Dean Moses

deanmdt@gmail.com

In this project, I’ve implemented and configured AWS Elastic Load Balancers to create a highly available and scalable web application infrastructure. This document demonstrates my understanding of AWS load balancing technologies, implementation approaches, and advanced configuration techniques.

It also demonstrates the implementation and configuration of Auto Scaling Groups (ASGs) in AWS to create a highly available, fault-tolerant, and elastically scalable web application architecture.

AWS ELB & ASG

Availability, Scalability

Contents

[AWS Load Balancing Implementation Project 2](#_Toc192530247)

[Project Overview 2](#_Toc192530248)

[Business Context 2](#_Toc192530249)

[Technologies Used 2](#_Toc192530250)

[Implementation Details 2](#_Toc192530251)

[Phase 1: Setting Up the Backend Infrastructure 2](#_Toc192530252)

[Phase 2: Implementing the Application Load Balancer 3](#_Toc192530253)

[Phase 3: Testing Fault Tolerance 3](#_Toc192530254)

[Phase 4: Advanced Load Balancer Configurations 3](#_Toc192530255)

[Technical Challenges and Solutions 5](#_Toc192530256)

[Project Outcomes 5](#_Toc192530257)

[Key Learnings 5](#_Toc192530258)

[Auto Scaling Groups in AWS: Implementation and Best Practices 6](#_Toc192530259)

[Architecture Overview 6](#_Toc192530260)

[Implementation Steps 6](#_Toc192530261)

[Creating a Launch Template 6](#_Toc192530262)

[Setting Up an Auto Scaling Group 7](#_Toc192530263)

[Integrating with Load Balancers 7](#_Toc192530264)

[Health Check Configuration 7](#_Toc192530265)

[Scaling Policies 8](#_Toc192530266)

[Manual Scaling 8](#_Toc192530267)

[Dynamic Scaling with Target Tracking 8](#_Toc192530268)

[Step Scaling 8](#_Toc192530269)

[Simple Scaling 8](#_Toc192530270)

[Scheduled Scaling 9](#_Toc192530271)

[Predictive Scaling 9](#_Toc192530272)

[Testing 9](#_Toc192530273)

[Stress Testing 9](#_Toc192530274)

[Monitoring Scaling Activities 9](#_Toc192530275)

[Demonstration Results 9](#_Toc192530276)

[Best Practices and Recommendations 10](#_Toc192530277)

[Conclusion 10](#_Toc192530278)

# AWS Load Balancing Implementation Project

## Project Overview

In this project, I’ve implemented and configured AWS Elastic Load Balancers to create a highly available and scalable web application infrastructure. This document demonstrates my understanding of AWS load balancing technologies, implementation approaches, and advanced configuration techniques.

It also demonstrates the implementation and configuration of Auto Scaling Groups (ASGs) in AWS to create a highly available, fault-tolerant, and elastically scalable web application architecture.

## Business Context

Modern web applications require high availability, fault tolerance, and the ability to scale with user demand. Load balancers are a critical component in achieving these requirements by distributing incoming traffic across multiple backend servers. This project demonstrates my ability to implement such solutions in AWS environments.

## Technologies Used

* Amazon EC2 (Elastic Compute Cloud)
* Elastic Load Balancing (ELB)
  + Application Load Balancer (ALB)
  + Network Load Balancer (NLB)
  + Gateway Load Balancer
* Security Groups
* Target Groups
* Health Checks

## Implementation Details

### Phase 1: Setting Up the Backend Infrastructure

The first step was to create the backend servers that would handle the application traffic:

1. Launched two EC2 instances named “My First Instance” and “My Second Instance”
2. Used Amazon Linux 2 on t2.micro instances for cost-effectiveness
3. Configured a security group (Launch-Wizard-1) to allow HTTP and SSH traffic
4. Implemented a bootstrap script using EC2 user data to initialize a simple web server on each instance
5. Verified direct access to each instance via their public IP addresses

Each instance displayed a simple “Hello World” message with a unique identifier, confirming they were operational and serving content independently.

