

# Real-Time Serverless Chat Application

## Introduction:

The **Real-Time Serverless Chat Application** is a cloud-based messaging platform designed to enable users to communicate instantly without managing any servers. This project leverages **AWS serverless technologies** to provide scalability, reliability, and cost efficiency. By integrating services such as **AWS Lambda**, **DynamoDB**, **API Gateway (WebSocket)**, **IAM**, and **S3**, the system ensures smooth real-time communication between users with minimal latency.

## Objective:

The main objective of this project is to develop a **fully serverless chat system** that allows real-time message exchange between users while maintaining scalability and security. The focus is on using AWS-managed services to eliminate infrastructure management and provide a highly available and efficient communication platform.

## Key goals:

- Enable real-time messaging through WebSocket connections.
- Store and retrieve chat data efficiently using DynamoDB.
- Use AWS Lambda for backend logic without maintaining servers.
- Secure access through IAM roles and permissions.
- Host the frontend interface using S3 for easy deployment.

## System Architecture:

### Components & Responsibilities:

#### Frontend (S3 static site)

- Hosts the UI (HTML/CSS/JS).
- Opens and maintains a WebSocket connection to the WebSocket API endpoint.
- Sends commands/events (e.g., connect, sendMessage, disconnect).
- Renders incoming messages and presence updates.

#### API Gateway — WebSocket API

- Acts as the WebSocket endpoint for clients.
- Routes messages to Lambda functions based on route keys (e.g., \$connect, \$disconnect, sendMessage).
- Provides a management API to push messages from backend to connected clients.

## AWS Lambda (backend handlers)

- \$connect handler: authenticate (optional), register connection in DynamoDB.
- \$disconnect handler: clean up connection, update presence.
- sendMessage handler: validate message, persist to DynamoDB, broadcast to room participants using API Gateway management API (or enqueue to other Lambdas).

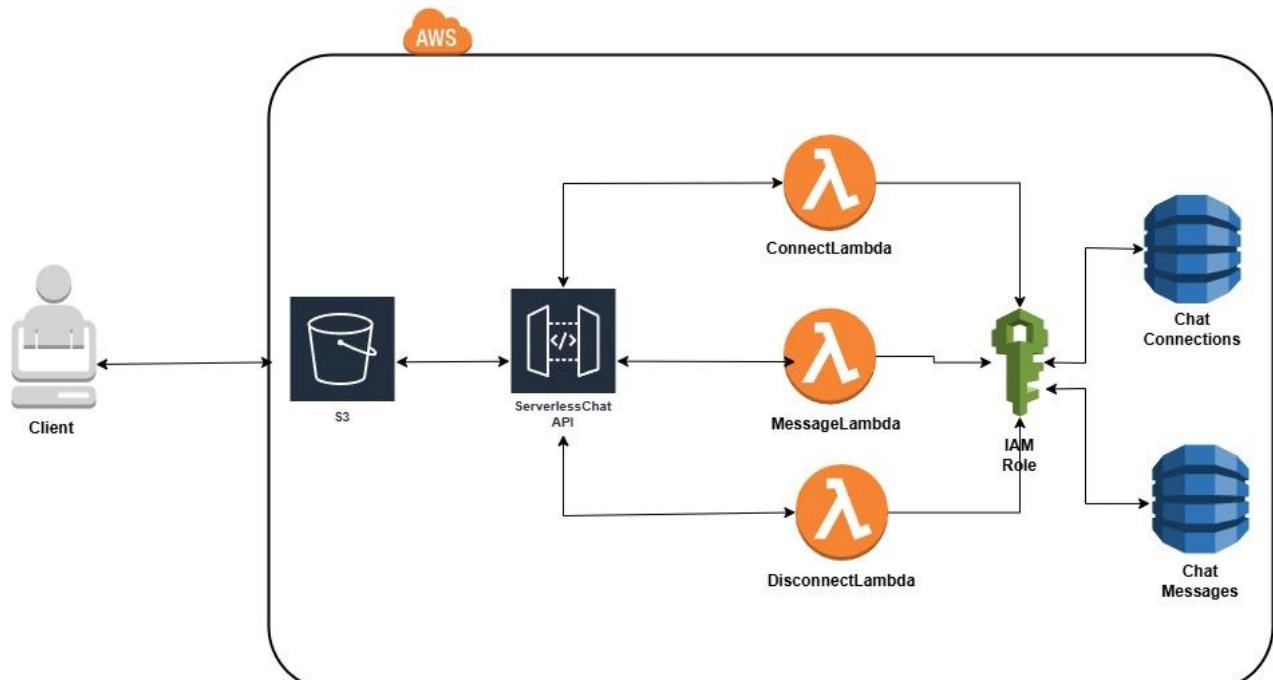
## DynamoDB

- Stores chat messages (partitioned by roomId or conversationId) with timestamps.
- Stores connection records and presence metadata (connectionId -> userId, lastSeen, currentRoom).
- Designed with appropriate keys and indexes for efficient queries (e.g., GSI for recent messages).

## IAM

- Fine-grained IAM roles for Lambda (permissions to read/write DynamoDB and call API Gateway management API).
- Least privilege for S3 read (public/private hosting) and any admin roles.

