**Project documentation guide.**

**Define Objectives:**

* What are the primary goals of this project?
  + This is to design a three-tier web architecture in AWS.
* What problem or need is this project intended to address?
  + This is to design a new web application to enable their users access their system everywhere they are.
  + Their users to be able to make transactions online.

**Requirements Gathering:**

* What are the functional requirements of the project?
  + The system should be online for their users to access it.
  + The system should
* What are the non-functional requirements (performance, scalability, etc.)?
  + Performance: The system needs to be in high performance to enable excellent interaction with users and to achieve high throughput.
  + Reliability: All systems are designed to withstand failure. Mechanisms like replication or backups are made to accommodate this failure and to achieve a high reliability. Through this, the system can gain customer’s trust.
  + Scalability: The system must scale in or scale out to accommodate high workload by adding more resources when the need arises or demand.
  + Cost optimization: One of the objectives of an organization in business is to make profit and make service more affordable to its customers. Affordable service will be used to execute this project in other to minimise cost.
  + High Availability: Global deployment into multiple zones will be made to ensure availability of the system.
* Have you identified any constraints or limitations?

**Architecture Design:**

* What is the high-level architecture for the project?

A diagram of a computer

Description automatically generated

* Have you chosen the appropriate technology stack and platforms?
  + The appropriate technology stack is web server nginx, JavaScript, react.js, application load balancers, database aurora dB and the platform that will be use is AWS.
* How will data be structured and managed within the project?
  + SQL will be used to structure and manage the data within the project.
* What are the communication protocols and data flow diagrams?
  + The protocols are HTTP and TCP
  + Users communicate to the system through HTTP.
  + External ELB communicate to the web server through HTTP.
  + Internal ELB communicate to the appserver through HTTP.
  + Appserver communicate to the database through TCP.

**Infrastructure Setup:**

* What hardware and software infrastructure are needed?
  + The hardware’s needed are EC2.
  + The software’s are ELB, Route 53, RDS for database, nginx webserver.
* How will you provision and configure servers, databases, and other components?
  + In this project, Route 53 receives the traffic from the users and forwards the traffic to public Load Balancer, the load balancer then forwards client traffic to the web tier EC2 instances servers. The web server will be running on Nginx webservers that will be configured to serve a React.js website and redirects the API calls to the internal load balancer. The internal load balancer then forwards that traffic to the application server, which is written in Node.js. The application server manipulates data in an Aurora MySQL(database) and returns it to the web server which is then forwards to the client.
* Have you considered scalability and redundancy in your setup?
  + The system will be configured with autoscaling groups in each layer to maintain the availability and will also be deployed in multi-AZ zones.
* What is the network and security configuration?
  + The system will be deployed in VPC and will be configured with security groups rules. AWS shield will also be configured to complement the security of the network.
* Step by step deployment in the cloud.
  + Part 1: Upload of code and Creation of s3 bucket.
    - In the project, we will upload our source code to S3 so our instances can access it. We will also create an AWS Identity and Access Management EC2 role so we can use AWS Systems Manager Session Manager to connect to our instances securely and without needing to create SSH key pairs.
    - Create S3 Bucket.
      * Note: Make sure you select your own region and available zone.
    - Create IAM EC2 Instance Role
      * EC2 instance will use this role to access s3 to download the source code.
    - Upload the source code into the s3 bucket you created.
  + Part 2: Network and Security
    - We will create VPC networking components as well as security groups that will add a layer of protection around our EC2 instances, Aurora databases, and Elastic Load Balancers.
    - VPC
      * We will create a VPC for the entire project.
    - Subnets
      * We will need six subnets across two availability zones. That means that three subnets will be in one availability zone, and three subnets will be in another zone. Each subnet in one availability zone will correspond to one layer of our three-tier architecture. Create each of the 6 subnets by specifying the VPC we created above and then choose a name, availability zone, and appropriate CIDR range for each of the subnets.
        + NOTE: It may be helpful to have a naming convention that will help you remember what each subnet is for. For example, in one AZ you might have the following: Public-Web-Subnet-AZ-1, Private-App-Subnet-AZ-1, Private-DB-Subnet-AZ-1.

