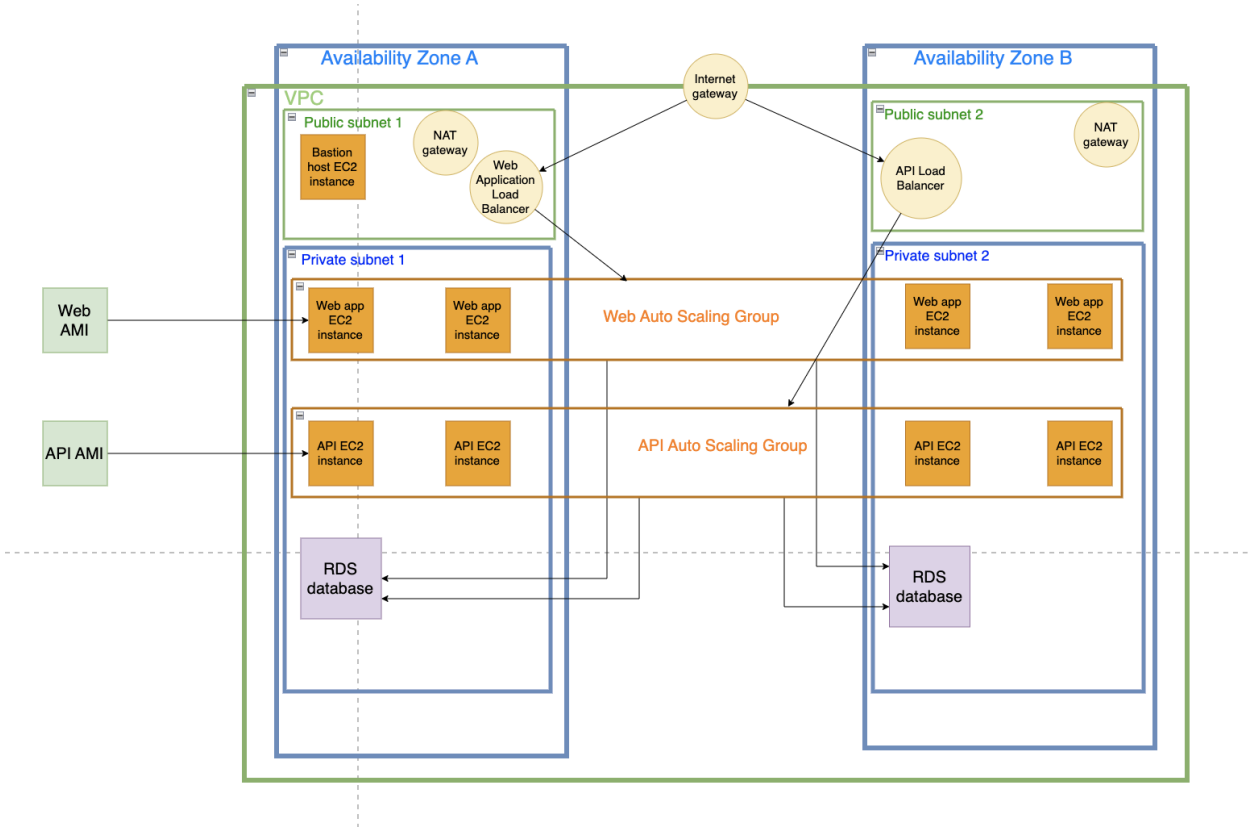
1. Solution Architecture



Web Application and API endpoint resources

a) VPC and Subnets

A Virtual Private Cloud named “project” is created to group the resources of the whole application for enhanced security and easy control. The VPC has two availability zones (AZs), and each zone has one public subnet and private subnet. The public subnets are for Application Load Balancer (ALB) and bastion host. The private subnets are for the EC2 instances hosting the web application, the API endpoints, and the database.

An Internet Gateway is also created and attached to the “project” VPC to enable internet access for resources in the public subnets.

