SIT706 Report: Deploying a Highly Available Wordpress Application

This document forms part of the assessment for SIT706 Cloud Computing

Anh Khoa Do
Faculty Of Science, Engineering And Built Environment
Deakin University
Melbourne, Australia
s223188874@deakin.edu.au

***Abstract*—This project describes the process of deploying a highly available WordPress application with Amazon Web Services (AWS). The first phase deals with deploying the application via AWS console to create VPC, AMI, Auto Scaling Group, RDS, S3, Application Load Balancer, and creating the whole architecture via Cloud Formation. In the second phase, we re-evaluate the architecture by adding or adjusting different services suitable based on the business requirements. Through this project, readers will have a good grasp on solution architecture on AWS.**

I. Introduction

WordPress is a popular open-source Content Management Platform (CMP). Although frequently used for blog management, it can also build e-commerce websites or simple social media platforms.

An initial architecture of AWS is given in the task sheet. In phase one of the task, we walk through the reimplementation of the architecture in AWS. If success, a highly scalable WordPress application that can horizontally scale via auto scaling group and utilize RDS and S3 for stateful management, deployed in a private subnet and accessed via Application Load Balancer. In phase two, we address various business requirements such as faster delivery time, disaster recovery, or security.

Amazon Web Services is suitable for their wide arrays of services that enable high availability and performance. Fully auto-managed services such as Auto Scaling Groups allow resiliency and there are various other services that support disaster on larger scales.

II. Design Diagram

AWS provides a robust toolkit for hosting WordPress and similar apps. Core services include VPC for secure networking, RDS for managed databases, EC2 for compute, S3 for storage, ALB for traffic distribution, ASG for scalability, and CloudFormation for infrastructure automation—ensuring performance, availability, and cost efficiency.

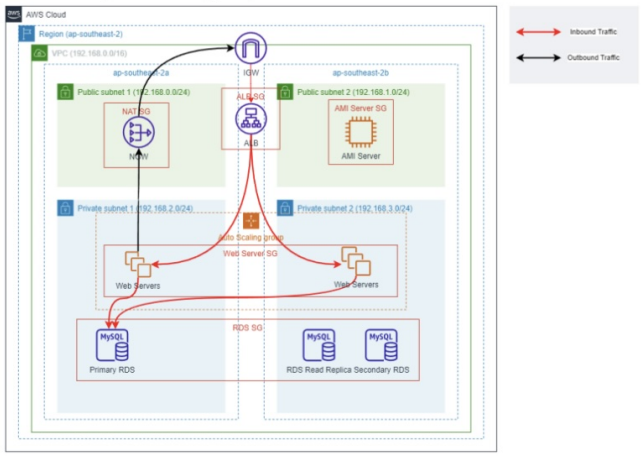

Fig. 1. Phase 1: WordPress architecture on AWS

- AWS VPC: offers secure networking environments and logical division of network.
- Public Subnets: host internet-facing resources like NAT and AMI server.
- Private Subnets: host internal resources like web servers and RDS.
- Internet Gateway (IGW): enables outbound internet access for public subnets.
- NAT Gateway: allows instances in private subnets to access the internet securely.
- Application Load Balancer (ALB): distributes traffic across web servers.
- Auto Scaling Group: manages scaling of web servers based on demand.
- Web Servers: handle application traffic and reside in private subnets.
- Security Groups (SG): control inbound/outbound traffic for resources.
- RDS Primary: stores application data with read/write access.
- RDS Read Replica: improves performance with read-only replicas.
- AMI Server: used for creating machine images in the

public subnet.

III. Implementation

A. Virtual Private Clouds

First, we navigate to the VPC console and create a new VPC. Configure the CIDR block as 192.168.0.0/16. Then we set up private and public subnets and associated them with route tables. Then we setup NAT gateway and Internet Gateway (IGW). The NAT gateway is deployed in a public subnet and be assigned an Elastic IP address

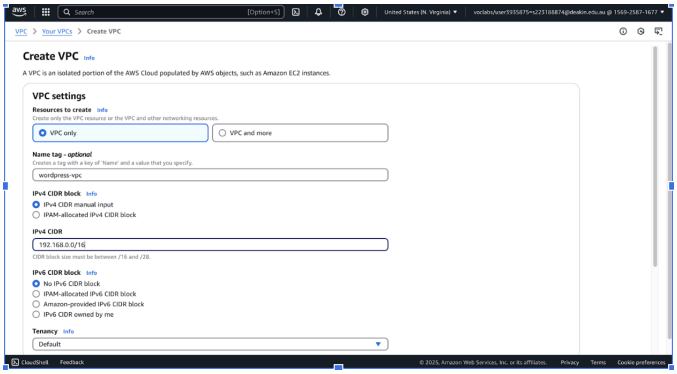

Fig. 2. Creating VPC

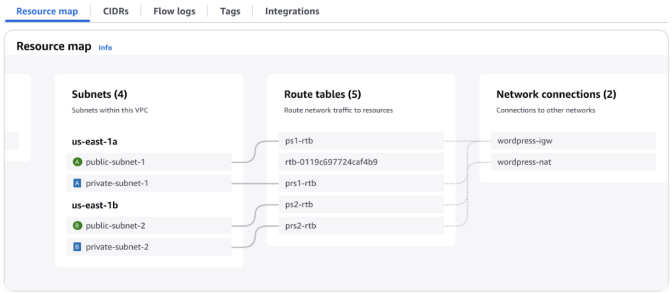

Fig. 3. AWS VPC Fully set up

B. Public and Private Subnets

We have to configure 4 subnets:

- Public Subnet 1: CIDR block 10.0.0.0/24 in Availability Zone us-east-1a.
- Public Subnet 2: CIDR block 10.0.1.0/24 in Availability Zone us-east-1b
- Private Subnet 1: CIDR block 10.0.2.0/24 in Availability Zone us-east-1a
- Private Subnet 2: CIDR block 10.0.3.0/24 in Availability Zone us-east-1b

For public subnets, the routing table should by default route traffic to the IGW.

For private subnets, the routing table should by default route traffic to NAT gateway.

C. Relational Database

Create a MySQL RDS instance with the following specifications:

- DB engine version: MySQL 8.0.35
- Template: Free tier
- Public access: Disabled
- Security Group configured to allow necessary traffic only from the Web Server

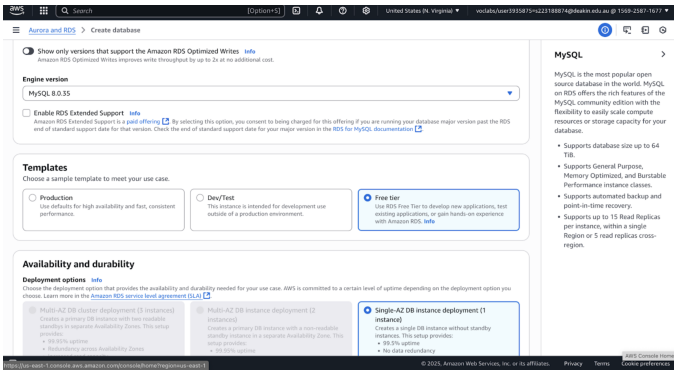

Fig. 4. Create an RDS

The Security group will accept have an inbound rule accepting MySQL/Aurora connection type at port 3306 from the security group of the web server.

Once the RDS instance is created, enable Multi-AZ from the AWS Management Console to ensure high availability.

