

Cloud Computing - Cloud Environment in Public Model Project on Auto-Scaling and Load Balancer



A Project by:

1. Dhwani Bhavankar (AIML - 034)
2. Dhruvi Ranwala (AIML - 035)
3. Hevardhan S (AIML - 046)
4. Srinivas Motepalli (AIML - 065)

### **Introduction**

In this project, we’ve built a robust and scalable infrastructure to host a Dockerized web application on AWS, leveraging the platform’s cloud-native services to ensure high availability, auto-scaling, and load balancing. The goal of this project is to automatically deploy and manage a containerized application across multiple instances to handle varying traffic loads efficiently. By utilizing AWS services like EC2 Auto Scaling, Elastic Load Balancing, and Amazon EC2, this setup can dynamically adjust resources to maintain optimal performance and ensure the application is always accessible to users.

The project showcases a modern approach to cloud deployment using containerization and automation. Every new instance launched by the auto-scaling group is configured to pull in the latest version of the web app, set up Docker, and run the application, making it an excellent example of deploying a scalable and resilient web application on AWS. This deployment can serve as the foundation for applications with high-traffic demands, and it can be easily modified or expanded as the project requirements grow.

### **Architecture**

### 

In this project, we implemented a structured approach for deploying a React-based frontend application using Docker and AWS services, as outlined below:

1. Frontend Development with React: The project starts with building the frontend application using React. This provides a robust and interactive user interface.
2. Containerization with Docker: Once the React application is developed, a Dockerfile is created to containerize the application. This step ensures that the application, along with all its dependencies, is bundled into a Docker image, which enhances portability and consistency across different environments.
3. Version Control and Repository Management: The application code and Dockerfile are then pushed to a GitHub repository. This step facilitates version control and centralized storage, enabling seamless collaboration and access for building Docker images.
4. Building the Docker Image: The Docker image is built from the Dockerfile. This image encapsulates the application and its dependencies, making it ready for deployment in a cloud environment.
5. Deployment on AWS EC2 Instance: An AWS EC2 instance is provisioned to host the application. The Docker image is pulled onto this instance, allowing the application to run within a managed and scalable infrastructure.
6. Running the Docker Image: Finally, the Docker image is run on the EC2 instance, effectively deploying the React application. This step enables the application to be accessed by end-users, leveraging AWS's scalability and operational efficiency.

### 

### **AWS Services Used**

**1. Amazon EC2 (Elastic Compute Cloud):**Amazon EC2 provides scalable virtual servers, known as instances, to host our application. For this project, EC2 instances are the backbone of our deployment, running a Docker container with the web application. We configure these instances to automatically install Docker, deploy the application, and start the container as soon as they are launched, ensuring that each instance is ready to handle requests.

**2. EC2 Auto Scaling:**Auto Scaling ensures that the number of EC2 instances adjusts dynamically based on the application’s load and demand. This means that if traffic increases, additional instances are automatically launched to meet the demand. Conversely, if traffic decreases, instances are terminated to save costs. This feature provides a cost-effective way to maintain the application's performance while handling fluctuations in usage.

**3. Elastic Load Balancing (ELB):**ELB distributes incoming traffic across multiple EC2 instances, providing redundancy and ensuring that no single instance is overwhelmed by traffic. By balancing the load across instances, it improves application availability and reliability. In our project, the load balancer ensures that traffic is evenly distributed across instances in the auto-scaling group, enhancing the system’s ability to handle high traffic efficiently.

**4. AWS CloudWatch (Monitoring):**AWS CloudWatch is used to monitor the health and performance of the EC2 instances and the load balancer. By collecting metrics such as CPU usage, memory, and network traffic, CloudWatch provides valuable insights into the system’s performance. Additionally, alerts can be set up in CloudWatch to automatically scale instances up or down based on predefined metrics, helping to optimize both performance and cost-efficiency.