### Phase 2: Implementing the Application Load Balancer

After establishing the backend infrastructure, I implemented an Application Load Balancer to distribute traffic:

1. Created an Application Load Balancer (DemoALB) configured for internet-facing traffic
2. Deployed the ALB across multiple availability zones for high availability
3. Created a dedicated security group (demo-sg-load-balancer) with appropriate rules
4. Configured HTTP listener on port 80
5. Created a target group (demo-tg-alb) and registered both EC2 instances
6. Verified load balancer functionality by observing traffic distribution between instances

The ALB successfully distributed traffic between both instances in a round-robin fashion, as evidenced by the changing instance identifiers in the response upon refreshing the browser.

### Phase 3: Testing Fault Tolerance

To demonstrate the fault tolerance capabilities of the load balancer:

1. Stopped one EC2 instance to simulate a failure
2. Observed the ALB’s health check mechanism marking the stopped instance as unhealthy
3. Confirmed that all traffic was automatically redirected to the remaining healthy instance
4. Restarted the stopped instance and verified it was automatically returned to service
5. Observed traffic being distributed to both instances again

This test confirmed that the load balancer could detect failed instances and automatically route traffic to healthy instances, providing continuous service availability.

### Phase 4: Advanced Load Balancer Configurations

#### Enhanced Security Configuration

I improved the security posture of the infrastructure by implementing a more restrictive traffic flow:

1. Modified the EC2 instance security group to only accept traffic from the load balancer
2. Removed direct public access to EC2 instances
3. Verified that direct access to instances via public IPs was no longer possible
4. Confirmed that access through the ALB continued to function correctly

This configuration ensures that all web traffic must flow through the load balancer, providing an additional security layer.

#### Custom Routing Rules

I implemented advanced routing rules in the ALB to handle specific request patterns:

1. Created a custom routing rule with path-based routing
2. Configured the rule to return a custom error response for requests to the “/error” path
3. Set the rule priority to ensure proper processing order
4. Tested the rule by accessing the “/error” path, confirming the custom error response

This demonstrates the ability to implement complex routing logic based on URL patterns, which is essential for modern web applications.

#### Session Stickiness Configuration

To support applications that require session persistence:

1. Enabled sticky sessions at the target group level
2. Configured load balancer-generated cookies with a one-day expiration
3. Tested session stickiness by repeatedly refreshing the browser
4. Verified that requests from the same browser session were consistently routed to the same backend instance
5. Examined network traffic to confirm the presence and functionality of session cookies

I also analyzed the different types of sticky session implementations: - Application-based cookies (custom or load balancer-generated) - Duration-based cookies (AWSALB for ALB, AWSELB for Classic Load Balancer)

#### Cross-Zone Load Balancing

I studied and configured cross-zone load balancing to understand traffic distribution across availability zones:

1. Analyzed the default behavior of different load balancer types:
   * ALB: Cross-zone balancing enabled by default
   * NLB and Gateway LB: Cross-zone balancing disabled by default
2. Configured cross-zone balancing settings on different load balancer types
3. Understood the cost implications of cross-zone data transfer

This configuration ensures even distribution of traffic across all instances regardless of their availability zone, which is important for balanced scaling.

#### SSL/TLS Certificate Implementation

To secure the application traffic, I explored SSL/TLS certificate implementation:

1. Added HTTPS listener on port 443 for the ALB
2. Configured TLS listener for the NLB
3. Explored certificate management options:
   * Amazon Certificate Manager (ACM)
   * IAM certificate management
   * Direct certificate import

This demonstrates my understanding of securing web traffic and implementing industry-standard encryption protocols.

## Technical Challenges and Solutions

1. **Challenge**: Ensuring instances remain accessible through the load balancer while restricting direct access **Solution**: Implemented security group rules that only allow traffic from the load balancer security group
2. **Challenge**: Managing session persistence for applications requiring user state **Solution**: Configured sticky sessions with appropriate cookie settings and expiration policies
3. **Challenge**: Understanding the performance implications of cross-zone load balancing **Solution**: Analyzed the default behaviors and cost implications for different load balancer types

## Project Outcomes

This implementation achieved several key business objectives:

1. **High Availability**: The application remains available even if individual instances fail
2. **Scalability**: The infrastructure can easily scale by adding more instances to the target group
3. **Enhanced Security**: Traffic flow is controlled and restricted to appropriate paths
4. **Flexibility**: Advanced routing rules support complex application requirements
5. **Session Management**: Support for stateful applications through sticky session configuration

## Key Learnings

This project provided valuable insights into AWS load balancing technologies:

1. Different load balancer types (ALB, NLB, Gateway LB) serve different use cases based on protocol and performance requirements
2. Health checks are crucial for automatic failover and high availability
3. Security groups can be linked to create secure traffic flows
4. Advanced routing rules enable sophisticated application architectures
5. Session stickiness must be balanced against even load distribution
6. Cross-zone load balancing affects both performance and cost
7. SSL/TLS implementation is straightforward but requires proper certificate management

# Auto Scaling Groups in AWS: Implementation and Best Practices

Auto Scaling Groups (ASGs) in AWS provide a powerful mechanism to ensure application availability and cost efficiency by automatically adjusting the number of EC2 instances based on demand. This project demonstrates the implementation of ASGs integrated with Application Load Balancers (ALBs) and various scaling policies to achieve optimal performance.

## Architecture Overview

The architecture implemented consists of:

* **Launch Template**: Defines the configuration of EC2 instances to be launched
* **Auto Scaling Group**: Manages the lifecycle of EC2 instances
* **Application Load Balancer**: Distributes traffic to healthy instances
* **Target Group**: Registers instances with the load balancer
* **Scaling Policies**: Determine when and how to scale instances
* **CloudWatch Alarms**: Trigger scaling actions based on metrics

## Implementation Steps

### Creating a Launch Template

The launch template defines the configuration for EC2 instances that will be created by the Auto Scaling Group.

1. Navigate to the EC2 console and select “Launch Templates” from the left navigation panel
2. Click “Create Launch Template”
3. Configure the template with the following settings:
   * **Name**: my-demo-template
   * **Description**: Template for ASG demo
   * **AMI**: Amazon Linux 2 (x86)
   * **Instance Type**: t2.micro (Free tier eligible)
   * **Key Pair**: EC2-tutorial (or your existing key pair)
   * **Security Group**: Select an existing security group (e.g., launch-wizard-1)
   * **Storage**: 8 GB GP2 volume
   * **User Data**: Install and configure a web server

#!/bin/bash  
yum update -y  
yum install -y httpd  
systemctl start httpd  
systemctl enable httpd  
echo "<h1>Hello World from $(hostname -f)</h1>" > /var/www/html/index.html

### Setting Up an Auto Scaling Group

1. Navigate to “Auto Scaling Groups” in the EC2 console
2. Click “Create Auto Scaling Group”
3. Configure the ASG with the following settings:
   * **Name**: Demo-ASG
   * **Launch Template**: Select the previously created template
   * **VPC and Availability Zones**: Select your VPC and multiple AZs for high availability
   * **Load Balancer Integration**: Enable
   * **Target Group**: Select your existing ALB target group
   * **Health Checks**: Enable both EC2 and ELB health checks
   * **Desired Capacity**: 1
   * **Minimum Capacity**: 1
   * **Maximum Capacity**: 1 (to be modified later for scaling demonstrations)

### Integrating with Load Balancers

Connecting your ASG with a load balancer ensures traffic is distributed across healthy instances:

1. During ASG creation, select “Attach to an existing load balancer”
2. Choose “Choose from your load balancer target groups”
3. Select your target group (e.g., demo-tg-alb)
4. This integration ensures new instances are automatically registered with the load balancer
5. When instances are terminated, they are automatically deregistered

### Health Check Configuration

Proper health check configuration ensures that unhealthy instances are replaced:

1. Enable EC2 health checks to detect instance-level failures
2. Enable ELB health checks to detect application-level failures
3. Set appropriate health check grace periods to allow instances time to initialize

## Scaling Policies

### Manual Scaling

Manual scaling allows you to adjust capacity as needed:

1. Select your ASG and click “Edit”
2. Modify the “Desired Capacity” value
3. Ensure the new desired capacity is within the minimum and maximum capacity range
4. The ASG will automatically launch or terminate instances to match the desired capacity

### Dynamic Scaling with Target Tracking

Target tracking is the simplest way to implement dynamic scaling:

1. Select your ASG and navigate to “Automatic Scaling”
2. Select “Create Target Tracking Scaling Policy”
3. Configure the policy:
   * **Policy Name**: target-tracking-policy
   * **Metric**: Average CPU Utilization
   * **Target Value**: 40%
4. The ASG will automatically adjust capacity to maintain the target metric value