After that, create a read replica in an Availability Zone different from the primary RDS instance to enhance fault tolerance and scalability

D. Application Load Balancer

- Next, we create an Application Load Balancer. We select the wordpress-vpc and its public subnets. Later, we will create the target group to the auto scaling group.

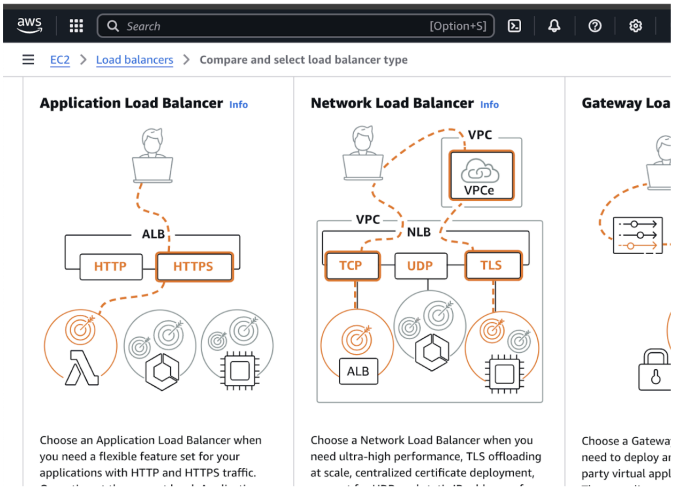

Fig. 5. Select ALB

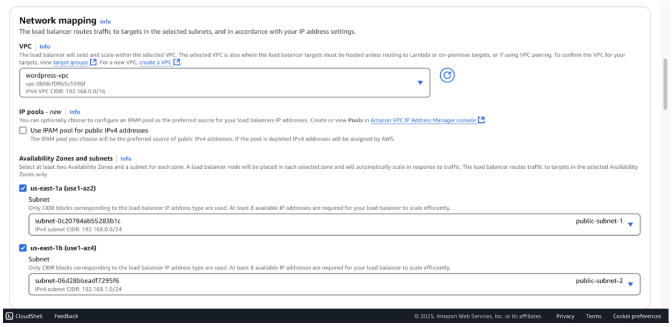

Fig. 6. Configure subnet for ALB

E. Blob Storage S3

For this project, we use S3 bucket to offload any media or static files. We will utilize a WordPress plugin that send media files to the bucket via IAM role (LabRole). The local media is deleted, leaving the actual application stateless.
The task requires public access to be off; however, this is not possible. To have media files accessible via public WordPress, we need an internet-facing URL to serve the content

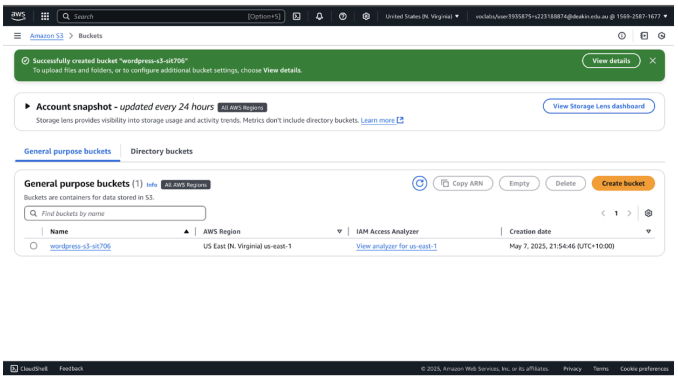

Fig. 7. Create S3 bucket

F. EC2

We will begin to set up EC2. We first install necessary package. First we initiate the EC2 instance:

- Amazon Machine Image: Amazon Linux 2 AMI (HVM), SSD Volume Type.
- Instance type: t2.micro.
- Create a new key pair for this EC2 instance.
- Configure Security Groups to allow only essential traffic types for the web server to function publicly. Remember to refer to this security group in the database's security group.
- Attach an Instance profile created in the IAM role section.
- Utilize the VPC and subnet as shown in the design diagram.
- Apply the following Advanced Details in User data.

We add the following script before creating the EC2. This code install php, httpd, mysql, and their dependencies to install WordPress. In addition, I also need php-magick installed for a related plugin.

```
#!/bin/bash
yum -y update
yum -y install php httpd mysql
PHP_VERSION=`php -v | head -n 1 | \
awk '{print $2}' | awk -F "." '{print $1}'`
while [ ${PHP_VERSION} -ne 7 ]
do
amazon-linux-extras install php7.4 -y
PHP_VERSION=`php -v | head -n 1 | \
awk '{print $2}' | awk -F "." '{print $1}'`
done
yum -y install php-mbstring php-xml
yum -y install php-magick
wget http://wordpress.org/latest.tar.gz -P \
/tmp/
tar zxvf /tmp/latest.tar.gz -C /tmp
cp -r /tmp/wordpress/* /var/www/html/
chown apache:apache -R /var/www/html
systemctl enable httpd.service
systemctl start httpd.service
```

After the EC2 is successfully created, we can open the WordPress application and start configuration.

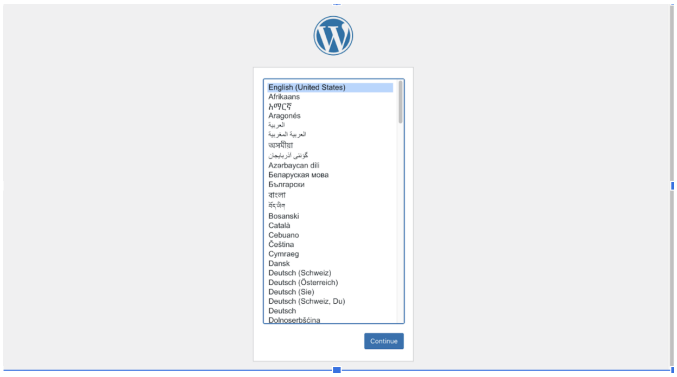

Fig. 8. WordPress starting screen

We also configure the database configurations seen in the database dashboard. Upon completion, login to the admin page.

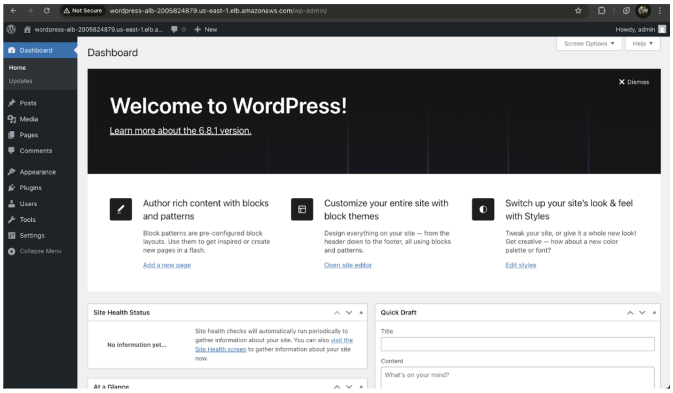

Fig. 9. WordPress admin page

In WordPress, we configure this plugin to offload S3 media. Although it is not mentioned, php-magick must also be installed via in order for this plugin to work.