NAT gateways are also created, one in each AZ, and they are deployed in public subnets. This is for enabling instances in private subnets to access the internet without exposing them to incoming traffic from the internet. I chose to deploy the Web and API instances in the Private Subnets, and the bastion host in Public Subnets because the Web and API contain the Web and API private resources folders, meanwhile the bastion host does

not contain any sensitive data. Therefore, I believe the private subnets can securely host the Web application and API endpoint instances.

b) EC2 instances

- Bastion Host instance
  The EC2 instance is deployed in a public subnet with t2.micro using key pair “bastion.pem” which is used for secure SSH access to the bastion host instances from the terminal.
- Web application instance
  The EC2 instance is deployed with t2.micro in private subnets with key pair “web.pem”. The instance will allow inbound and outbound rules specified in its security group named “web-security”.
- API endpoint instance
  The EC2 instance is deployed with t2.micro in private subnets with key pair “web.pem”. The instance will allow inbound and outbound rules specified in its security group named “api-security”.

Choosing t2.micro for cost-effectiveness and sufficient performance for the PHP-based web application.

c) Security groups

The bastion host inbound rules allow SSH access only from my IP address, and outbound rules to the web and API instances as we need to SSH to the web and API instances for zip files uploading.
The web application inbound rules allow HTTP on port 80 traffic from the web ALB, and allow SSH on port 22 from the bastion host. The outbound rules of the web allows the outgoing MYSQL traffic on port 3306.

The API endpoint inbound rules allow HTTP traffic on port 5000 from the API ALB, and allow SSH on port 22 from the bastion host. The outbound rules of the API also allows the outgoing MYSQL traffic on port 3306.

The web instance allows HTTP on port 80 because when the web ALB receives the incoming traffic from the internet, the web ALB is able to forward the incoming traffic to the web instances which is not possible if the web instances do not allow HTTP access.
The web allows SSH access from the bastion host as this allows the bastion host to access the web instances for zip files uploading. The

bastion host acts as a security layer before the web EC2 instances, enhancing the security for our web EC2 instances.
The web has the MYSQL outbound rule as the web instances need access to the database for the read and write functionality.

The API allows access on port 5000 HTTP because when the API ALB receives the incoming traffic on port 5000, the ALB is able to forward the incoming traffic to the API instances which is not possible if the API instances do not allow HTTP:5000.
The API allows SSH access from the bastion host as this allows the bastion host to access the API instances for zip files uploading. The bastion host acts as a security layer before the API EC2 instances, enhancing the security for our API EC2 instances.
The API has the MYSQL outbound rule as the API instances need access to the database for the read functionality.

d) AMIs

After uploading the cafe zip files to the web instance, and cafe api zip files to API instance through bastion host, modifying the config file to have the database endpoint URL from the database created on RDS, and then create database tables, we create an AMI from the web EC2 instance, and and an AMI for API instance. As these AMIs will be used in Web and API launch configurations, and because we want every web instances and API instances that are generated later by the Auto Scaling group will have the same resources as the original ones, it is important that we configure the initial instances the way we want all later web EC2, and API instances to be.

e) Load Balancer and Auto Scaling Group

The resources in this section, including the ALB, ASG, ALB security groups are built using the CloudFormation template. Please refer the template here.

The web launch configuration is deployed using the web AMI we created above. The web and API ALBs are deployed in the public subnets to distribute incoming HTTP on port 80 traffic to the web, and port 5000 traffic to the API instances across multiple AZs as required to provide high availability. The web and API ALBs are configured with health checks to

ensure traffic is only routed to healthy instances. This ensures the high availability as required, and prevents interruptions.

The web ALB security group will have the inbound rules allowing the access from the internet on port 80 HTTP. The web ALB target groups will have the web instances as its targets. The web target group listens on port 80 HTTP. The template also configures the target tracking scaling policy in which the metric type used is the CPU utilization with value of 50%. The template also configures the web listener which specifies which target groups the web ALB are listening to so that the web ALB knows which instances it can forward the traffic to.

