DESIGN FOR A SCALABLE AWS WEB APPLICATION ARCHITECTURE

The architecture leverages a combination of AWS services to ensure scalability, cost-efficiency, and fault tolerance:

- **VPC and Subnets**: The foundation of the architecture begins with the creation of a Virtual Private Cloud (VPC) with multiple subnets across different Availability Zones (AZs). This setup ensures that resources are logically isolated and can communicate securely, while the use of multiple AZs provides redundancy.
- **Auto-Scaling Group with EC2 Instances**: EC2 instances are placed in an Auto-Scaling Group (ASG) that spans multiple subnets in different AZs. This allows the application to automatically scale in response to traffic, maintaining performance under varying loads. The ASG automatically adjusts the number of EC2 instances to match the current demand, ensuring cost efficiency by only using resources when needed.

- **Load Balancer (ALB/NLB)**: A Load Balancer is added to distribute incoming traffic evenly across the EC2 instances in different AZs. This not only balances the load but also enhances fault tolerance by redirecting traffic to healthy instances if any become unavailable. The use of an Application Load Balancer (ALB) is recommended for web applications that require Layer 7 routing based on HTTP/HTTPS, while a Network Load Balancer (NLB) is ideal for high-performance, low-latency applications.

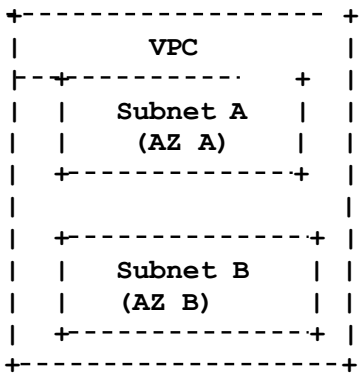
- **RDS Instance with Multi-AZ**: The architecture includes a Relational Database Service (RDS) instance configured with Multi-AZ deployment. This ensures high availability by automatically replicating the database to another AZ, providing a fail-over mechanism. This setup guarantees that in case of an AZ failure, the database is still accessible, minimizing downtime.

- **S3 Buckets**: Simple Storage Service (S3) buckets are included for static content storage. S3 is highly scalable, durable, and cost-effective, making it ideal for storing static assets like images, CSS, and JavaScript files. By offloading static content to S3, the load on EC2 instances is reduced, which improves performance and scalability.

- **Security Groups and IAM Roles**: Security Groups are configured to control inbound and outbound traffic to the EC2 instances, RDS, and Load Balancer. IAM Roles are used to manage permissions for AWS resources, ensuring that only authorized services can access critical components. This adheres to the principle of least privilege, enhancing security.

Stage 1: VPC and Subnets

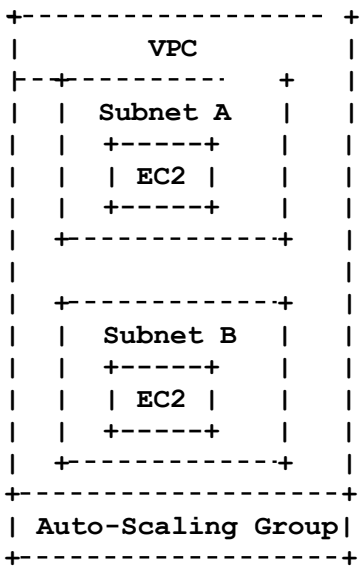
The first stage involves setting up the foundational network.



- **VPC**: The virtual private cloud where all resources are deployed.
- **Subnets**: Two subnets in different Availability Zones (AZ A and AZ B).