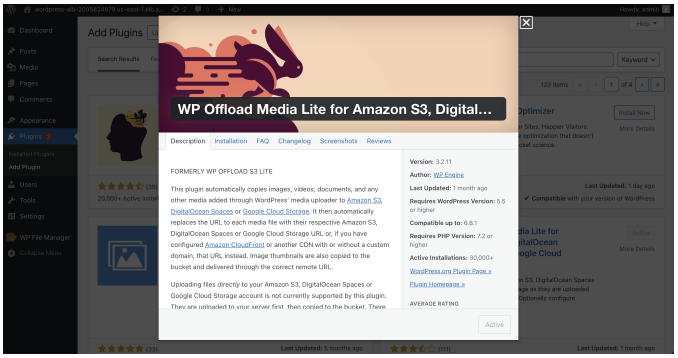

Fig. 10. S3 Offload Media Lite plugin

G. Auto Scaling Group

First we create Amazon Machine Image. This is a copy of our EC2 that can be reused for the auto scaling group.

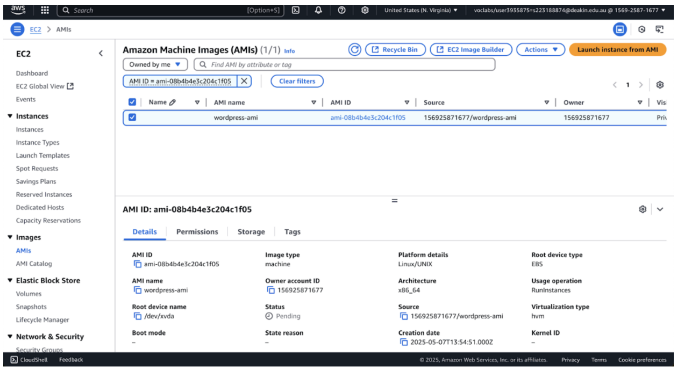

Fig. 11. AMI

Then we create an Auto Scaling Group with the following configuration

- The minimum number of servers is 1.
- The maximum number of servers is 3.
- Desired capacity of 1.
- Configure a simple scaling policy to:
  - Scale out when the average CPU utilisation of your ELB target group is above 70%.
  - Scale in when the average CPU utilisation of your ELB target group is below 25%.
- The ASG should launch instances into the private subnets.

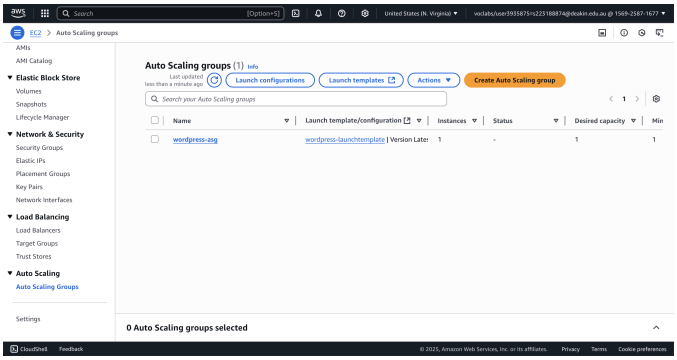

Fig. 12. Auto Scaling Group

Make sure the load balancer target is pointing to our ASG

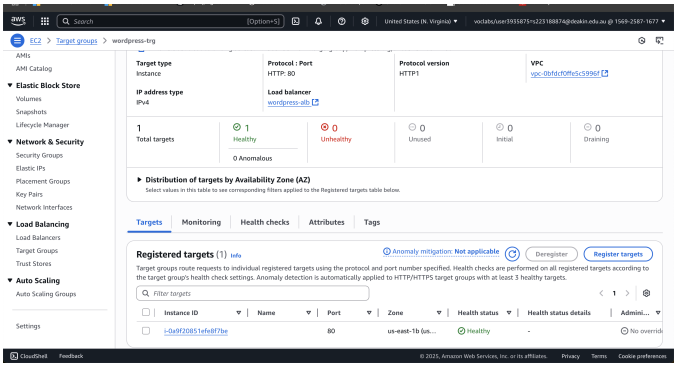

Fig. 13. Elastic Load Balancer Target

H. Cloud Formation

1) VPC: First we define the network layer of the network.

- MyVPC: Creates a VPC with DNS support and hostname resolution enabled.
- PublicSubnet1: Public subnet in AZ1 with public IP auto-assigned.
- PublicSubnet2: Public subnet in AZ2 with public IP auto-assigned.
- PrivateSubnet1: Private subnet in AZ1 without public IP mapping.
- PrivateSubnet2: Private subnet in AZ2 without public IP mapping.
- InternetGateway: Provides internet access to the VPC.
- AttachGateway: Attaches Internet Gateway to the VPC.
- PublicRouteTable: Route table for the public subnets.
- PublicRoute: Adds default route (0.0.0.0/0) to Internet Gateway.
- PrivateRouteTable1: Route table for PrivateSubnet1.
- PrivateRoute1: Adds default route to NAT Gateway for PrivateSubnet1.
- PrivateRouteTable2: Route table for PrivateSubnet2.
- PrivateRoute2: Adds default route to NAT Gateway for PrivateSubnet2.
- NatGatewayEIP: Allocates Elastic IP for NAT Gateway.
- NatGateway: NAT Gateway placed in PublicSubnet1.

- PublicSubnet1RouteTableAssociation: Links PublicSubnet1 with PublicRouteTable.
- PublicSubnet2RouteTableAssociation: Links PublicSubnet2 with PublicRouteTable.
- PrivateSubnet1RouteTableAssociation: Links PrivateSubnet1 with PrivateRouteTable1.
- PrivateSubnet2RouteTableAssociation: Links PrivateSubnet2 with PrivateRouteTable2.

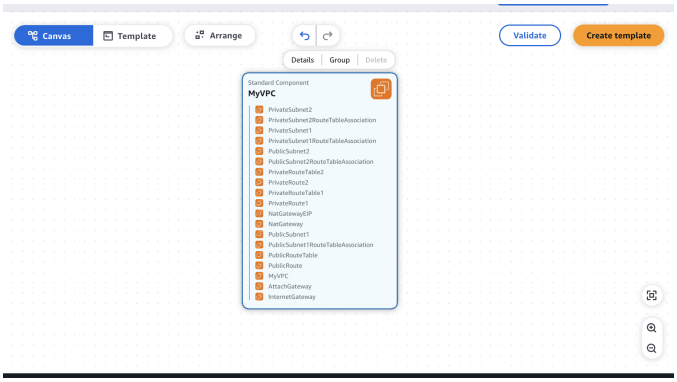

Fig. 14. CloudFormation: VPC

2) S3 and RDS: Next, we create the database layer for the project. First, we create an S3 bucket to offload media:

- BucketName: User-defined name for the S3 bucket.
- S3Bucket: Creates an S3 bucket with public access allowed and ACLs enabled.
- PublicAccessBlockConfiguration: Disables blocking of public ACLs and policies.
- OwnershipControls: Sets bucket owner as preferred for uploaded objects.
- DeletionPolicy: Retains the bucket even if the stack is deleted.
- Output BucketName: Exports the bucket name after creation.

