AWS Full-Stack Deployment with Load Balancing: A Scalable Architecture Report

1. Aim

The aim of this project is to successfully design and deploy a modern, full-stack application environment on Amazon Web Services (AWS) that adheres to the best practices for **high availability (HA)**, **fault tolerance**, and **dynamic scalability**.

2. Objective

Upon completion of this deployment, the infrastructure must achieve the following objectives:

1. **Distribute traffic** efficiently across multiple application servers using an Application Load Balancer (ALB).
2. **Ensure application resilience** by deploying instances across a minimum of two Availability Zones (AZs).
3. **Automatically scale** compute resources (EC2 instances) up or down based on real-time application load metrics, optimizing cost and performance.
4. Maintain application uptime through continuous **health checks** performed by the ALB and Auto Scaling Group (ASG).

3. Theory and Core AWS Services

The architecture utilizes a scalable, stateless 3-Tier model:

A. The 3-Tier Architecture

Layer	Function	AWS Services
Presentation/Web	Handles initial user requests. (Load Balancer)	Application Load Balancer (ALB)
Application/Logic	Processes business logic, hosts the full-stack backend (API).	Amazon EC2 via Auto Scaling Group (ASG)
Data	Stores persistent application data (e.g., users, transactions).	Amazon RDS (Relational Database Service) / DynamoDB

B. Core AWS Services for Scalability

1. **Amazon Virtual Private Cloud (VPC):** Provides an isolated, private virtual network where all resources are launched, logically separating them from other AWS customers. This includes configuring subnets across multiple Availability Zones (AZs) for high availability.
2. **Application Load Balancer (ALB):** A Layer 7 (HTTP/HTTPS) load balancer that intelligently routes client traffic based on content, host headers, or URL paths. It is internet-facing and responsible for health-checking application instances.
3. **Auto Scaling Group (ASG):** A service that automatically adjusts the number of EC2 instances in a fleet to maintain desired performance. Key components include:
    - **Launch Template:** Defines the instance configuration (AMI, instance type, security groups, and *User Data* script).
    - **Scaling Policy:** Defines the conditions (e.g., Target Tracking based on 50% CPU Utilization) that trigger scale-out (add instances) or scale-in (remove instances) actions.
4. **Target Group (TG):** A logical grouping of EC2 instances that the ALB routes requests to. The TG performs periodic health checks and ensures traffic is only sent to healthy instances.

4. Procedure (Step-by-Step Deployment)

The following steps outline the setup using the AWS Management Console:

Step 1: Network Configuration (VPC)

1. Create a new VPC with public subnets spanning at least two different Availability Zones (e.g., us-east-1a and us-east-1b).
2. Create two Security Groups:
    - **ALB-SG:** Allows inbound traffic on HTTP (Port 80) and HTTPS (Port 443) from 0.0.0.0/0 (the Internet).
    - **App-SG:** Allows inbound traffic on the application port (e.g., Port 3000 for a Node.js app) *only* from the ALB-SG (Source: Custom, ID of ALB-SG). This protects the application instances from direct internet access.

Step 2: Create Application Startup Script and Launch Template

1. **Application Code (User Data):** Prepare a shell script to be executed when an EC2 instance launches. This script installs dependencies (like Node.js or a web server) and starts the application.
2. Create an **EC2 Launch Template** defining:
    - A suitable Amazon Machine Image (AMI) (e.g., Amazon Linux 2023).
    - Instance Type (e.g., t2.micro).
    - The App-SG security group.

- Paste the Application Startup Script (User Data) into the advanced configuration.

Step 3: Configure Load Balancing Components

1. Create an **Application Load Balancer (ALB)** in the VPC, spanning the two public subnets. Attach the ALB-SG.
2. Create a **Target Group (App-TG)** targeting EC2 instances running on the application's port (e.g., 3000). Set the health check path to a simple endpoint (e.g., /health).
3. Modify the ALB's Listener (Port 80) to forward all traffic to the newly created App-TG.

Step 4: Create Auto Scaling Group (ASG)

1. Create an **Auto Scaling Group** using the Launch Template created in Step 2.
2. Define the network settings: select the VPC and the two public subnets.
3. Configure the group size:
   - Min capacity: 2 (Ensures high availability across two AZs).
   - Desired capacity: 2
   - Max capacity: 4 (Allows for scaling during traffic spikes).
4. Attach the ASG to the existing App-TG for load balancing integration.
5. Set up a **Scaling Policy** (e.g., Target Tracking Policy) to maintain average CPU utilization at 50%.

5. Code (Example Full-Stack Backend Component)

The deployment assumes a full-stack application where the backend is running on the EC2 instances. Below is a minimal Node.js/Express application that listens on Port 3000, displaying the unique hostname (Instance ID) to demonstrate load balancing:

```
// This file would be placed on the EC2 instance via the Launch Template
User Data script.

const express = require('express');

const os = require('os');

const app = express();

const port = 3000; // Must match the Target Group port



// The EC2 hostname often maps to the private IP or a unique identifier

const instanceId = os.hostname();
```

```
app.get('/', (req, res) => {
    // This is the main endpoint used by the user
    res.send(`
        <!DOCTYPE html>
        <html lang="en">
        <head>
            <title>AWS Load Balancer Test</title>
            <style>body { font-family: sans-serif; background-color:
#e6f0ff; text-align: center; padding-top: 50px; }</style>
        </head>
        <body>
            <h1>Deployment Successful!</h1>
            <p>You are currently served by:</p>
            <strong style="color: #007bff; font-size: 1.5rem;">Instance
ID: ${instanceId}</strong>
            <p>This demonstrates successful load balancing and Auto
Scaling.</p>
        </body>
        </html>
    `);
});
```

```
app.get('/health', (req, res) => {

    // This is the health check endpoint used by the ALB

    res.status(200).send('OK');

});



app.listen(port, () => {

    console.log(`Application listening on port ${port}`);

});
```

6. Output and Validation

1. **Access Test:** Copy the DNS name of the Application Load Balancer and paste it into a web browser. The browser should display the message "Deployment Successful!" and the "Instance ID."
2. **Load Balancing Proof:** Continuously refresh the browser page. The "Instance ID" displayed should toggle between the private hostnames of the two currently running EC2 instances (Min Capacity = 2), confirming that the ALB is distributing requests across both targets.
3. **Scalability Test:** Initiate a stress test (e.g., using a tool like Apache JMeter) to push the CPU utilization of the instances above the 50% threshold defined in the ASG policy.
4. **Expected Scaling Output:** Observe the EC2 console. After a few minutes, the Auto Scaling Group should automatically launch a new EC2 instance to handle the increased load, moving the desired capacity toward the maximum of 4. Once the load subsides, the ASG will terminate the excess instances (scale-in).

7. Learning Outcomes

1. **Decoupling and High Availability (HA):** Learned how to achieve fault tolerance by distributing EC2 instances across multiple Availability Zones (AZs) and using a load balancer as a single point of entry, thus preventing a single component failure from

causing application downtime.

2. **Infrastructure Elasticity:** Mastered the configuration of an Auto Scaling Group (ASG) and its integration with an ALB to enable dynamic scaling, which ensures performance under variable traffic loads while simultaneously managing cloud costs by terminating idle resources.
3. **Security Best Practices:** Understood the principle of *least privilege* by using Security Groups to restrict direct public access to the application instances (`App-SG` only allows traffic from the `ALB-SG`), thereby securing the compute layer from external threats.