**5. IAM Users:**IAM Users provide secure, controlled access to AWS resources for this project, enabling you to manage permissions specifically for tasks like accessing EC2 instances, load balancing, autoscaling. Instead of using the root account, IAM Users allow for safe, limited access, whether through the AWS Console or programmatically with access keys. This approach enhances security by defining precise permissions for each user or role involved in your deployment, making resource management safer and more efficient.

**6. Docker:**Docker was selected for this project to ensure consistent and portable application deployment. By packaging the application and its dependencies into containers, Docker eliminates environment-related issues, making it easier to move the application across development and production environments. It provides efficient resource utilization and scalability, enabling quick deployment and flexible scaling with minimal overhead. Docker also simplifies dependency management and enhances security by isolating the application, making it an ideal choice for a reliable and streamlined deployment process.

### **Implementation**

**1. Set up the Launch Template:**

* Create an EC2 launch template in AWS that specifies the configuration for instances in the autoscaling group. Include the AMI(Ubuntu), instance type(micro), security groups(HTTP, SSH), and key pair.
* Add the Docker installation and static website setup in the User Data field to automate the setup on each instance launch:

#!/bin/bash

apt update -y

apt install -y npm

apt install -y docker.io

git clone https://github.com/hevardhan/Payas.git

cd Payas

docker build -t payas:v1 .

docker run -d -p 80:80 payas:v1

**2. Create an Auto Scaling Group (ASG):**

* Define the ASG with minimum(1), maximum(5), and desired(2) instance counts, link it to the launch template.
* Enable health checks to monitor instance health regularly and Cloud Watch is enabled to monitor the health and performance of the EC2 instances and the load balancer.

**3. Set up the Load Balancer (LB):**

* Create an Application Load Balancer (ALB) in AWS, which distributes traffic
* across the instances.
* Configure Availability Zones and Security Groups with two inbound rules to enable HTTP and SSH access.
* Attach the target group with health checks to ensure instances are running properly and route traffic only to healthy instances.

**4. Define Target Group:**

* Configure Target type(Instances), Protocol:Port(HTTP:80) and anything else leave with the default option enabled.
* Set health check parameters to monitor instance health regularly.

**5. CloudWatch:**

* Set an Alarm to notify when the number of instances running exceeds a particular value.

**6. Deploy and Test:**

* Launch the ASG to automatically create instances, and verify the static website is accessible through the **load balancer’s DNS**.
* Test auto scaling by generating load and observing if new instances are added or removed.

### **Challenges**

**Dynamic Scaling and Load Management**

* **Challenge:** Ensuring that the app scales up or down to handle traffic without overloading any single server. Scaling too quickly or too slowly can lead to high costs or poor user experience.
* **Solution Strategy:** We can set up AutoScaling policies carefully, using metrics like CPU usage and network traffic to decide when to add or remove servers, and use cooldowns to prevent frequent scaling.

**Cost Optimization for AWS Resources**

* **Challenge:** Auto Scaling and other AWS services can increase costs if not well-managed, especially during periods of fluctuating traffic.
* **Solution Strategy:** Keep a close watch on resource usage and set cost-conscious scaling policies. AWS Free Tier or savings plans can help minimize costs, especially during the early testing phases.

**Reliable Load Balancing and Health Checks**

* **Challenge:** Configuring the load balancer to evenly distribute user traffic can be complex. Accurate health checks are also needed to avoid sending traffic to unstable servers.
* **Solution Strategy:** Customize health checks based on what keeps the app running smoothly. Test the load balancer under different traffic conditions to make sure it works as expected.

### **Conclusion**

This project shows a reliable way to host a static website using AWS tools. With an autoscaling group and a load balancer, the website can handle changes in traffic automatically without much manual work. The Docker setup makes it easy to deploy the site on new instances whenever needed. Autoscaling also helps control costs by adding or removing instances based on demand. Overall, this setup provides a practical and manageable approach for running a website on the cloud, ensuring it stays available and performs well under different traffic loads.