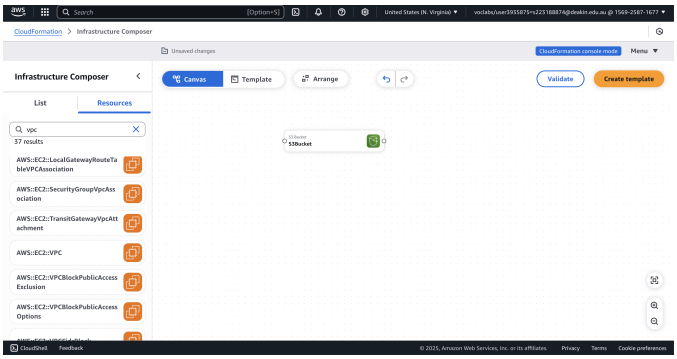

Fig. 15. CloudFormation: S3

- DBInstanceIdentifier: Sets the unique name for the RDS instance.
- DBName: Defines the initial database name inside the instance.
- DBMasterUsername: Username for the database admin.
- DBMasterPassword: Password for the admin (hidden from logs).
- VpcId: ID of the VPC where the RDS will reside.
- SubnetIds: List of private subnets for RDS placement.
- DBSecurityGroup: Security group to control RDS inbound/outbound access.
- DBSubnetGroup: Defines which subnets the RDS instance can use.
- DBInstance: Creates the actual MySQL RDS instance in the private subnet.

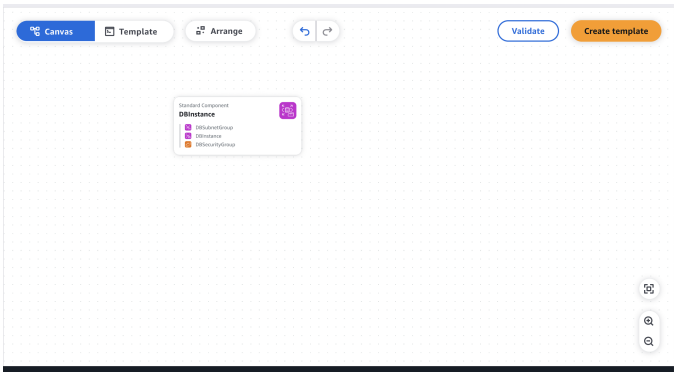

Fig. 16. CloudFormation: RDS

3) EC2: The EC2 file is as followed:

- InstanceName: Name tag for the EC2 instance.
- VpcId: Specifies which VPC the instance belongs to.
- KeyPairName: Name of the EC2 key pair for SSH access.
- SubnetId: Subnet ID where the instance will be launched.
- LatestAmiId: Uses the latest Amazon Linux 2 AMI from SSM.
- NewKeyPair: Creates an EC2 key pair with the specified name.
- EC2Instance: Launches an EC2 instance with user data to install WordPress.
- EC2SecurityGroup: Allows inbound SSH (22) and HTTP (80) traffic.
- Output KeyPair: Exports the name of the created key pair.
- Output EC2InstanceId: Exports the ID of the launched EC2 instance.

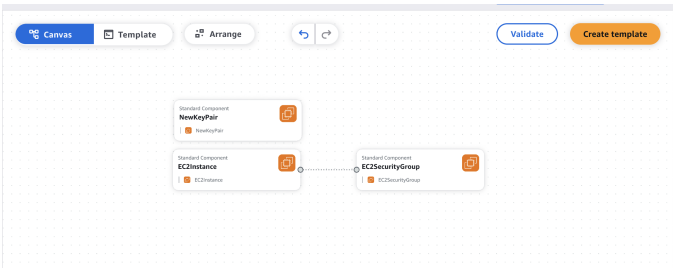

Fig. 17. CloudFormation: EC2

At this step, we need to manually setup the WordPress Application for the first time. This includes connection to

RDS, setting up offload plugin to S3. After the EC2 is set up, we create an AMI.

4) Auto Scaling Group:

- VpcId: The ID of the VPC used for all resources.
- LaunchTemplateName: Name for the EC2 launch template.
- EC2KeyName: Key pair name to SSH into EC2 instances.
- AutoScalingGroupName: Name assigned to the Auto Scaling Group.
- PrivateSubnetId1/2: Subnets for EC2 instances in the ASG.
- PublicSubnet1Id/2Id: Subnets for placing the ALB.
- WebServerLaunchTemplate: Defines instance type, AMI, and user data.
- WebServerSG: Security group for EC2 allowing HTTP/SSH access.
- WebServerAutoScalingGroup: ASG using launch template and scaling settings.
- LoadBalancerSecurityGroup: Security group allowing HTTP traffic to the ALB.
- ApplicationLoadBalancer: ALB distributing traffic across instances.
- TargetGroup: Group of instances registered to receive traffic.
- HTTPListener: Forwards traffic on port 80 to the target group.
- WebServerScaleOutPolicy: Scales out EC2 instances by 1.
- WebServerScaleInPolicy: Scales in EC2 instances by 1.
- CPUUtilizationScaleOutAlarm: Triggers scale-out if CPU more than 70%.
- CPUUtilizationScaleInAlarm: Triggers scale-in if CPU less than 25%.

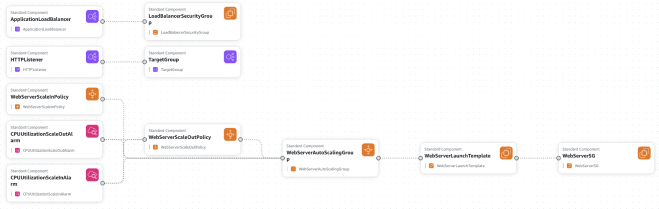

Fig. 18. CloudFormation: ASG

The code for CloudFormation can be found here: https://github.com/anhkhoado932/sit706-cloudformation

IV. Discussion and Analysis

A. Testing the application

1) Testing WordPress: First, we retrieve the ALB DNS endpoint to access the WordPress application. If not logged in, we will be redirect to the login page.

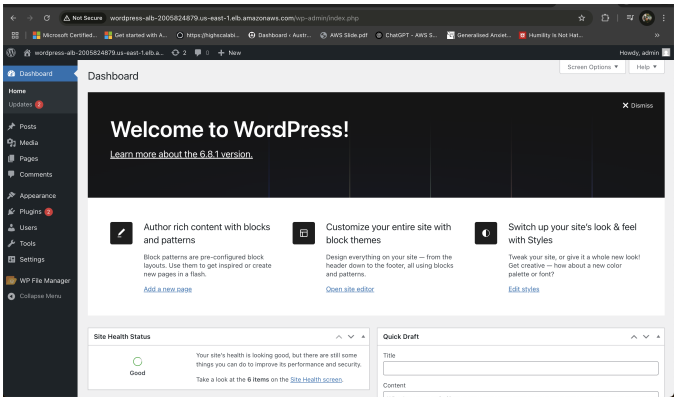

Fig. 19. Accessing WordPress with ALB

2) Testing Auto Scaling Group: If we terminate an instance, health checks by the Application Load Balancer will register the instance as unhealthy. A new instance should immediately spins up thanks to the Auto Scaling Group