A screenshot of a computer

Description automatically generated

* + - * + NOTE: Remember, your CIDR range for the subnets will be subsets of your VPC CIDR range.
    - Route Tables
      * Create a Route Table for the web layer public subnets.
      * After creating the route table, you'll automatically be taken to the details page. Scroll down and click on the Routes tab and Edit routes.
      * Add a route that directs traffic from the VPC to the internet gateway. In other words, for all traffic destined for IPs outside the VPC CDIR range, add an entry that directs it to the internet gateway as a target. Save the changes.
      * Edit the Explicit Subnet Associations of the route table by navigating to the route table details again. Select Subnet Associations and click Edit subnet associations.
      * Select the two-web layer public subnets you created earlier and click Save associations.
      * Now create 2 more route tables, one for each app layer private subnet in each availability zone. These route tables will route app layer traffic destined for outside the VPC to the NAT gateway in the respective availability zone, so add the appropriate routes for that.
      * Once the route tables are created and routes added, add the appropriate subnet associations for each of the app layer private subnets.
* A screenshot of a computer

  Description automatically generated
  + - Internet Gateway
      * To give the public subnets in our VPC internet access we will have to create and attach an Internet Gateway.
        + Note: After creating the internet gateway, attach it to your VPC that you create above.
    - NAT gateway
      * For our instances in the app layer private subnet to be able to access the internet they will need to go through a NAT Gateway. For high availability, you’ll deploy one NAT gateway in each of your public subnets.
        + Note: In creating of the NAT gateway make sure you allocate an Elastic IP to it.
        + Make sure you create the NAT gateway for all the public subnets.
    - Security Groups:
      * Security groups will tighten the rules around which traffic will be allowed to our Elastic Load Balancers and EC2 instances.
      * First security group we create is for the public, internet facing load balancer.
        + Add an inbound rule to allow HTTP type traffic for your IP.
      * Second security group we create is for the public instances in the web tier.
        + Add an inbound rule that allows HTTP type traffic from your internet facing load balancer security group you created above.

Note: This will allow traffic from your public facing load balancer to hit your instances.

* + - * + Add an additional rule that will allow HTTP type traffic for your IP.

This will allow you to access your instance when we test.

* + - * Third security group will be for our internal load balancer.
        + Add an inbound rule that allows HTTP type traffic from your public instance security group.

Note: This will allow traffic from your web tier instances to hit your internal load balancer.

* + - * Fourth security group we’ll configure is for our private instances.
        + Add an inbound rule that will allow TCP type traffic on port 4000 from the internal load balancer security group you created above.

Note: This is the port our app tier application is running on and allows our internal load balancer to forward traffic on this port to our private instances.

* + - * + Add another route for port 4000 that allows your IP for testing.
      * Fifth security group we’ll configure protects our private database instances.
        + Add an inbound rule that will allow traffic from the private instance security group to the MYSQL/Aurora port (3306).

A screenshot of a computer

Description automatically generated

* + Part 3: Database Deployment
    - We will walk you through deploying the database layer of the three-tier architecture. We will create a subnet group for our database and deployed it in multi-AZ zones.
    - Subnet Groups
      * Navigate to the RDS dashboard and create a Subnet group for the database.
        + Note: When adding subnets, make sure to add the subnets we created in each availability zone specifically for our database layer. You may have to navigate back to the VPC dashboard and check to make sure you're selecting the correct subnet IDs.
    - Multi-AZ Database
      * Navigate to Databases and create database.
      * We'll now go through several configuration steps. Start with a Standard create for this MySQL-Compatible Amazon Aurora database. Leave the rest of the defaults in the Engine options as default.
      * Select Dev/Test since this isn't being used for production now. Under Settings set a username and password of your choice and note them down since we'll be using password authentication to access our database.
      * Next, under Availability and durability change the option to create an Aurora Replica or reader node in a different availability zone. Under Connectivity, set the VPC, choose the subnet group we created earlier, and select no for public access.
      * Set the security group we created for the database layer, make sure password authentication is selected as our authentication choice, and create the database.
      * When your database is provisioned, you should see a reader and writer instance in the database subnets of each availability zone.
        + Note down the writer endpoint for your database for later use.