The API launch configuration is deployed using the API AMI we created above. The most important thing to note is the user data used in this launch config. We need to make sure to include the command for running the app.py in the user data which is needed for the API endpoints to work. The API ALB security group will have the inbound rules allowing the access from the internet on port 5000. The API ALB target groups will have the API instances as its targets. The API target group listens on port 5000.
The template also configures the API listener which specifies which target groups the API ALB are listening to so that the API ALB knows which instances it can forward the traffic to.
Both the Web and API ASG are configured to maintain a minimum of two instances, each in different AZs which is to ensure the high availability. The web ASG utilizes the target tracking scaling policy based on the CPU utilization.
The ALB is chosen as it provides load distribution, preventing overload traffic to the same instance. The ASG ensures scalability and high availability by adjusting the number of instances based on load.

Database resource

The database is created within “project” VPC in the private subnets on RDS with MySQL as engine. The database has the database name of cafe_db and the password of Re:Start!9. The database is configured to be deployed in multi AZs in the given subnet group.
The database subnet group I created to be used in the database include the 2 private subnets, one in each AZ. This ensures the high availability for the database, avoiding one point of failure.

The database security group allows incoming MySQL on port 3306 traffic from web and API instances.

2. Interaction between the components

The architecture solution is designed to ensure the interaction between the web, the API endpoints and the database securely. The web application hosted on the EC2 instances in the private subnets interacts with the web ALB deployed in public subnets to reach the internet.
The web ALB is responsible for distributing incoming HTTP on port 80 requests evenly across the web application instances which ensures the traffic balancing between the instances, and prevents the traffic going into the unhealthy or irresponsive instances. Therefore, when we send requests from the internet to access the web application, the web ALB will receive the request first and then send the traffic requests to one of the web instances, and then the web instance will respond with the web application interface which is shown on the screen.
The interaction between the bastion host and the EC2 instances is also important as we need to upload the cafe zip files that contain the important files for the web interface to the web instance securely. We can SSH to the bastion host first, and then send the files from the bastion host to the web instance. Thanks to this, the web instance has the necessary files needed to display the web application interface.

The interaction with ASG is critical as it ensures the scalability and high availability for the web application. The ASG is configured to maintain a minimum of two EC2 instances across multiple AZs. It will regularly monitor the CPU utilization as configured in the launch configuration above. When the CPU utilization exceeds the target value of 50% as configured, the ASG automatically scales up by adding more additional instances to make sure the load distributed more evenly. In contrast, when the CPU utilization reduces below the target, the ASG scales downs by terminating unnecessary instances which optimizes the budget and resources, and maintaining the performance and availability.

The interaction between the web application, the API endpoints and the database is the most important. When we place an order through the web application, a sequence of interactions will happen to ensure the order is processed and can be retrieved via the API endpoints. After placing the order through the web application’s interface, the web application instance processes the order details and writes the order information to the RDS database instance that we have created. In order to retrieve the order we have just made, we can use API ALB and put :5000/orders, the queries will reach the database

and retrieve all of the orders that have been made. The retrieved data is sent back and is displayed on the screen.
The interaction flow ensures that the order data is securely processed and stored. This also makes sure the data can be efficiently retrieved via API endpoints. The integration with the RDS database ensures data being consistent and securely stored.

3. Resource provision

To implement the solution architecture, the provisioned resources needed are:

- Virtual Private Cloud (VPC) containing two AZs with four subnets (two publics and two privates), an Internet Gateway, a NAT gateway in each AZ
- Security groups.
- Auto Scaling Group for both web and API, Application Load Balancers for both web and API using launch configuration in the template and deployed through stack in CloudFormation.
- Bastion host, API, and Web EC2 instances
- API and web AMIs.

Each resource provides a specific purpose in ensuring security, scalability and availability of the solution architecture.

Provisioning resources steps

Step 1. Create key pairs and VPC. When creating a VPC in AWS, there is an option for creating the VPC and more. Choosing this option will create the VPC along with subnets, internet gateway, NAT gateways, route tables association in a few clicks. The public subnets for the ALBs and bastion host, and private subnets for the web and API EC2 instances. This setup makes sure that the application instances are not exposed to the internet, improving the security.
The internet gateway is attached to the VPC in the setup to enable internet access. NAT gateway is deployed in public subnets. This setup allows resources in private subnets to reach the internet securely.