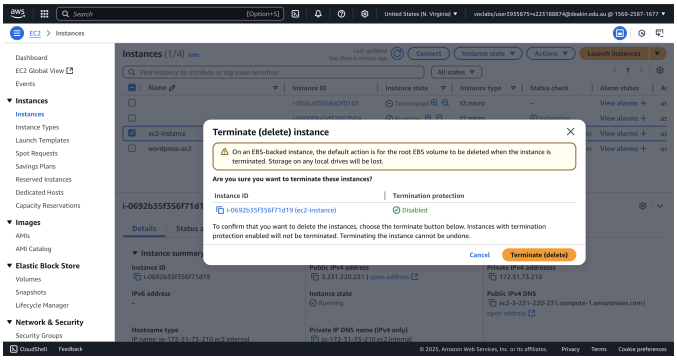

Fig. 20. Deleting

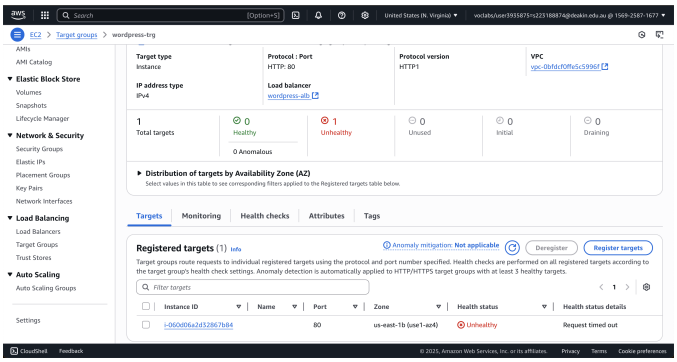

Fig. 21. The target group notice unhealthy instance

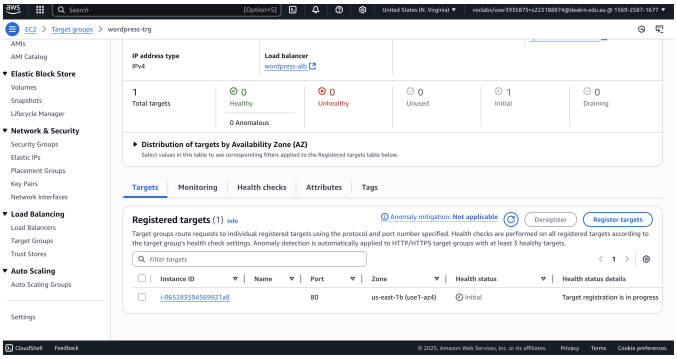

Fig. 22. A new instance is created

3) Testing Session Manager: We can log into the via session manager via communication between the instance and AWS Systems Manager services.

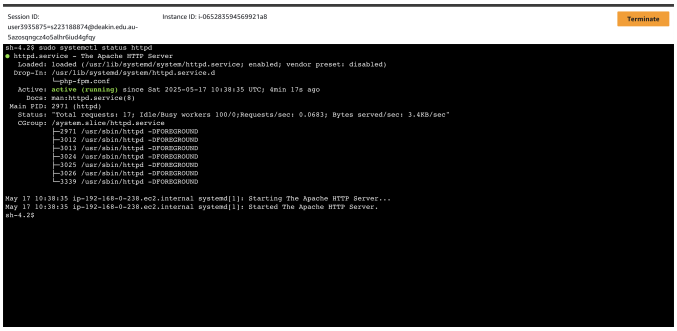

Fig. 23. Session Manager

4) Testing S3 Bucket: Uploading media in WordPress stores it in S3. Media URLs point to the S3 bucket. Files no longer exist on local EC2 storage.

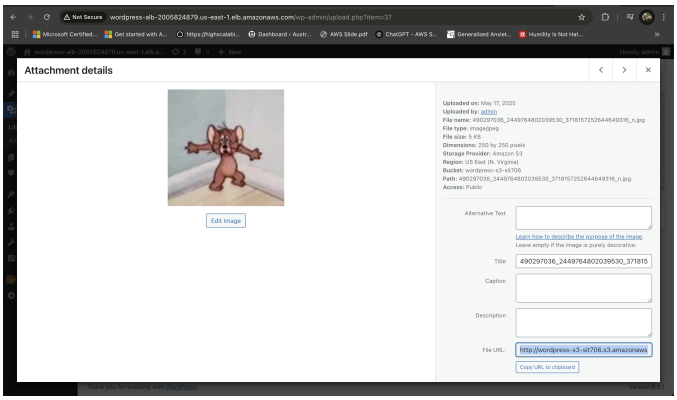

Fig. 24. Image with S3 URL indicate successful offload

5) Testing RDS: Rebooting the primary RDS triggers a failover. The standby becomes the active instance. WordPress continues functioning without downtime.

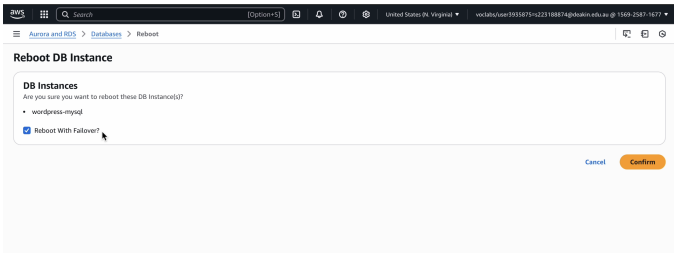

Fig. 25. Reboot with failover on the main RDS instance

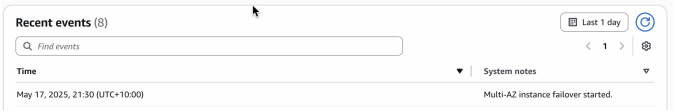

Fig. 26. Multi-AZ instance becomes failover instance

V. Business Scenario Overview

In the second phase, we redesign the architecture based on business scenarios:

- The company anticipates continuous growth in application demand, doubling annually. The architecture must accommodate this uncertain but ongoing expansion.
- The application must have the lowest possible latency. Shopping cart data must be highly available, and user session data should persist even after disconnection and reconnection.
- A serverless/event-driven architecture should be adopted wherever feasible.
- Preference should be given to managed cloud services to minimize internal system administration needs.
- The current relational database is slow and costly. Given the simple schema, more cost-efficient alternatives should be explored.
- All application data must be encrypted both in transit and at rest.
- With global expansion, response times outside Australia are slow. The architecture should improve global response times.
- The design must meet a Recovery Point Objective (RPO) of 15 minutes and a Recovery Time Objective (RTO) of 1 hour.

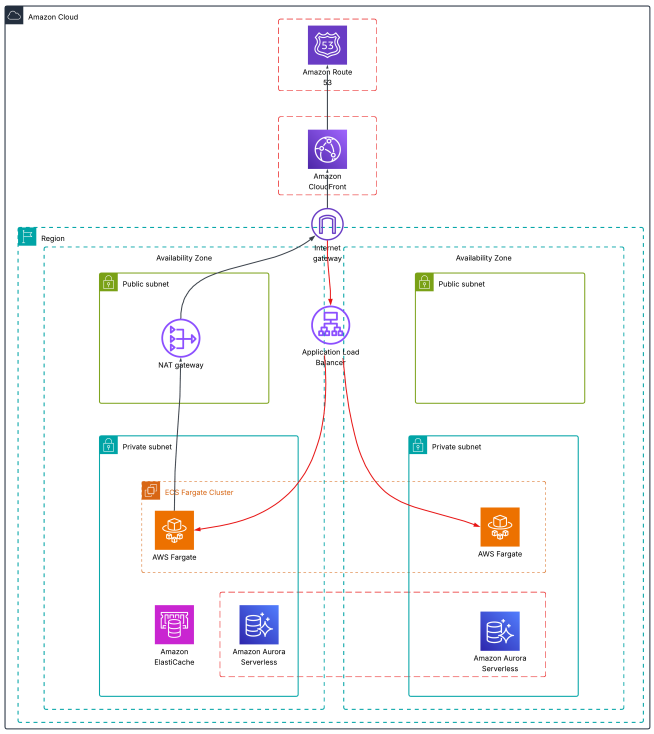

Fig. 27. Proposed Architecture

AWS Fargate is introduced to replace EC2 instances, providing a fully managed, serverless container platform. This directly support the company's goal of adopting serverless where feasible, reduces operational overhead, and allows the application to scale automatically with growing demand. It also aligns with the need for managed services and simplifies scaling for uncertain growth patterns. [1]

Amazon Aurora Serverless is chosen over the previous RDS setup due to its ability to scale automatically based on load while offering high availability and fault tolerance. Its support for fast failover and automatic backups ensure the architecture meets the RPO of 15 minutes and RTO of 1 hour. [2]

Amazon ElastiCache is used to store session and shopping cart data in-memory, ensuring extremely low latency and high availability. This directly addresses the requirement for persistent user sessions and shopping carts, even after a disconnection, and contribute to an overall faster user experience. [3]

Amazon CloudFront is added to distribute content globally, minimizing latency for users outside Australia. It caches static content closer to users around the world, significantly improving global response times and aligning with the expansion goals but doesn't need any change in the origin. [4]

Amazon Route 53 supports latency-based routing and global DNS management. It ensure that users are directed to the closest and healthiest endpoints, reducing response times globally and contributing to application availability and resilience. [5] Table I summarize all of the proposed change, which business requirement the address, and whether they are implemented. If a service is not implemented, a detail plan on how the configuration should be is proposed.