A screenshot of a computer

Description automatically generated

* + Part 4: App Tier Instance Deployment
    - In this section we will create an EC2 instance for our app layer and make all necessary software configurations so that the app can run. The app layer consists of a Node.js application that will run on port 4000. We will also configure our database with some data and tables.
    - App Instance deployments
      * Navigate to the EC2 service dashboard create and launch an Instances.
        + Note: make sure you configure all the necessary parts, VPC, private subnet, AMI, security groups.
    - Connect to instance.
      * When the instance state is running, connect to your instance by clicking the checkmark box to the left of the instance you created, and click the connect button on the top right corner of the dashboard. Select, the Session Manager tab, and click connect. This will open a new browser tab for you.
        + NOTE: If you get a message saying that you cannot connect via session manager, then check that your instances can route to your NAT gateways and verify that you gave the necessary permissions on the IAM role for the Ec2 instance.
      * When you first connect to your instance like this, you will be logged in as ssm-user which is the default user. Switch to ec2-user by executing the following command in the browser terminal: => sudo -su ec2-user
      * Now let’s check if we can reach the internet via our NAT gateways. If your network is configured correctly up till this point, you should be able to ping the google DNS servers: => ping 8.8.8.8
        + NOTE: If you can’t reach the internet then you need to double check your route tables and subnet associations to verify if traffic is being routed to your NAT gateway!
    - Configure Database.
      * Let’s start by downloading the MySQL CLI:
        + sudo yum install mysql -y
      * Initiate your DB connection with your Aurora RDS writer endpoint. In the following command, replace the RDS writer endpoint and the username, and then execute it in the browser terminal:
        + mysql -h CHANGE-TO-YOUR-RDS-ENDPOINT (replace here with writer endpoint in part 3) -u CHANGE-TO-USER-NAME (replace with the username of the database) -p

Note: You will then be prompted to type in your password. Once you input the password and hit enter, you should now be connected to your database.

NOTE: If you cannot reach your database, check your credentials and security groups.

* + - * + Create a database called webappdb with the following command using the MySQL CLI:

CREATE DATABASE webappdb;

You can verify that it was created correctly with the following command: => SHOW DATABASES;

* + - * + Create a data table by first navigating to the database we just created: => USE webappdb;
        + Then, create the following transactions table by executing this create table command:

CREATE TABLE IF NOT EXISTS transactions (id INT NOT NULL AUTO\_INCREMENT, amount DECIMAL (10,2), description VARCHAR (100), PRIMARY KEY (id));

SHOW TABLES;

Insert data into table for use/testing later:

INSERT INTO transactions (amount,description) VALUES ('400','groceries');

Verify that your data was added by executing the following command: => SELECT \* FROM transactions;

Type exit to exits the MySQL command line.

* + - Configure App Instance
      * The first thing we will do is update our database credentials for the app tier. To do this, open the application-code/app-tier/DbConfig.js file from your local repo in your favorite text editor on your computer. You’ll see empty strings for the hostname, user, password and database. Fill this in with the credentials you configured for your database, the writer endpoint of your database as the hostname, and webappdb for the database. Save the file.
        + NOTE: This is NOT considered a best practice and is done for the simplicity of this project. Moving these credentials to a more suitable place like Secrets Manager is left as an extension for this project.
      * Upload the app-tier folder to the S3 bucket that you created in part 1.
      * Go back to your SSM session. Now we need to install all of the necessary components to run our backend application. Start by installing NVM (node version manager).
        + curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
        + source ~/.bashrc
      * Next, install a compatible version of Node.js and make sure it's being used.
        + nvm install 16
        + nvm use 16
      * PM2 is a daemon process manager that will keep our node.js app running when we exit the instance or if it is rebooted. Install that as well.
        + npm install -g pm2
      * Now we need to download our code from our s3 buckets onto our instance. In the command below, replace BUCKET\_NAME with the name of the bucket you uploaded the app-tier folder to:
        + cd ~/
        + aws s3 cp s3://BUCKET\_NAME/app-tier/ app-tier –recursive
      * Navigate to the app directory, install dependencies, and start the app with pm2.
        + cd ~/app-tier
        + npm install
        + pm2 start index.js
        + To make sure the app is running correctly run the following: => pm2 list
        + If you see a status of online, the app is running. If you see errored, then you need to do some troubleshooting. To look at the latest errors, use this command: => pm2 logs

NOTE: If you’re having issues, check your configuration file for any typos, and double check that you have followed all installation commands till now.