Stage 2: Auto-Scaling Group with EC2 Instances

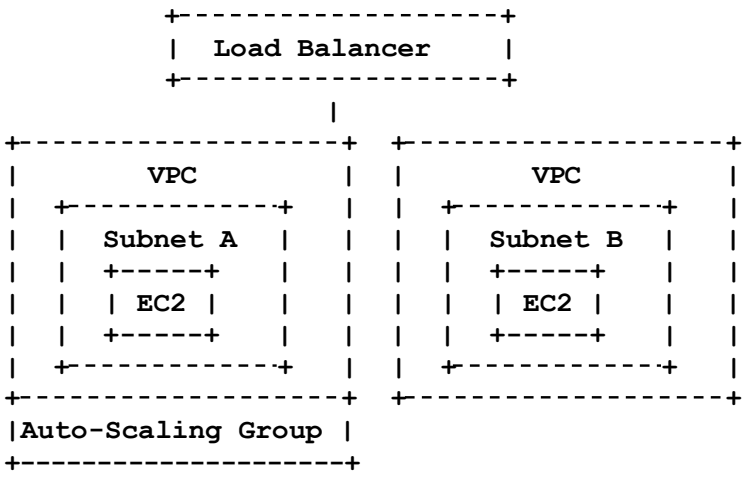
In this stage, we add EC2 instances in an auto-scaling group across the subnets.



- **Auto-Scaling Group**: Ensures that EC2 instances are added or removed based on demand.

Stage 3: Load Balancer

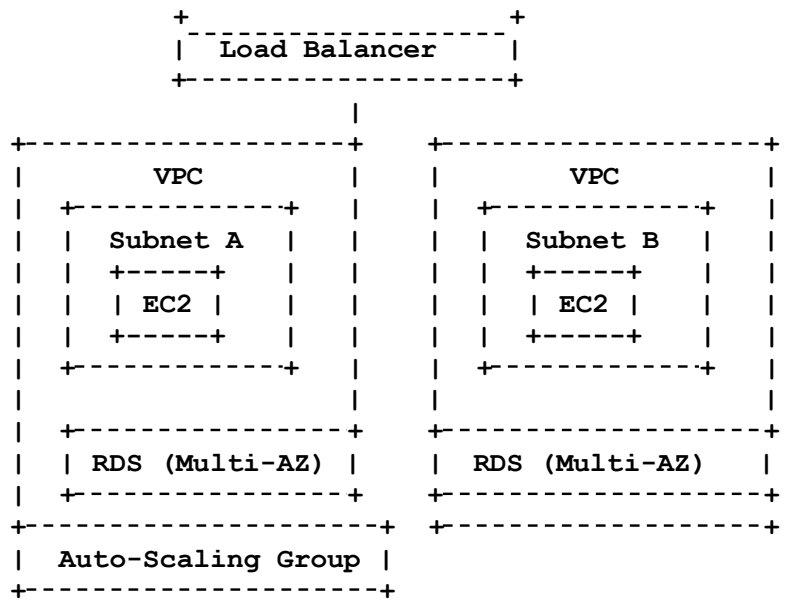
Now, we'll introduce a Load Balancer that distributes traffic across the EC2 instances.



- **Load Balancer**: Distributes incoming traffic to the EC2 instances.

Stage 4: RDS Database Instance

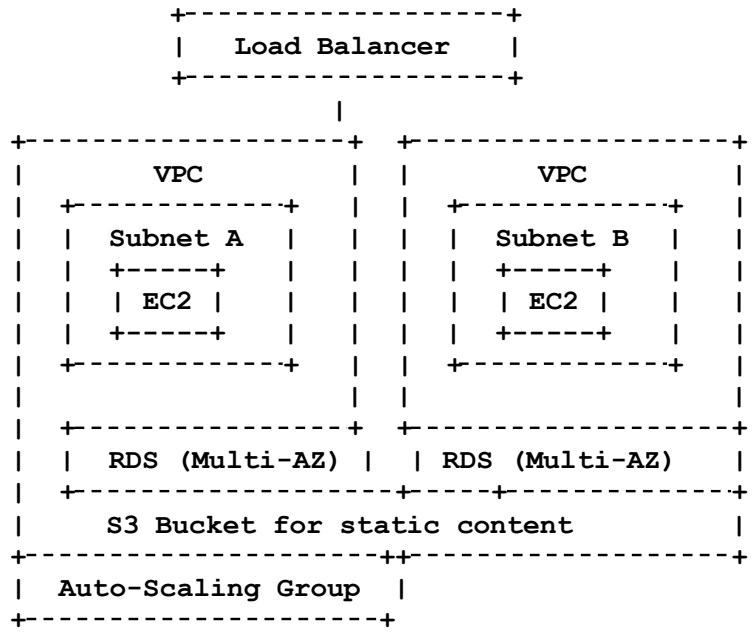
Next, we add an RDS instance configured for high availability.