TABLE I
PROPOSED SERVICES AND MAPPED REQUIREMENTS

Service Proposed	Business Requirements Addressed	Status
AWS Fargate	Req 1: Scalability Req 3: Serverless Req 4: Managed Services	**Implemented**
Amazon Aurora Serverless	Req 1: Growth Req 4: Managed Req 8: RPO/RTO	**Planned**
Amazon Elasti-Cache	Req 2: Low latency	**Implemented**
Amazon CloudFront	Req 2: Low latency Req 7: Global response time	**Planned**
Amazon Route 53	Req 2: Low latency Req 7: Global routing, Req 8: Availability	**Planned**
Disaster Recovery plan	Req 8: Disaster Recovery	**Planned**
Encryption at-rest and in-transit	Req 6: Encryption	**Implemented**

A. AWS ElastiCache

Before serving the website to end user, WordPress queries the database to retrieve blog contents. This step can be made faster by utilizing AWS ElastiCache. Essentially, caching will be carried out if the object is usually queried. Figure 28 explained how caching works. We create a Memcache server as follow:

- Select Design your own cache
- Select Demo tier
- Customize Default settings
- Select our VPC and private subnets
- In the Memcache security group, add inbound **Custom TCP rule** targeting our EC2 Auto Scaling Group at port 11211.

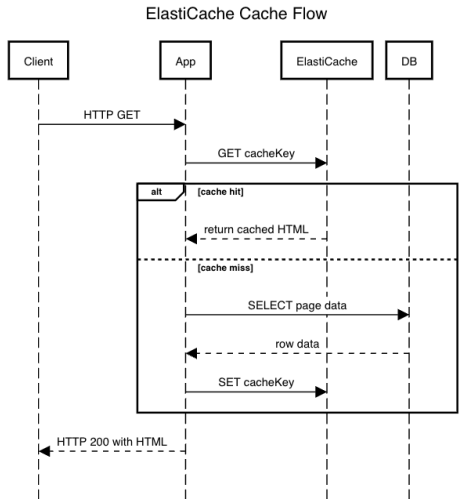

Fig. 28. Sequence Diagram of ElastiCache

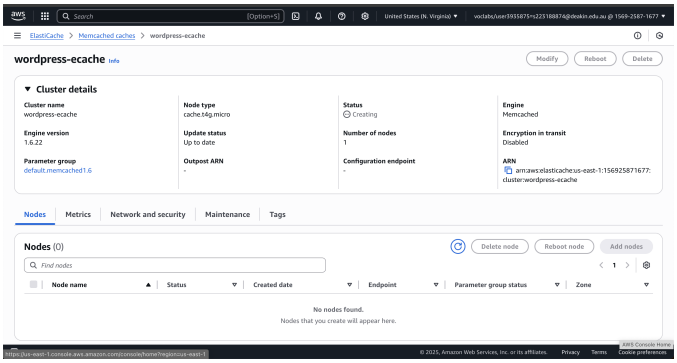

Fig. 29. Creating AWS ElastiCache

Fig. 30. Setting security Group for AWS ElastiCache

In WordPress, we use S3 Total Cache plugin, which automatically cache frequently used query objects and is compatible with memcache. Simply install the plugin, go to **W3 Total Cache**, **Database Caching**, then enable caching and add the Memcache endpoint.

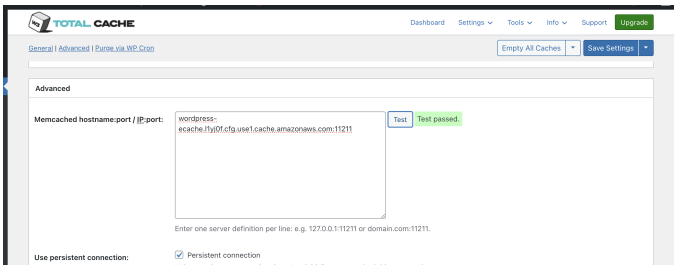

Fig. 31. Connect WordPress to ElastiCache

B. AWS Fargate

AWS Fargate is a serverless compute engine for containers that lets us run Docker workloads without provisioning or managing servers. It can automatically scale, patch, and secure the underlying infrastructure.
First, we create the ECS cluster. Our underlying should be Fargate (Serverless).

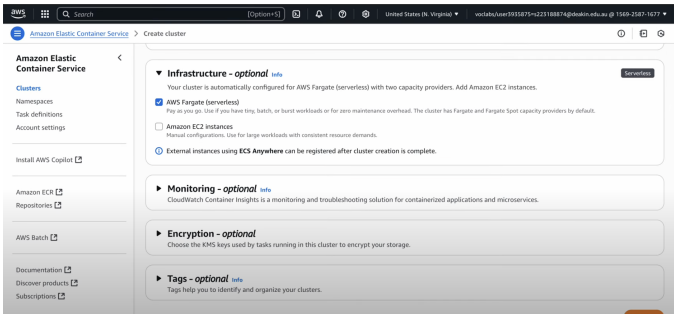

Fig. 32. Creating ECS Cluster

Next, we setup a task definition, defining which Docker image to use, port, and environment variable. Here, I opt for WordPress official image to simplify the Docker building process

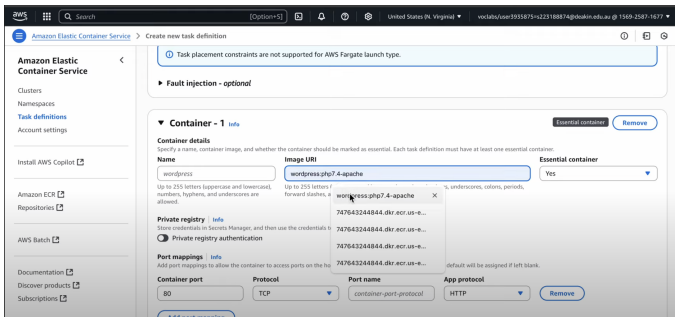

Fig. 33. Creating Task Definition

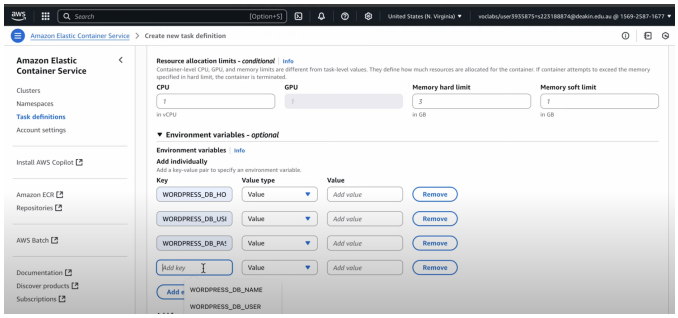

Fig. 34. Defining Environment in Task Definition

After the task definition is created, we deploy this task definition. Here, we create a new target group to point the ALB to our ECS cluster.