* + - * Right now, pm2 is just making sure our app stays running when we leave the SSM session. However, if the server is interrupted for some reason, we still want the app to start and keep running. This is also important for the AMI we will create: => pm2 startup
        + After running this you will see a message similar to this.
        + [PM2] To setup the Startup Script, copy/paste the following command: sudo env PATH=$PATH:/home/ec2-user/.nvm/versions/node/v16.0.0/bin /home/ec2-user/.nvm/versions/node/v16.0.0/lib/node\_modules/pm2/bin/pm2 startup systemd -u ec2-user —hp /home/ec2-user
        + DO NOT run the above command, rather you should copy and past the command in the output you see in your own terminal. After you run it, save the current list of node processes with the following command:

pm2 save

* + - Test App Tier
      * Now let's run a couple tests to see if our app is configured correctly and can retrieve data from the database.
      * To hit out health check endpoint, copy this command into your SSM terminal. This is our simple health check endpoint that tells us if the app is simply running.
        + curl <http://localhost:4000/health>
        + The response should looks like the following:

"This is the health check"

* + - * + Next, test your database connection. You can do that by hitting the following endpoint locally:

curl <http://localhost:4000/transaction>

* + - * + You should see a response containing the test data we added earlier:

{"result":[{"id":1,"amount":400,"description":"groceries"},{"id":2,"amount":100,"description":"class"},{"id":3,"amount":200,"description":"other groceries"},{"id":4,"amount":10,"description":"brownies"}]}

* + - * + If you see both of these responses, then your networking, security, database and app configurations are correct.
        + Congrats! Your app layer is fully configured and ready to go.
  + Part 5: Internal Load Balancing and Auto Scaling
    - In this section of the project, we will create an Amazon Machine Image (AMI) of the app tier instance we just created and use that to set up autoscaling with a load balancer in order to make this tier highly available.
    - Create an AMI of our App Tier
      * Navigate to Instances. Select the app tier instance we created and under Actions select Image and templates and Create Image.
      * Give the image a name and description and then click Create image. This will take a few minutes, but if you want to monitor the status of image creation you can see it by clicking AMIs under Images on the left-hand navigation panel of the EC2 dashboard.
    - Target group
      * We will create our target group to use with the load balancer. On the EC2 dashboard navigate to Target Groups under Load Balancing on the left-hand side. Click on Create Target Group.
      * The purpose of forming this target group is to use with our load balancer to balance the traffic across our private app tier instances.
      * Select Instances as the target type and give it a name.
      * Then, set the protocol to HTTP and the port to 4000. Remember that this is the port our Node.ja app is running on. Select the VPC we've been using thus far, and then change the health check path to be /health. This is the health check endpoint of our app. Click Next.
      * We are NOT going to register any targets for now, so just skip that step and create the target group.
    - Deploy Internal Load Balancer
      * Now let’s Create Load Balancer
      * We'll be using an Application Load Balancer for our HTTP traffic so click the create button for that option.
      * After giving the load balancer a name, be sure to select internal since this one will not be public facing, but rather it will route traffic from our web tier to the app tier.
      * Select the correct network configuration for VPC and private subnets.
      * Select the security group we created for this internal ALB. Now, this ALB will be listening for HTTP traffic on port 80. It will be forwarding the traffic to our target group that we just created, so select it from the dropdown, and create the load balancer.
    - Create a Launch Template
      * Before we configure Auto Scaling, we need to create a Launch template with the AMI we created earlier. Navigate to Launch Template under Instances and click Create Launch Template.
      * Name the Launch Template, and then under Application and OS Images include the app tier AMI you created.
      * Under Instance Type select t2.micro. For Key pair and Network Settings don't include it in the template. We don't need a key pair to access our instances and we'll be setting the network information in the autoscaling group.
      * Set the correct security group for our app tier, and then under Advanced details use the same IAM instance profile we have been using for our EC2 instances.
    - Configure Autoscaling
      * We will now create the Auto Scaling Group for our app instances. Navigate to Auto Scaling Groups under Auto Scaling and click Create Auto Scaling group.
      * Give your Auto Scaling group a name, and then select the Launch Template we just created and click next.
      * On the Choose instance launch options page set your VPC, and the private instance subnets for the app tier and continue to step 3.
      * For this next step, attach this Auto Scaling Group to the Load Balancer we just created by selecting the existing load balancer's target group from the dropdown. Then, click next.
      * For Configure group size and scaling policies, set desired, minimum, and maximum capacity to 2. Click skip to review and then Create Auto Scaling Group.
      * You should now have your internal load balancer and autoscaling group configured correctly. You should see the autoscaling group spinning up 2 new app tier instances. If you wanted to test if this is working correctly, you can delete one of your new instances manually and wait to see if a new instance is booted up to replace it.
        + NOTE: Your original app tier instance is excluded from the ASG so you will see 3 instances in the EC2 dashboard. You can delete your original instance that you used to generate the app tier AMI but it's recommended to keep it around for troubleshooting purposes
  + Part 6: Web Tier Instance Deployment
    - In this part we will deploy an EC2 instance for the web tier and make all necessary software configurations for the NGINX web server and React.js website.
    - Update NGINX Configuration Files
      * Before we create and configure the web instances, open up the application-code/nginx.conf file from the repo we downloaded. Scroll down to line 58 and replace [INTERNAL-LOADBALANCER-DNS] with your internal load balancer’s DNS entry. You can find this by navigating to your internal load balancer's details page.
      * Then, upload this file and the application-code/web-tier folder to the s3 bucket you created for this lab.
    - Create Web Tier Instance
      * Follow the same instance creation instructions we used for the App Tier instance in Part 4: App Tier Instance Deployment, with the exception of the subnet. We will be provisioning this instance in one of our public subnets. Make sure to select the correct network components, security group, and IAM role. This time, auto-assign a public ip on the Configure Instance Details page. Remember to tag the instance with a name so we can identify it more easily.
      * Then at the end, proceed without a key pair for this instance.
    - Connect to Instance
      * Follow the same steps you used to connect to the app instance and change the user to ec2-user. Test connectivity here via ping as well since this instance should have internet connectivity:
        + sudo -su ec2-user
        + ping 8.8.8.8

