Design Doc: Highly Available and Auto-Scaling Containerized Application Deployment

1 Executive summary/Purpose

This document details the final architecture and implementation of a Capstone project designed to host a containerized application in a highly available, scalable, and secure manner on AWS. The purpose is to present the completed system, outline the technical decisions made during its construction, and serve as the primary written submission for the project. The core of the architecture uses Amazon ECS on EC2, an Application Load Balancer, an Auto Scaling Group, and AWS WAF to solve common application deployment challenges.

1.1 BACKGROUND & PROBLEM STATEMENT

The project addresses a common scenario where a developer has a finished containerized application but lacks a robust deployment strategy. The core problem is that a basic deployment often lacks the elasticity to handle variable traffic, the resilience to withstand component failure, and the security to protect against common web threats. This leads to poor performance, potential downtime, and security vulnerabilities.

1.2 CURRENT SITUATION/CURRENT DESIGN

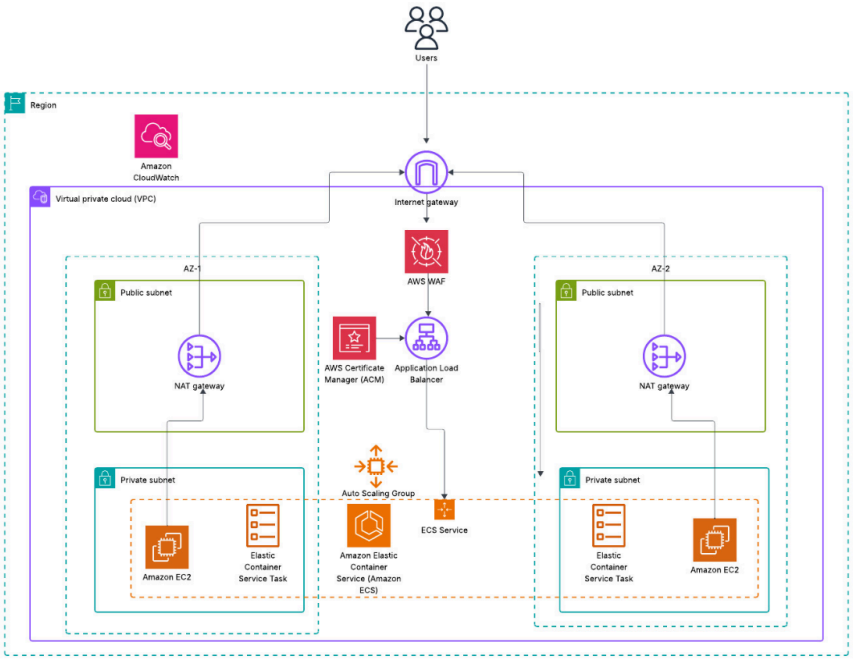
Prior to this architecture, the assumed "current design" would be a simple, single-instance deployment. This typically involves running the container on a lone EC2 instance using a tool like Docker. While functional for development, this approach has critical weaknesses in a production-like environment:

- Single Point of Failure: If the EC2 instance fails, the entire application goes down.
- No Scalability: It cannot automatically handle a sudden increase in users, leading to slow performance or crashes.
- Manual Management: All updates, scaling, and recovery actions are manual and time-consuming.
- Limited Security: It lacks a dedicated firewall to protect against application-layer attacks.

2 Proposal

The implemented solution is a comprehensive architecture that provides a resilient and automated environment for the containerized application.

2.1 ARCHITECTURE



The architecture is designed with security and high availability as its core principles, structured within an Amazon Virtual Private Cloud (VPC) that spans two Availability Zones.

A user's request to the application first passes through an **Internet Gateway** and is immediately inspected by **AWS WAF**, which filters for malicious traffic. Legitimate requests are then sent to an **Application Load Balancer (ALB)**. The ALB is deployed across public subnets in both Availability Zones, ensuring it is highly available. The ALB's primary role is to terminate the initial connection and distribute incoming requests across the application fleet.

The application itself runs as tasks within **Amazon ECS**. These tasks are hosted on a fleet of **EC2 instances** that are placed securely in private subnets. This two-tier design (public-facing ALB, private application servers) is a critical security measure, as it prevents the EC2 instances from being directly exposed to the public internet. The ALB communicates with the EC2 instances over their private IP addresses. For any outbound internet access, such as downloading software updates, the EC2 instances route their traffic through a **NAT Gateway** located in the public subnet.