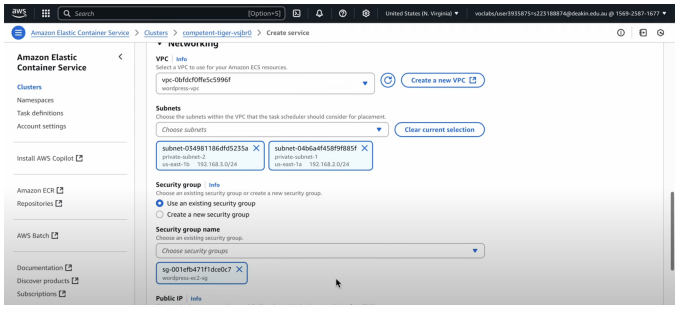

Fig. 35. Deploying Task Definition

After everything went successful, the ALB will be redirected to the ECS cluster

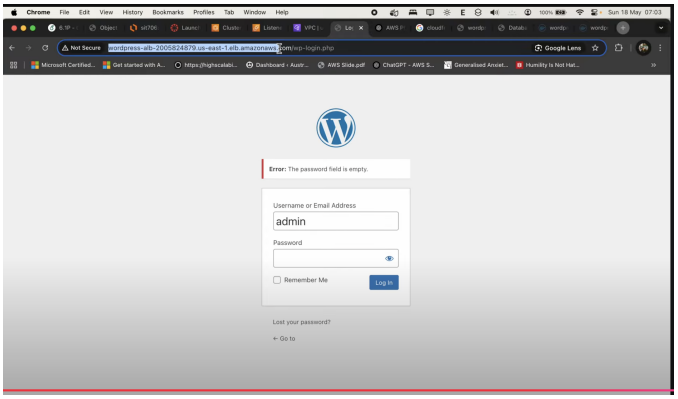

Fig. 36. ECS is accessible via ALB

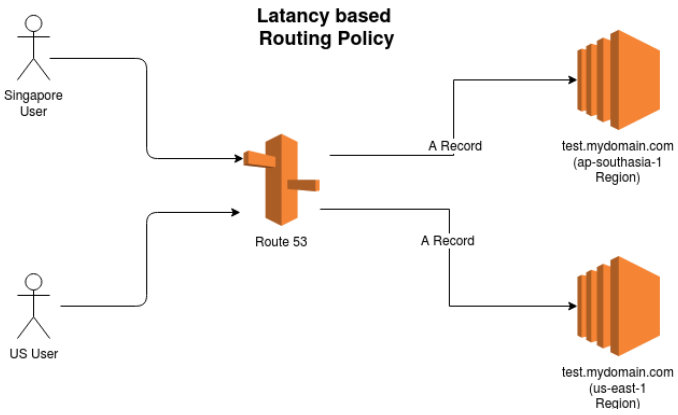

Fig. 37. Route 53 with latency-based routing [6]

C. AWS Aurora Serverless

Amazon Aurora Serverless is chosen over the previous RDS setup due to its ability to scale automatically based on load while offering high availability and fault tolerance. Its support for fast failover and automatic backups ensure the architecture meets the RPO of 15 minutes and RTO of 1 hour. It is also more cost-effective for simple schemas, resolving the issues with the legacy relational database.

- In the RDS console, select **Create database** → **Amazon Aurora** → **MySQL-compatible** → **Serverless (v2)**.
- Configure **Capacity settings**: set minimum and maximum ACUs to enable automatic scaling.
- Choose the appropriate **VPC**, **subnet group**, and **security group** for network placement.
- Enable **encryption at rest** (AWS KMS) and enforce **SSL/TLS** connections.
- Set **backup retention**, enable **Multi-AZ** for high availability, and optionally configure **Global Database** for multi-region reads.
- Provide administrator credentials or integrate with **AWS Secrets Manager** for secure credential management.

D. AWS Route 53

The decision to use Route 53 serve two purposes. First, we take advantage of its latency-based routing policy, which redirect users' requests to the fastest response endpoint. In international settings (outside of Australia), this policy works especially well to guarantee quick response rate. Additionally, route 53 is also a domain registra. Having a domain allow us to request for a CA SSL certificate in AWS through AWS Certificate Manager (ACM). We can then use this certificate for HTTPS encryption, which at an extra layer of security in-transit for outbounding connections towards end users.

E. AWS CloudFront

CloudFront is AWS's global Content Delivery Network that caches both static and dynamic assets at edge locations worldwide, dramatically reducing latency and offloading origin servers. It can be integrated with S3, ALB, API Gateway, ensuring fast, reliable, and secure delivery of your website content to users everywhere. This service cannot be demonstrated due to student account restriction, but the summary of how it should be setup is demonstrated below

- Create a CloudFront service in the AWS console or via IaC.
- Specify origin as S3 bucket for media
- Define cache behaviors
- Update Route 53 DNS to alias your domain to the CloudFront distribution.

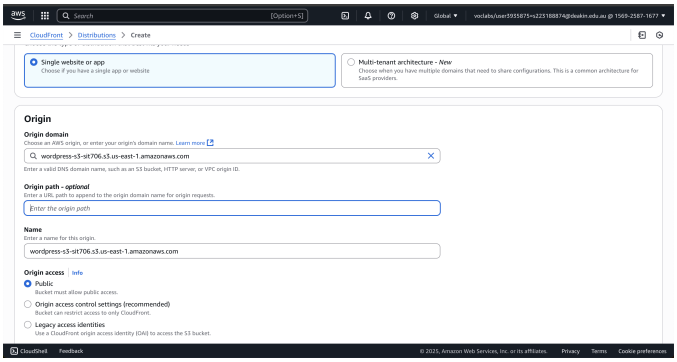

Fig. 38. AWS CloudFormation creation interface

F. Securing data at-rest and on-transit

Encrypting data both in transit and at rest protects sensitive information from being accessed ot intercepted, such at man-in-the-middle attacks. In production, compliance to data standards such as GDPR is crucial to build customer trust by preserving confidentiality, and mitigates risks.
Investigating the AWS Services, we note that at rest, AWS RDS by default enable encryption.