Note: If you don't see a transfer of packets then you'll need to verify your route tables attached to the subnet that your instance is deployed in.

* + - Configure Software Stack
      * We now need to install all of the necessary components needed to run our front-end application. Again, start by installing NVM and node :
        + curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
        + source ~/.bashrc
        + nvm install 16
        + nvm use 16
      * Now we need to download our web tier code from our s3 bucket:
        + cd ~/
        + aws s3 cp s3://BUCKET\_NAME/web-tier/ web-tier –recursive
      * Navigate to the web-layer folder and create the build folder for the react app so we can serve our code:
        + cd ~/web-tier
        + npm install
        + npm run build
      * NGINX can be used for different use cases like load balancing, content caching etc, but we will be using it as a web server that we will configure to serve our application on port 80, as well as help direct our API calls to the internal load balancer.
        + sudo amazon-linux-extras install nginx1 -y
      * We will now have to configure NGINX. Navigate to the Nginx configuration file with the following commands and list the files in the directory:
        + cd /etc/nginx
        + ls
      * You should see an nginx.conf file. We’re going to delete this file and use the one we uploaded to s3. Replace the bucket name in the command below with the one you created for this workshop:
        + sudo rm nginx.conf
        + sudo aws s3 cp s3://BUCKET\_NAME/nginx.conf .
      * Then, restart Nginx with the following command:
        + sudo service nginx restart
      * To make sure Nginx has permission to access our files execute this command:
        + chmod -R 755 /home/ec2-user
      * Make sure the service starts on boot, run this command:
        + sudo chkconfig nginx on
      * Now when you plug in the public IP of your web tier instance, you should see your website, which you can find on the Instance details page on the EC2 dashboard. If you have the database connected and working correctly, then you will also see the database working. You’ll be able to add data. Careful with the delete button, that will clear all the entries in your database.
  + Part 7: External Load Balancer and Auto Scaling
    - In this part of the project, we will create an Amazon Machine Image (AMI) of the web tier instance we just created and use that to set up autoscaling with an external facing load balancer in order to make this tier highly available.
    - Create an AMI of our Web Tier
      * Navigate to Instances of the EC2 dashboard. Select the web tier instance we created and under Actions select Image and templates. Click Create Image.
      * Give the image a name and description and then click Create image. This will take a few minutes, but if you want to monitor the status of image creation you can see it by clicking AMIs under Images on the left-hand navigation panel of the EC2 dashboard.
    - Target Group
      * We now go ahead and create our target group to use with the load balancer. On the EC2 dashboard navigate to Target Groups under Load Balancing on the left-hand side. Click on Create Target Group.
      * The purpose of forming this target group is to use with our load balancer so it may balance traffic across our public web tier instances. Select Instances as the target type and give it a name.
      * Then, set the protocol to HTTP and the port to 80. Remember this is the port NGINX is listening on. Select the VPC we've been using thus far, and then change the health check path to be /health. Click Next.
      * We are NOT going to register any targets for now, so just skip that step and create the target group.
    - Deploy External Load Balancer
      * On EC2 dashboard select Load Balancers under Load Balancing and click Create Load Balancer.
      * We'll be using an Application Load Balancer for our HTTP traffic so click the create button for that option.
      * After giving the load balancer a name, be sure to select internet facing since this one will not be public facing, but rather it will route traffic from our web tier to the app tier.
      * Select the correct network configuration for VPC and public subnets.
      * Select the security group we created for this internal ALB. Now, this ALB will be listening for HTTP traffic on port 80. It will be forwarding the traffic to our target group that we just created, so select it from the dropdown, and create the load balancer.
    - Create a Launch Template
      * Before we configure Auto Scaling, we need to create a Launch template with the AMI we created earlier. On EC2 dashboard navigate to Launch Template under Instances and click Create Launch Template.
      * Name the Launch Template, and then under Application and OS Images include the app tier AMI you created.
      * Under Instance Type select t2.micro. For Key pair and Network Settings don't include it in the template. We don't need a key pair to access our instances and we'll be setting the network information in the autoscaling group.
      * Set the correct security group for our web tier, and then under Advanced details use the same IAM instance profile we have been using for our EC2 instances.
    - Configure Auto Scaling
      * We will now create the Auto Scaling Group for our web instances. On the EC2 dashboard navigate to Auto Scaling Groups under Auto Scaling and click Create Auto Scaling group.
      * Give your Auto Scaling group a name, and then select the Launch Template we just created and click next.
      * On the Choose instance launch options page set your VPC, and the public subnets for the web tier and continue to step 3.
      * For this next step, attach this Auto Scaling Group to the Load Balancer we just created by selecting the existing web tier load balancer's target group from the dropdown. Then, click next.
      * Configure group size and scaling policies, set desired, minimum and maximum capacity to 2. Click skip to review and then Create Auto Scaling Group.
      * You should now have your external load balancer and autoscaling group configured correctly. You should see the autoscaling group spinning up 2 new web tier instances. If you wanted to test if this is working correctly, you can delete one of your new instances manually and wait to see if a new instance is booted up to replace it. To test if your entire architecture is working, navigate to your external facing load balancer, and plug in the DNS name into your browser.
      * NOTE: Again, your original web tier instance is excluded from the ASG so you will see 3 instances in the EC2 dashboard. You can delete your original instance that you used to generate the web tier AMI but it's recommended to keep it around for troubleshooting purposes.
      * Congrats! You’ve Implemented a 3 Tier Web Architecture!