Step 2. Create a bastion host, web and API EC2 instances. The bastion host should be in a public subnet, and both the web and API should be in private subnets. This setup enhances security for the web and API instances, as they are not exposed to external access.

Step 3. Create a database on RDS with the instance class “db.t3.micro”, and MySQL as the engine in the private subnets. The database name should be “cafe_db”, username is “admin” and password should be “Re:Start!9”. The database is configured with the database subnet group containing the two private subnets, one in each AZ.

Step 4. Setup the bastion host security group to allow SSH access from my IP addresses. The web application security group allows SSH incoming access from the bastion host, and allows the HTTP access on port 80 from the web ALB (this step can be done after we create the web ALB successfully in step 7). The web outbound rules allow outgoing MySQL connection to the database.
The API endpoints security group allows SSH incoming access from the bastion host, and allows the incoming access on port 5000 from the API ALB (this step can be done after we create the web ALB successfully in step 7). The API outbound rules allow outgoing MySQL connection to the database. These security groups act as firewalls ensuring secure connection between components.

Step 5. Uploading the cafe_app_release files to the web instance, and uploading the cafe_api_release to the API instance by SSH through the bastion host. After that, we need to unzip and change the database endpoint URL in the config files of both folders to the database endpoint URL of the database we created in step 2.

Step 6. Create an AMI from API EC2 instance and an AMI from the web instance.

Step 7. We use the CloudFormation template to deploy the web and API ALBs, ASGs, launch configurations, security groups, target groups, listeners, and target tracking scaling policy for web launch configuration.
The web launch configuration is configured with the web AMI that was created in step 6, and its user data include PHP and necessary libraries to start the web application.
The API launch configuration is configured with the API AMI created in step 6, and its user data contains commands to run the app.py which is important for the API endpoints to function when the instance is launched, along with other necessary commands.
The template deploys the ALBs in the public subnets, and ASGs in private subnets. The ASGs are to maintain two instances across different AZs. The target tracking scaling policy of CPU utilization is created in the template and is attached to the web ASG. The outputs of the template are the DNS names of the API and web ALB

CloudFormation Stack Creation

a) Prepare the CloudFormation template. The template needs to include all the resources mentioned in step 6 above.

b) We go to CloudFormation in the AWS console.
c) Create a stack by providing the template we have created in step 7. As I have specified some parameters in the beginning of the template, the stack will ask to fill in the information, for example some parameters like the bastion host, web and API security groups, the subnets IDs.
d) Submit. When the status shows create complete, we can access the DNS of the web and API ALBs in the output tab and perform the functionalities.

4. Demonstration Plan

00:00 - 03:18: Introducing VPCs, the bastion host, web and API EC2 instances, and explaining security groups of each resource. Explaining how the resources in public subnets and private subnets can be reached from the internet used internet gateway and NAT gateways. Explaining why the security contains the specific rules.

03:18 - 03:39: Showing the AMIs, and explain why AMIs should be created after uploading the necessary zip files to the instances, and not before.

03:39 - 04:44: Showing the database configuration on RDS, and its security group. Explaining why the inbound rules allow access from web and API instances.

04:44 - 06:32: Going through the CloudFormation template for web and API ALBs and ASGs resources deployment. Explaining the components needed along with the ALBs and ASGs such as target groups, listeners, etc which provide high availability. Explaining how the internet access reaches the ALBs and then the instances. Emphasizing and explaining the important command in the API launch config's user data.

6:32 - 09:37: Showing the results produced from the template in CloudFormation. Testing the functionalities of the web application and API endpoints by going to the web ALB and API ALB URLs which are in the output tab.

09:37 - 11:41: Testing the ASG by terminating a web instance to see whether another web instance will be started to replace the one that gets terminated.