2.2 ARCHITECTURE COMPONENTS

The system consists of several key AWS services working in concert:

- Amazon ECS on EC2: The application containers are managed by an ECS cluster running on a fleet of EC2 instances. This provides high availability by distributing tasks across multiple instances.
- Application Load Balancer (ALB): The ALB serves as the single entry point for all user traffic. It distributes requests evenly across the healthy application containers running in the ECS cluster.
- ECS Service Auto Scaling: The number of running application tasks is automatically managed by a target tracking scaling policy. This policy adds or removes tasks based on the average CPU utilization to match capacity with demand.
- AWS WAF (Web Application Firewall): Integrated directly with the ALB, AWS WAF inspects web traffic and blocks malicious requests before they reach the application.
- Amazon CloudWatch: Used for comprehensive monitoring. A custom dashboard displays key metrics for the cluster, ALB, and WAF, while CloudWatch Logs aggregates container logs for debugging.

3 Alternatives Considered

To validate the chosen design, several alternative architectures were considered.

- **ECS on AWS Fargate:** Fargate is a serverless compute engine for containers.
    - **Pros:** Simplifies operations by removing the need to manage the underlying EC2 instances. AWS handles all server patching and scaling.
    - **Cons:** Less granular control over the compute environment. Can be more expensive for long-running, predictable workloads compared to the EC2 launch type.
    - **Reason for Rejection:** The EC2 launch type was chosen for this project to provide deeper experience with the full ECS stack and to have explicit control over the compute layer, a key learning objective.
- **Single EC2 Instance with Docker Compose:** This represents the basic "do nothing" alternative.
    - **Pros:** Simple to set up for a single developer.
    - **Cons:** Fails to meet any of the project's requirements for scalability, high availability, or automated security. It is not a viable production solution.
    - **Reason for Rejection:** Does not solve the core problem statement.
- **AWS Elastic Beanstalk:** A higher-level Platform as a Service (PaaS) that abstracts away much of the underlying infrastructure configuration.
    - **Pros:** Very fast to get an application deployed. Handles provisioning of load balancers, auto-scaling, and deployments automatically with minimal configuration.
    - **Cons:** Less flexible than ECS. While it uses ECS under the hood for Docker deployments, you have less direct control over task definitions, service configurations, and networking.
    - **Reason for Rejection:** ECS was chosen for its flexibility and to gain direct experience with container orchestration, which is a more foundational and transferable skill.

4 Appendices

Appendix A: Lessons Learned

- **The Value of Health Checks:** Implementing both ECS task health checks and ALB health checks is critical for system reliability. The ALB health check was essential for preventing traffic from being sent to a running-but-unresponsive container, ensuring a seamless user experience during failures.
- **EC2 vs. Fargate Trade-offs:** The project provided a clear understanding of the trade-offs between control (EC2) and convenience (Fargate). For a learning objective focused on infrastructure, EC2 is superior. For rapid deployment with minimal operational overhead, Fargate would be the clear winner.
- **Effortless Security with WAF:** Integrating AWS WAF was surprisingly straightforward and provided an immediate, powerful security layer. Using AWS Managed Rules covers a vast array of common threats with just a few clicks.
- **The Importance of Monitoring:** The CloudWatch dashboard proved indispensable during load testing. Without it, it would have been impossible to visualize the auto-scaling events in real-time or to verify that traffic was being distributed evenly.

Appendix B: FAQ

This section addresses common questions about the architecture.

- **Q: How does this architecture handle container health?**
    - **A:** Health is managed at two levels: The ECS scheduler replaces any tasks that fail their internal health check, and the ALB stops sending traffic to any container that fails the ALB's health check endpoint, ensuring users are

never routed to a faulty container.

- **Q: How is auto-scaling configured to balance cost and performance?**
  - **A:** It uses a Target Tracking policy to maintain an average CPU utilization of 60% across all tasks. This automatically scales out during spikes and scales in during quiet periods. Cooldown periods prevent the system from scaling too rapidly, which controls cost.
- **Q: What are the most critical WAF rules implemented?**
  - **A:** The implementation uses the AWS Managed Rules - Core rule set to protect against OWASP Top 10 vulnerabilities, the Amazon IP reputation list to block known malicious actors, and the SQL database rule set to prevent SQL injection attacks.
- **Q: What best practices were followed for the ECS Task Definition?**
  - **A:** Key practices included defining specific CPU and memory reservations for resource stability, configuring the `awslogs` log driver for centralized logging, and using the `awsvpc` network mode to provide each task with its own dedicated network interface for enhanced security and simplified networking.

Version	Author	Status	Comments
1. 0	blboykin	Draft	Final draft for Capstone submission