## Diagram:



## Steps Involved in this Project:

### Step1:

- ✓ Set Up the DynamoDB Table:
- ✓ Created a Two DynamoDB table

The screenshot shows the Amazon DynamoDB console interface. On the left, there's a navigation sidebar with options like Dashboard, Tables, Explore items, PartiQL editor, Backups, Exports to S3, Imports from S3, Integrations, Reserved capacity, and Settings. Under the 'Tables' section, it says 'Tables (2)'. The main area displays a table with two rows. The columns are Name, Status, Partition key, Sort key, Indexes, Replication Regions, Deletion protection, Favorite, and Read cap. The first row is for 'ChatConnections' with a Partition key of 'ConnectionID (\$)' and a Sort key of '-'. The second row is for 'ChatMessages' with a Partition key of 'MessageID (\$)' and a Sort key of '-'. Both rows show 'Active' status and 'Off' for deletion protection. The table has a header with filters for 'Any tag key' and 'Any tag value'. At the top right, there are buttons for 'Actions', 'Delete', and 'Create table'. The browser address bar shows 'eu-north-1.console.aws.amazon.com/dynamodbv2/home?region=eu-north-1#tables'. The bottom of the screen shows the Windows taskbar with various pinned icons.

### Step 2: Create IAM role for Lambda Function:

Created an IAM role for Lambda with permissions to:

```
# Access DynamoDB tables.
```

Policies:

- ✓ AmazonDynamoDBFullAccess
- ✓ AWSLambdaBasicExecutionRole
- ✓ AmazonApiInvokeFullAccess

Luffy

Allows Lambda functions to call AWS services on your behalf.

**Summary**

Creation date: November 05, 2025, 10:16 (UTC+05:30)

Last activity: 1 hour ago

ARN: arn:aws:iam::672454134884:role/Luffy

Maximum session duration: 1 hour

**Permissions**

Permissions policies (4) Info

You can attach up to 10 managed policies.

Policy name	Type	Attached entities
AmazonAPIGatewayInvokeFullAccess	AWS managed	1
AmazonDynamoDBFullAccess	AWS managed	1
API	Customer inline	0
AWSLambdaBasicExecutionRole	AWS managed	1

Permissions boundary (not set)

## IAM Role For Lambda

### Step 3: Create a Lambda Function:

Created Lambda functions in Python to handle:

- ConnectLambda
- DisconnectLambda
- MessageLambda

Functions (3)

Function name	Description	Package type	Runtime	Last modified
DisconnectLambda	-	Zip	Python 3.10	9 hours ago
ConnectLambda	-	Zip	Python 3.10	9 hours ago
MessageLambda	-	Zip	Python 3.10	9 hours ago

## Step 4: API Creation With AWS API Gateway :

Created a WEBSOCKET API in API Gateway.

The screenshot shows the AWS API Gateway console. On the left, there's a sidebar with 'API Gateway' selected. Under 'APIs', it lists 'Custom domain names', 'Domain name access associations', and 'VPC links'. Below that are 'Usage plans', 'API keys', 'Client certificates', and 'Settings'. The main area is titled 'APIs (1/1)' and shows a table with one row. The row contains the 'Name' (ServerlessChatAPI), 'ID' (xxona4ob3d), 'Protocol' (WebSocket), 'API endpoint type' (Regional), and 'Created' date (2025-11-05). There are buttons for 'Delete' and 'Create API' at the top right of the table. The bottom of the screen shows the Windows taskbar with various pinned icons.

Serverless Named API Websocket

## Add Routes:

- \$connect
- \$disconnect
- SendMessage (Custom)

The screenshot shows the AWS API Gateway console. The left sidebar has 'API: ServerlessChatAPI' expanded, showing 'Routes' selected. The main area is titled 'Routes' and shows three routes listed: '\$connect', '\$disconnect', and 'sendMessage'. Each route has a 'Route request' section with a flow diagram: Client → Route request → Integration request → Lambda integration. There are tabs for 'Route request', 'Integration request', 'Integration response', and 'Route response'. The '\$connect' route also has an 'ARN' field with the value 'arn:aws:execute-api:eu-north-1:672454134884:xxona4ob3d/\*/\$connect'. At the top right, there are 'Delete API' and 'Deploy API' buttons. The bottom of the screen shows the Windows taskbar.

Three Routes

## Step 5: Hosted The Frontend on Amazon S3 :

- Created an S3 bucket to host the static web application files (HTML, CSS, JS).
- Enabled Static Website Hosting and uploaded all necessary frontend assets.

The screenshot shows the AWS S3 console interface. On the left, a sidebar lists various S3 features like General purpose buckets, Storage Lens, and CloudShell. The main area is titled 'serverless-chat-01' and shows the 'Objects' tab selected. A single object, 'index.html', is listed with its details: Type: html, Last modified: November 5, 2025, 12:04:09 (UTC+05:30), Size: 5.6 KB, and Storage class: Standard. There are also buttons for Copy S3 URI, Copy URL, Download, Open, Delete, Actions, Create folder, and Upload.

## Final Output:

The screenshot shows a web browser window with the title 'Not secure serverless-chat-01.s3-website.eu-north-1.amazonaws.com'. The page displays a simple chat interface with a purple header bar. The header says 'Serverless Chat' and 'Real-time using API Gateway WebSocket'. Below the header, it says 'Connected to chat server'. A message bubble from 'Sridhar' at 08:43 PM contains the text 'Hey!'. At the bottom, there's an input field with placeholder 'Write a message...' and a 'Send' button. The browser's taskbar at the bottom shows various pinned icons.

## **Conclusion:**

- ❖ This project successfully demonstrates the power of AWS serverless architecture in building a scalable and real-time communication platform.
- ❖ By using DynamoDB, Lambda, API Gateway (WebSocket), IAM, and S3, the chat application delivers instant messaging without the need to manage servers.
- ❖ The solution is highly cost-effective, reliable, and easy to maintain, making it an ideal model for modern real-time applications.