- **RDS Instance**: A multi-AZ deployed database for high availability.

Stage 5: S3 Bucket

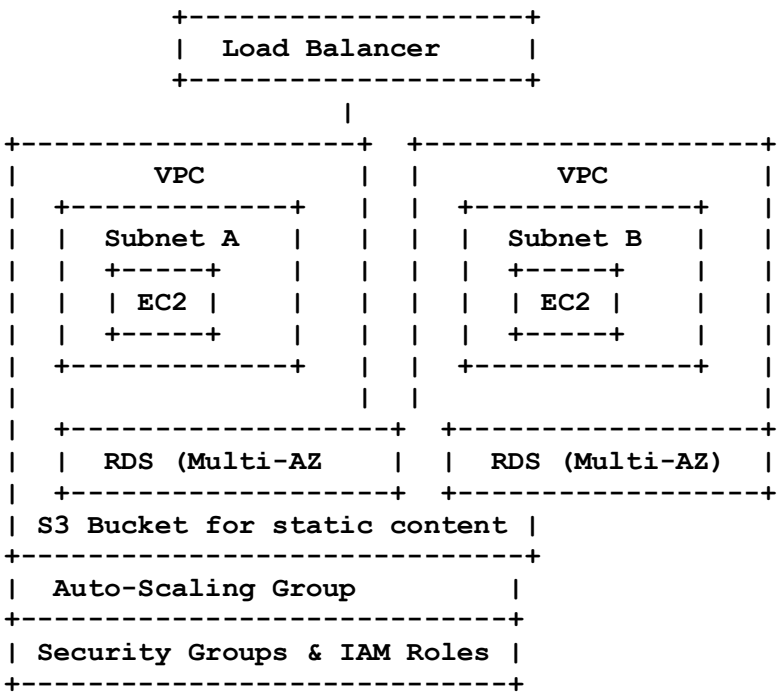
We add S3 buckets for storing static content.



S3 Buckets: Used for storing static assets like images, CSS, etc.

Stage 6: Security Measures

Finally, we add security measures, including security groups and IAM roles.



- **Security Groups**: Control access to the instances.
- **IAM Roles**: Manage permissions and access to AWS resources.

2. Best Practices

This architecture follows AWS's best practices for design:

- **High Availability and Fault Tolerance**: The architecture is designed with multiple AZs to ensure that the application can withstand the failure of an entire data center. By distributing resources across different AZs, the architecture is resilient to single points of failure.

- **Scalability**: Auto-scaling ensures that the application can handle varying traffic loads without manual intervention. The use of an ALB/NLB helps distribute the load effectively, while S3 provides a scalable solution for static content delivery.

- **Security**: Security Groups are configured to control access at the instance level, while IAM Roles ensure that only authorized services and users have access to resources. This layered security approach protects the application from external threats and unauthorized access.

- **Cost Efficiency**: The architecture uses Auto-Scaling to dynamically adjust resources based on demand, minimizing costs by running only the necessary number of instances. S3 is used for cost-effective storage of static content, reducing the need for larger and more expensive EC2 instances.