**Development:**

* How will you manage the source code and version control?
  + AWS code commit will be used to manage the source codes and version control.

**Testing:dd**

* What is the testing strategy and test plan?
  + Testing will be done at each tier. From the web tier to the database tier
* Who will be responsible for testing and what types of testing will be performed?
  + Testing team will help to do the testing, or the solution architect can assist with the testing. Code review, unit testing, scalability testing, security testing, penetrating testing, and acceptance testing.

**Deployment:**

* What is the deployment plan and process?
  + The new system will be deployed on backup servers on tier-by-tier steps. From the web tier to database tier.
* How will you manage the rollout of the project to production?
  + Testing will also be done and if accepted before rollout to production.
* Are there specific configuration management steps?
* What rollback procedures are in place in case of issues?
  + Rollout to production will be in tier-by-tier basis. This can be easy to rollback when there is an issue.

**Monitoring and Optimization:**

* What monitoring tools and metrics will be used to track performance?
  + AWS cloud watch will be used to track the system performance by defining the metrics. Cloud config will also be used to tract the resources.
* How will you address performance bottlenecks and optimize the system?
  + The system will be configured will higher CPUs and memory to ensure high system performance and optimization. The system will also be configured to scale-in or scale-out according to the demand.
* Have you set up alerts for system health and incidents?
  + AWS cloud watch will be configured to trigger an alarm when the health status of a resource is down or failed to respond to a request or responds delay.
* What is the plan for continuous improvement?
  + AWS code pipeline will be used for CI/CD to improve the system.
  + AWS trust advisor will also be used to monitor the resource and ensure the best practice to use resources with security compliance in an efficient way.

**Backup and Disaster Recovery:**

* What is the backup strategy for data and system configurations?
  + Snapshots will be configured on each resource. Database replicas will also be configured to backup data in the database.
  + Muti-AZ deployments will be configured to ensure system backup.
* How will you handle disaster recovery in case of system failures?
  + Backup and restore tool will be used.

**Security and Compliance:**

* What security measures are in place to protect the system and data?
  + AWS Cloud shied to protect the system.
  + Security groups will be configured to allow only define traffics.

1. What other new AWS services have you used in this project that were unaware of prior to this project
2. What are the main challenges you have resolved to get this project working.