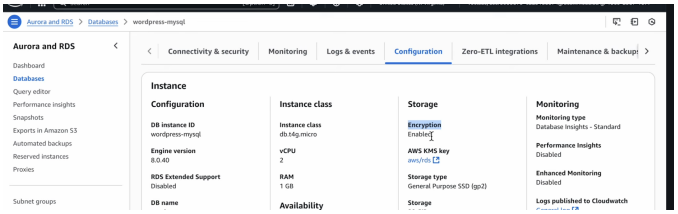

Fig. 39. AWS RDS is encrypted at-rest

AWS RDS also has self-signed certificate for SSL-enhanced data transmission. We can download the key bundle from the AWS document pages

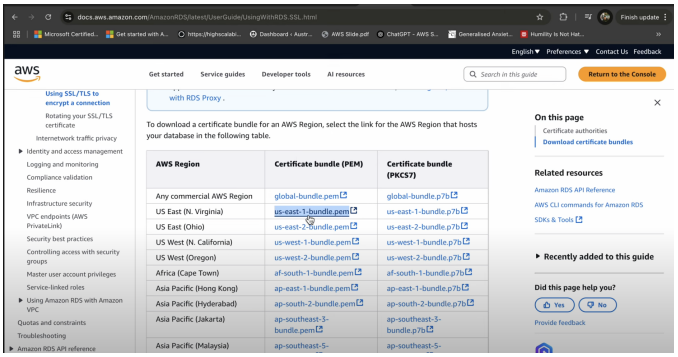

Fig. 40. AWS provides certificate bundle key for the self-signed RDS

Now, we login to the EC2 via session manager. We can upload the .pem file and enable encryption in WordPress by adding to wp-config.php. Detailed code is listed in figure 41

```
 * * ABSPATH
 *
 * @link https://developer.wordpress.org/advanced-administration/wordpress/wp-config/
 *
 * @package WordPress
 */

// ** Database settings - You can get this info from your web host ** //
/** The name of the database for WordPress */
define( 'DB_NAME', 'wordpress' );

/** Database username */
define( 'DB_USER', 'admin' );

/** Database password */
define( 'DB_PASSWORD', 'password' );

/** Database hostname */
define( 'DB_HOST', 'wordpress-mysql.c1b1mvjntpiu.us-east-1.rds.amazonaws.com' );

/** Database charset to use in creating database tables. */
define( 'DB_CHARSET', 'utf8mb4' );

/** The database collate type. Don't change this if in doubt. */
define( 'DB_COLLATE', '' );

define('MYSQL_SSL_CA', '/etc/ssl/certs/us-east-1-bundle.pem');
define( 'MYSQL_CLIENT_FLAGS', MYSQLI_CLIENT_SSL |
MYSQLI_CLIENT_SSL_DONT_VERIFY_SERVER_CERT );
define( 'MYSQL_SSL_CA', getenv( 'MYSQL_SSL_CA' ));
```

Fig. 41. Enable database encryption for WordPress

After that, we run the command sudo systemctl restart httpd. Now our application should be connecting to RDS via HTTPS.

Next we investigate the data transmission from our application to end users. AWS Certificate Manager (ACM) is a fully managed service for provisioning, validating, and renewing TLS/SSL certificates. We can attach ACM certificates to edge services like CloudFront or load balancers, it enforces HTTPS and encrypts all data in transit. Due to student account restriction, this part is not implemented. In the ACM console, we could implement the SSL certificate as follow:

- request a public certificate for our domain
- complete DNS or email validation
- attach the issued certificate to CloudFront.

G. Disaster Recovery Plan

Disaster Recovery (DR) refers to the resiliency of the system in unfortunate situation. Any highly available system must be ready for uncertain future, no matter how unlikely it is. Here, we are tasked with building a system with Recovery Point Objective (RPO) of 15 minutes and Recovery Time Objective (RTO) of 1 hour. To draw a decision, we refer to AWS student guide on selecting DR implementation pattern.

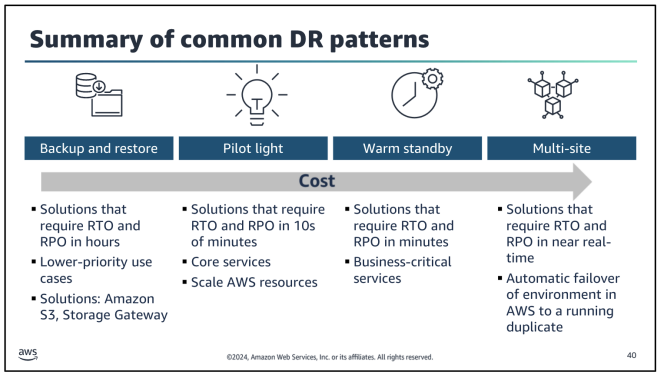

Fig. 42. Disaster Recovery Student Guide [7]

The selected method is **warm standby**, a pattern that run a scaled-down but fully functional version of the system. In a disaster, Route 53, upon failed health checks with the primary system, redirects traffic to this environment, which then scales horizontally to handle production load.

Although we did not implement this due to student account limitation, running a minified version of our main system at all time can be automated thanks to AWS CloudFormation we used in phase 1. We can use CI/CD pipeline to automate the process of infrastructure change. Figure 43 describe the architecture where a warm stand-by deployed in case of disaster.

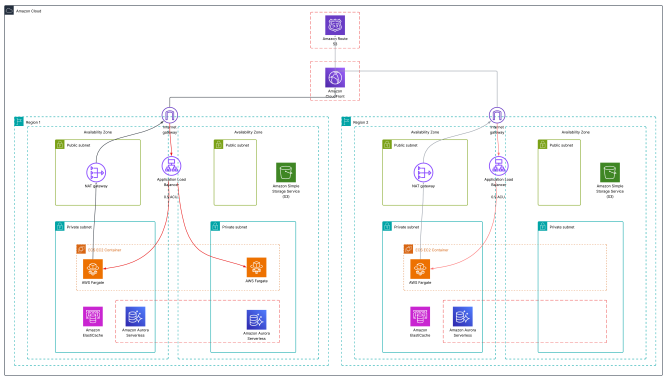

Fig. 43. A standby environment (right) is deployed in another region and will have incoming traffic in case of disaster

VI. Conclusion

This project demonstrate the deployment and improvement of a highly available WordPress application on AWS. In Phase 1, we built a basic architecture using EC2, RDS, ALB, and CloudFormation to deploy a scalable and highly-available system. In Phase 2, we improved the architecture to match business requirements. We used AWS Fargate and Aurora Serverless to improve performance and save cost. ElastiCache helped store WordPress quickly and made the user experience faster. CloudFront helped speed up delivery for static contents and route 53 made global DNS utilize a latency-based routing policy.

References

[1] R. Alvarez-Parmar and J. Hayes, "Running wordpress on amazon ecs on aws fargate with amazon efs," May 2021, accessed: 2025-05-18. [Online]. Available: https://aws.amazon.com/blogs/containers/running-wordpress-amazon-ecs-fargate-ecs/

[2] Amazon Web Services, "Using aurora serverless v2 - amazon aurora," 2024, accessed: 2025-05-18. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.html

[3] ——, "Speeding up wordpress with amazon elasticache for memcached," 2021, accessed: 2025-05-18. [Online]. Available: https://aws.amazon.com/elasticache/memcached/wordpress-with-memcached/

[4] R. Guilfoyle, "How to accelerate your wordpress site with amazon cloudfront," November 2017, accessed: 2025-05-18. [Online]. Available: https://aws.amazon.com/blogs/startups/how-to-accelerate-your-wordpress-site-with-amazon-cloudfront/

[5] Amazon Web Services, "Latency-based routing - amazon route 53," 2024, accessed: 2025-05-18. [Online]. Available: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy-latency.html

[6] C. Nanayakkara, "Aws route 53 and routing policies," April 2021, accessed: 2025-05-18. [Online]. Available: https://crishantha.medium.com/aws-route-53-and-routing-policies-b7dc67e74516

[7] AWS Academy, "Module 16: Planning for disaster – student guide," *AWS Academy Cloud Architecting [104153]*, 2025, accessed: 2025-05-18.

Github: https://github.com/anhkhoado932/sit706-cloudformation

Youtube: https://youtu.be/2hIcvEsXQVg