APPENDIX A

```
AWSTemplateFormatVersion: '2010-09-09'
Description: CloudFormation template for deploying ALBs and ASGs for the Cafe Web
application
```

```yaml
Parameters:
 ExistingVPC:
   Type: AWS::EC2::VPC::Id
   Description: v
 BastionSecurityGroup:
   Type: AWS::EC2::SecurityGroup::Id
   Description: s
 WebAppSecurityGroup:
   Type: AWS::EC2::SecurityGroup::Id
   Description: s
 APIAppSecurityGroup:
   Type: AWS::EC2::SecurityGroup::Id
   Description: s
 PublicSubnet1:
   Type: AWS::EC2::Subnet::Id
   Description: s
 PublicSubnet2:
   Type: AWS::EC2::Subnet::Id
   Description: s
 PrivateSubnet1:
   Type: AWS::EC2::Subnet::Id
   Description: s
 PrivateSubnet2:
   Type: AWS::EC2::Subnet::Id
   Description: s
 WebAppAMI:
   Type: String
   Description: ami-024ca8d45bcd80179
 APIAppAMI:
   Type: String
   Description: ami-0358956f15876b22a
 InstanceType:
   Type: String
   Default: t2.micro
   AllowedValues: [t2.micro, t3.micro]
   Description: EC2 instance type
 KeyPairName:
   Type: AWS::EC2::KeyPair::KeyName
   Description: web

Resources:
 # Security Group for API ALB
```

```yaml
APISecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId: !Ref ExistingVPC
    GroupDescription: Enable API access on port 5000 and SSH from bastion
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 5000
        ToPort: 5000
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        SourceSecurityGroupId: !Ref BastionSecurityGroup
    Tags:
      - Key: Name
        Value: cafe-api-sg

# Security Group for web ALB
 LoadBalancerSecurityGroup:
  Type: "AWS::EC2::SecurityGroup"
  Properties:
    GroupDescription: "Security group for Load Balancer allowing HTTP from anywhere"
    VpcId: !Ref ExistingVPC
    SecurityGroupIngress:
      - IpProtocol: "tcp"
        FromPort: 80
        ToPort: 80
        CidrIp: "0.0.0.0/0" # Allow HTTP from anywhere
    SecurityGroupEgress:
      - IpProtocol: "-1"
        FromPort: 0
        ToPort: 0
        CidrIp: "0.0.0.0/0" # Allow all outbound traffic

# Launch Configuration for Web App
WebAppLaunchConfiguration:
  Type: AWS::AutoScaling::LaunchConfiguration
  Properties:
    ImageId: "ami-024ca8d45bcd80179"
    InstanceType: !Ref InstanceType
    KeyName: !Ref KeyPairName
```

```
    SecurityGroups:
      - !Ref WebAppSecurityGroup
    UserData:
      Fn::Base64: !Sub |
        #!/bin/bash
        # Install Apache Web Server and PHP
        yum install -y httpd mysql
        amazon-linux-extras install -y php7.2
        # Download and install the AWS SDK for PHP
        wget https://github.com/aws/aws-sdk-php/releases/download/3.62.3/aws.zip
        unzip aws -d /var/www/html
        # Turn on web server
        chkconfig httpd on
        service httpd start

# Auto Scaling Group for Web App
WebAppAutoScalingGroup:
  Type: AWS::AutoScaling::AutoScalingGroup
  Properties:
    VPCZoneIdentifier:
      - !Ref PrivateSubnet1
      - !Ref PrivateSubnet2
    LaunchConfigurationName: !Ref WebAppLaunchConfiguration
    MinSize: 2
    MaxSize: 6
    DesiredCapacity: 2
    TargetGroupARNs:
      - !Ref WebAppTargetGroup
    Tags:
      - Key: Name
        Value: cafe-webapp-asg
        PropagateAtLaunch: true

# Load Balancer for Web App
WebAppLoadBalancer:
  Type: AWS::ElasticLoadBalancingV2::LoadBalancer
  Properties:
    Name: webapp-loadbalancer
    Subnets:
      - !Ref PublicSubnet1
      - !Ref PublicSubnet2
    SecurityGroups:
```

```
      - !Ref LoadBalancerSecurityGroup
    Scheme: internet-facing

# Target Group for Web App
WebAppTargetGroup:
  Type: AWS::ElasticLoadBalancingV2::TargetGroup
  Properties:
    Name: webapp-targetgroup
    Port: 80
    Protocol: HTTP
    VpcId: !Ref ExistingVPC
    HealthCheckProtocol: HTTP
    HealthCheckPort: "80"
    HealthCheckPath: /cafe/index.php
    HealthCheckIntervalSeconds: 10
    HealthyThresholdCount: 2
    TargetType: instance

# Target Tracking Scaling Policy for Web App
 WebAppTargetTrackingScalingPolicy:
  Type: AWS::AutoScaling::ScalingPolicy
  Properties:
    AutoScalingGroupName: !Ref WebAppAutoScalingGroup
    PolicyType: TargetTrackingScaling
    TargetTrackingConfiguration:
      PredefinedMetricSpecification:
        PredefinedMetricType: ASGAverageCPUUtilization
      TargetValue: 50.0

# Listener for Web App Load Balancer
WebAppListener:
  Type: AWS::ElasticLoadBalancingV2::Listener
  Properties:
    DefaultActions:
      - Type: forward
        TargetGroupArn: !Ref WebAppTargetGroup
    LoadBalancerArn: !Ref WebAppLoadBalancer
    Port: 80
    Protocol: HTTP

 # Launch Configuration for API Endpoints
APILaunchConfiguration:
```

```
  Type: AWS::AutoScaling::LaunchConfiguration
  Properties:
    ImageId: "ami-0358956f15876b22a"
    InstanceType: !Ref InstanceType
    KeyName: !Ref KeyPairName
    SecurityGroups:
      - !Ref APIAppSecurityGroup
    UserData:
      Fn::Base64: !Sub |
        #!/bin/bash
        # Update and install necessary packages
        yum update -y
        yum install -y python3-pip
        pip3 install flask requests mysql-connector-python

        # Navigate to the directory where app.py is located
        cd /home/ec2-user/cafe_api_release

        # Make sure the app.py is executable
        chmod +x app.py

        # Run the Flask application
        /usr/bin/python3 /home/ec2-user/cafe_api_release/app.py > app.log 2>&1 &

# Auto Scaling Group for API Endpoints
APIAutoScalingGroup:
  Type: AWS::AutoScaling::AutoScalingGroup
  Properties:
    VPCZoneIdentifier:
      - !Ref PrivateSubnet1
      - !Ref PrivateSubnet2
    LaunchConfigurationName: !Ref APILaunchConfiguration
    MinSize: 2
    MaxSize: 2
    DesiredCapacity: 2
    TargetGroupARNs:
      - !Ref APITargetGroup
    Tags:
      - Key: Name
        Value: cafe-api-asg
        PropagateAtLaunch: true
```

```yaml
# Load Balancer for API Endpoints
APILoadBalancer:
  Type: AWS::ElasticLoadBalancingV2::LoadBalancer
  Properties:
    Name: api-loadbalancer
    Subnets:
      - !Ref PublicSubnet1
      - !Ref PublicSubnet2
    SecurityGroups:
      - !Ref APISecurityGroup
    Scheme: internet-facing
    LoadBalancerAttributes:
      - Key: idle_timeout.timeout_seconds
        Value: '60'

# Target Group for API Endpoints
APITargetGroup:
  Type: AWS::ElasticLoadBalancingV2::TargetGroup
  Properties:
    Name: api-targetgroup
    Port: 5000
    Protocol: HTTP
    VpcId: !Ref ExistingVPC
    HealthCheckProtocol: HTTP
    HealthCheckPort: "5000"
    HealthCheckPath: /products
    HealthCheckIntervalSeconds: 30
    HealthyThresholdCount: 5
    TargetType: instance

# Listener for API Load Balancer
APIListener:
  Type: AWS::ElasticLoadBalancingV2::Listener
  Properties:
    DefaultActions:
      - Type: forward
        TargetGroupArn: !Ref APITargetGroup
    LoadBalancerArn: !Ref APILoadBalancer
    Port: 5000
    Protocol: HTTP
```

```
Outputs:
 WebAppLoadBalancerDNSName:
   Description: DNS name of the web application load balancer
   Value: !GetAtt WebAppLoadBalancer.DNSName

 APILoadBalancerDNSName:
   Description: DNS name of the API load balancer
   Value: !GetAtt APILoadBalancer.DNSName
```