3. Terraform Script Quality

Below is a simplified Terraform script that represents the architecture:

```
provider "aws" {

  region = "us-west-2"  # Specifies the AWS region where resources will be created.

}

resource "aws_vpc" "main" {

  cidr_block = "10.0.0.0/16"  # CIDR block for the VPC, defining its IP address range.

}
```

```
resource "aws_subnet" "subnet_a" {
  vpc_id            = aws_vpc.main.id  # Associates this subnet with the VPC.
  cidr_block        = "10.0.1.0/24"  # CIDR block for this subnet, defining its IP address range.
  availability_zone = "us-west-2a"  # Availability zone for this subnet.
}
resource "aws_subnet" "subnet_b" {
  vpc_id            = aws_vpc.main.id  # Associates this subnet with the VPC.
  cidr_block        = "10.0.2.0/24"  # CIDR block for this subnet, defining its IP address range.
  availability_zone = "us-west-2b"  # Availability zone for this subnet.
}
resource "aws_security_group" "sg" {
  vpc_id = aws_vpc.main.id  # Associates this security group with the VPC.
  ingress {
    from_port   = 80  # Allows incoming traffic on port 80.
    to_port     = 80  # Allows incoming traffic on port 80.
    protocol    = "tcp"  # Specifies the TCP protocol.
    cidr_blocks = ["0.0.0.0/0"]  # Allows traffic from any IP address.
  }
}
resource "aws_instance" "web" {
  count         = 2  # Creates two EC2 instances.
  ami           = "ami-0c55b159cbfafe1f0"  # AMI ID for the instances. (Example ID)
  instance_type = "t2.micro"  # Type of the EC2 instance.
  subnet_id     = element([aws_subnet.subnet_a.id, aws_subnet.subnet_b.id], count.index)  # Assigns each
instance to one of the defined subnets.
  security_groups = [aws_security_group.sg.name]  # Associates the security group with the instances.
  user_data = <<-EOF
              #!/bin/bash
              yum install -y httpd  # Installs the Apache HTTP server.
              systemctl start httpd  # Starts the Apache HTTP server service.
              EOF
```

```
}

resource "aws_elb" "web_lb" {

  availability_zones = ["us-west-2a", "us-west-2b"]  # Availability zones for the load balancer.

  listeners {

    instance_port     = 80  # Port on the instance that the load balancer forwards traffic to.

    instance_protocol = "HTTP"  # Protocol for the connection between the load balancer and instances.

    lb_port           = 80  # Port on the load balancer that accepts incoming traffic.

    lb_protocol       = "HTTP"  # Protocol for the connection from clients to the load balancer.

  }

  instances = aws_instance.web[*].id  # Associates the EC2 instances with the load balancer.

}

resource "aws_db_instance" "default" {

  allocated_storage    = 20  # Amount of storage allocated for the RDS instance (in gigabytes).

  storage_type         = "gp2"  # Type of storage. `gp2` stands for General Purpose SSD.

  engine               = "mysql"  # Database engine for the RDS instance (MySQL).

  instance_class       = "db.t2.micro"  # Instance class for the RDS instance.

  name                 = "mydb"  # Name of the database to create.

  username             = "admin"  # Master username for the database.

  password             = "password"  # Master password for the database (use a more secure method in real
scenarios).

  multi_az             = true  # Enables Multi-AZ deployments for high availability.

  vpc_security_group_ids = [aws_security_group.sg.id]  # Associates the security group with the RDS
instance.

}

resource "aws_s3_bucket" "static_content" {

  bucket = "my-static-bucket"  # Name of the S3 bucket (must be globally unique).

}

output "load_balancer_dns_name" {

  value = aws_elb.web_lb.dns_name  # Outputs the DNS name of the load balancer.

}
```

The script starts by defining the VPC, then subnets across two AZs, a security group to control access, then EC2 instances to host the application, a classic ELB is set up to distribute traffic across the EC2 instances, an RDS

instance with Multi-AZ deployment is created for database services and an S3 bucket is created for static content storage.

Each component in the architecture is chosen with specific goals in mind:

- **VPC and Subnets**: To isolate resources and provide a secure, scalable environment.
- **Auto-Scaling and EC2**: To dynamically adjust compute resources based on demand, ensuring both scalability and cost-efficiency.
- **Load Balancer**: To distribute traffic and enhance fault tolerance, ensuring that the application remains accessible even during peak loads or failures.
- **RDS Multi-AZ**: To provide a highly available, fault-tolerant database solution that automatically fails over to a standby instance in case of a primary failure.
- **S3 Buckets**: To offload static content from the web servers, reducing load and improving performance.
- **Security Groups and IAM Roles**: To secure the environment by controlling access at multiple levels, adhering to AWS's best practices.