Implementation example:

Metric: Average CPU Utilization  
Target: 40%  
Scale Out: When CPU > 40% for 3 consecutive data points (3 minutes)  
Scale In: When CPU < 28% for 15 consecutive data points (15 minutes)

### Step Scaling

Step scaling provides more granular control over scaling actions:

1. Create CloudWatch alarms for your scaling conditions
2. Configure step scaling policies with different adjustment amounts based on alarm thresholds
3. For example:
   * If CPU > 60%, add 2 instances
   * If CPU > 80%, add 4 instances

### Simple Scaling

Simple scaling allows basic scale out/in based on a single CloudWatch alarm:

1. Create a CloudWatch alarm for your scaling condition
2. Configure a simple scaling policy that adds or removes a specific number of instances or percentage of the group

### Scheduled Scaling

Scheduled scaling is useful for predictable workload patterns:

1. Navigate to “Scheduled Actions” in your ASG
2. Create a schedule with specific capacity settings
3. Configure the recurrence pattern (one-time, daily, weekly, etc.)
4. Specify start and end times if applicable

### Predictive Scaling

Predictive scaling uses machine learning to forecast capacity needs:

1. Navigate to “Predictive Scaling Policies”
2. Select metrics to analyze (e.g., CPU utilization)
3. Set target utilization values
4. AWS will use machine learning to analyze historical patterns and scale proactively

## Testing

### Stress Testing

To test dynamic scaling policies:

1. Connect to an EC2 instance using EC2 Instance Connect
2. Install the stress utility:

* sudo amazon-linux-extras install epel -y  
  sudo yum install stress -y

1. Run a stress test to increase CPU utilization:

* stress -c 4

1. Monitor the ASG activities and CloudWatch metrics to observe scaling actions

### Monitoring Scaling Activities

1. Navigate to the “Activity” tab in your ASG
2. Review “Activity History” to see scaling actions
3. Check CloudWatch alarms to understand what triggered the scaling
4. Monitor instance states in the “Instance Management” tab

## Demonstration Results

The following results were observed during testing:

1. **Initial State**:
   * 1 EC2 instance running
   * CPU utilization near 0%
   * Instance healthy and registered with the load balancer
2. **After Stress Test Began**:
   * CPU utilization increased to nearly 100%
   * CloudWatch alarm triggered (CPU > 40%)
   * ASG scaled out from 1 to 2 instances
   * As CPU remained high, scaled to 3 instances
3. **After Stress Test Ended**:
   * CPU utilization dropped below 28%
   * After 15 consecutive data points, scale-in action triggered
   * ASG reduced capacity from 3 to 2 instances
   * After continued low utilization, reduced to 1 instance

This demonstration confirmed the target tracking policy correctly adjusted capacity based on CPU utilization.

## Best Practices and Recommendations

1. **Right-size your instances**: Choose appropriate instance types for your workload
2. **Set appropriate thresholds**: Avoid scaling too aggressively or too conservatively
3. **Configure scale-in protection**: Prevent important instances from being terminated
4. **Use multiple AZs**: Distribute instances across availability zones for high availability
5. **Implement proper health checks**: Ensure unhealthy instances are detected and replaced
6. **Use cool-down periods**: Prevent rapid scaling oscillations
7. **Combine scaling policies**: Use scheduled scaling for predictable patterns and dynamic scaling for unexpected changes
8. **Monitor scaling activities**: Regularly review scaling actions and adjust policies as needed

## Conclusion

This project demonstrates my ability to design, implement, and configure AWS load balancing solutions that provide high availability, security, and scalability for web applications. The implementation showcases both basic functionality and advanced configurations that address real-world application requirements. By successfully completing this project, I’ve shown proficiency in AWS infrastructure, networking concepts, and security practices that are essential for modern cloud environments.

I have demonstrated the implementation of Auto Scaling Groups in AWS to create a scalable and highly available web application architecture. By configuring launch templates, integrating with load balancers, and implementing various scaling policies, we achieved automatic adjustment of capacity based on demand.

The target tracking policy effectively maintained CPU utilization around the target value, scaling out during high load and scaling in during periods of low utilization. This approach ensures optimal performance while minimizing costs by running only the necessary number of instances.

Auto Scaling Groups provide a powerful solution for handling variable workloads and ensuring application availability without manual intervention.