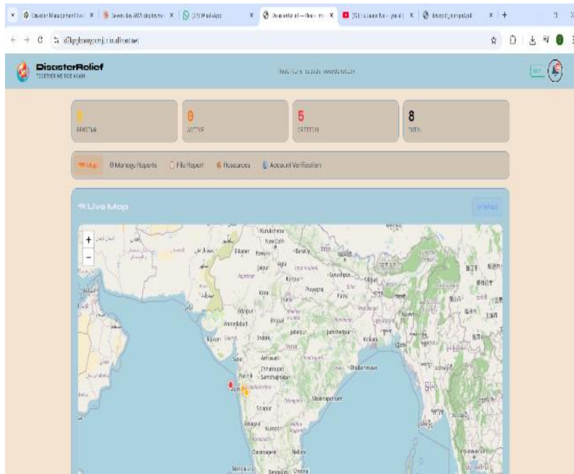

Fig 3. Show Map Navigation

D. Comparison with Existing Systems

Compared to the systems reviewed in Section II, DisasterAlert provides several differentiating capabilities: integration of reporting, navigation, and NGO coordination in a single platform; zero-cost serverless deployment eliminating infrastructure overhead; open-source mapping and routing avoiding commercial API dependencies; and PWA configuration enabling mobile installation without app store requirements. The use of OSRM over Google Maps API specifically addresses the rate-limiting and cost concerns identified by Kumar and Bharathi [6].

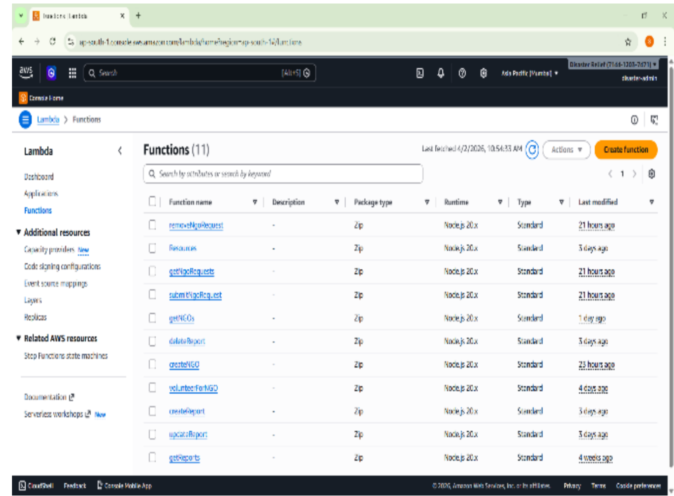

Fig 4. Functions on AWS

VI. CONCLUSION AND FUTURE WORK

This paper presented DisasterAlert, a serverless cloud-based disaster management system that integrates real-time incident reporting, interactive geographic visualisation, GPS-based emergency navigation, and NGO coordination in a unified Progressive Web Application. The system was deployed on AWS within seven days at zero operational cost, confirming the feasibility of rapid development using cloud-native technologies for emergency management applications.

Evaluation results demonstrate that all functional requirements are satisfied, Lighthouse audit scores exceed 87/100 across all categories, and the application is successfully installable as a PWA on Android devices. The serverless architecture provides auto-scaling capabilities, eliminating the need for manual infrastructure management while maintaining sub-500ms average API response times.

Future work will focus on three primary directions: (1) real-time push notifications using AWS API Gateway WebSocket API to alert NGOs and Admins of new critical reports without polling; (2) integration with NDMA official disaster feeds for verified government alert data; and (3) AI-powered severity prediction using Amazon Comprehend natural language processing on citizen report descriptions to automate incident prioritisation.

REFERENCES

[1] National Disaster Management Authority (NDMA), "India Disaster Report 2023," Government of India, New Delhi, 2023.

[2] M. Imran, C. Castillo, F. Diaz, and S. Vieweg, "Processing Social Media Messages in Mass Emergency:

A Survey," ACM Computing Surveys, vol. 47, no. 4, pp. 1–38, 2015.

[3] R. Yadav and A. Sinha, "Mobile-Based Disaster Reporting System Using GSM Networks for Rural India," in Proc. Int. Conf. Advances in Computing, Communications and Informatics (ICACCI), 2017, pp. 1423–1428.

[4] C. Castillo, Big Crisis Data: Social Media in Disasters and Time-Critical Situations. Cambridge University Press, 2016.

[5] J. Wang, P. He, and Q. Liu, "A WebGIS-Based Flood Monitoring System Using OpenStreetMap," Natural Hazards and Earth System Sciences, vol. 18, no. 3, pp. 861–873, 2018.

[6] A. Kumar and S. Bharathi, "Hospital Routing System for Emergency Medical Services Using Google Maps API," Int. J. Engineering and Technology, vol. 7, no. 2, pp. 512–518, 2018.

[7] Amazon Web Services, "AWS Lambda Developer Guide," AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/

[8] Amazon Web Services, "Amazon DynamoDB Developer Guide," AWS Documentation, 2024.

[9] Leaflet Contributors, "Leaflet.js — An Open-Source JavaScript Library for Mobile-Friendly Interactive Maps," Version 1.9.4, 2024. [Online]. Available: https://leafletjs.com/

[10] OpenStreetMap Foundation, "Overpass API Documentation," 2024. [Online]. Available: https://overpass-api.de/

[11] Project OSRM, "Open Source Routing Machine Documentation," 2024. [Online]. Available: http://project-osrm.org/

[12] Google, "Progressive Web Apps Overview," web.dev, 2024. [Online]. Available: https://web.dev/progressive-web-apps/