Harvey Aland
March 9, 2025

AWS MULTI-REGION DR WITH TERRAFORM: BUILDING A RESILIENT INFRASTRUCTURE

Why This Architecture?

From Certification to Real-World Implementation

AWS certifications (CCP, SAA) provided the theory, this project proves I can:

✅ **Build production-ready DR:** Not just conceptual but tested with simulated outages.

✅ **Automate recovery:** Reduced manual intervention from hours to <4 minutes.

✅ **Balance cost & resilience:** Passive region scales up *only* during failures (~40% savings vs. active/active).

Business Impact:

- Minimized downtime: Critical for revenue-sensitive applications.
- Future-proof: Modular Terraform code adapts to new regions/services.

Showcasing Core AWS Services

- The project allowed me to work with key AWS services, including:
    - **Route 53**: For DNS failover.
    - **Application Load Balancer (ALB)**: For distributing traffic.
    - **RDS**: For a highly available database.
    - **Auto Scaling Groups (ASG)**: For dynamic scaling of compute resources.
    - **AWS Backup**: For automated database backups.
    - **IAM**: For managing permissions and roles.
    - **CloudWatch**: For monitoring, logging, and triggering alarms for failover events.
    - **SNS**: For sending failover notifications and triggering the Lambda function.
    - **Lambda**: For automating failover actions, such as scaling up the failover ASG.
    - **AWS KMS**: For encrypting the RDS Database

Showcasing Terraform Expertise

- Everything was deployed using **Infrastructure-as-Code (IaC)**, ensuring a fully automated and reproducible setup. This demonstrates proficiency in Terraform, a critical tool for modern cloud engineering

Resume Enhancement

- This challenge aligns well with the skills required for cloud engineering roles. It demonstrates proficiency in **AWS infrastructure design**, **disaster recovery planning**, and **Terraform automation**.

Key Objectives

1. **High Availability**:
    - Ensure the application remains operational even if one region experiences an outage.
2. **Disaster Recovery**:
    - Automatically fail over to a secondary region in the event of a regional failure.
3. **Scalability**:
    - Dynamically scale resources based on demand.
4. **Cost Optimization**:
    - Minimize costs by keeping failover resources scaled down (passive) when not in use.

Overview of the Architecture

This challenge involved building a highly available, multi-region **Active-Passive** disaster recovery architecture in AWS using Terraform. The goal was to ensure that both the compute layer (EC2 & ALB) and the database layer (RDS) could automatically fail over in case of failure, minimizing downtime. In this setup, the **primary region actively handles traffic**, while the **failover region remains idle** until an outage occurs. This approach optimizes costs while ensuring redundancy. The architecture is built to handle regional failures gracefully, ensuring minimal downtime and seamless failover.

What is Active Passive?

Active-Passive is a disaster recovery setup where one region (the active region) handles all traffic, while another region (the passive region) remains idle. If the active region fails, the passive region is activated to take over, ensuring minimal downtime. This approach saves costs by keeping resources in the passive region inactive until needed, while still allowing for fast recovery.

Active & Passive regions

1. Primary Region (us-east-1) (Active/Primary)

The primary region is the main deployment environment where the application runs under normal conditions. Key components include:

- **VPC**:
    - A Virtual Private Cloud (Primary_vpc) with CIDR block 10.0.0.0/16 provides isolated networking.
    - Public and private subnets are created across multiple Availability Zones (AZs) for redundancy.
- **EC2 Instances**:
    - Managed by an Auto Scaling Group (Prim_ASG), these instances run the application.
    - Launched in the ASG by the pre-defined EC2 launch template
    - The ASG ensures that the desired number of instances is always available.
- **Application Load Balancer (ALB)**:
    - Distributes incoming traffic across EC2 instances in the target group.
    - Monitors the health of instances and routes traffic only to healthy targets.
- **RDS Database**:
    - A Multi-AZ MySQL RDS instance (Primrds) ensures high availability for the database layer.
    - Automatic backups are enabled to protect against data loss.
- **Route 53**:
    - DNS routing directs traffic to the primary ALB under normal conditions.
    - Health checks monitor the primary ALB and trigger failover if necessary.

2. Failover Region (us-west-2) (Passive/Failover)

The failover region acts as a standby environment that becomes active only during a failure in the primary region. Key components include:

- **VPC**:
    - A Virtual Private Cloud (Failover_vpc) with CIDR block 10.1.0.0/16 provides isolated networking.
    - Public subnets are created across multiple AZs for redundancy.
- **EC2 Instances**:
    - Managed by an Auto Scaling Group (Fail_ASG), these instances remain scaled down to save costs.
    - The ASG scales up only during a failover event.
- **Application Load Balancer (ALB)**:
    - Distributes incoming traffic across EC2 instances in the target group.
    - Takes over traffic routing when the primary ALB becomes unhealthy.
- **Route 53**:
    - DNS routing directs traffic to the failover ALB during a primary region outage.
    - Ensures seamless failover with minimal user disruption.

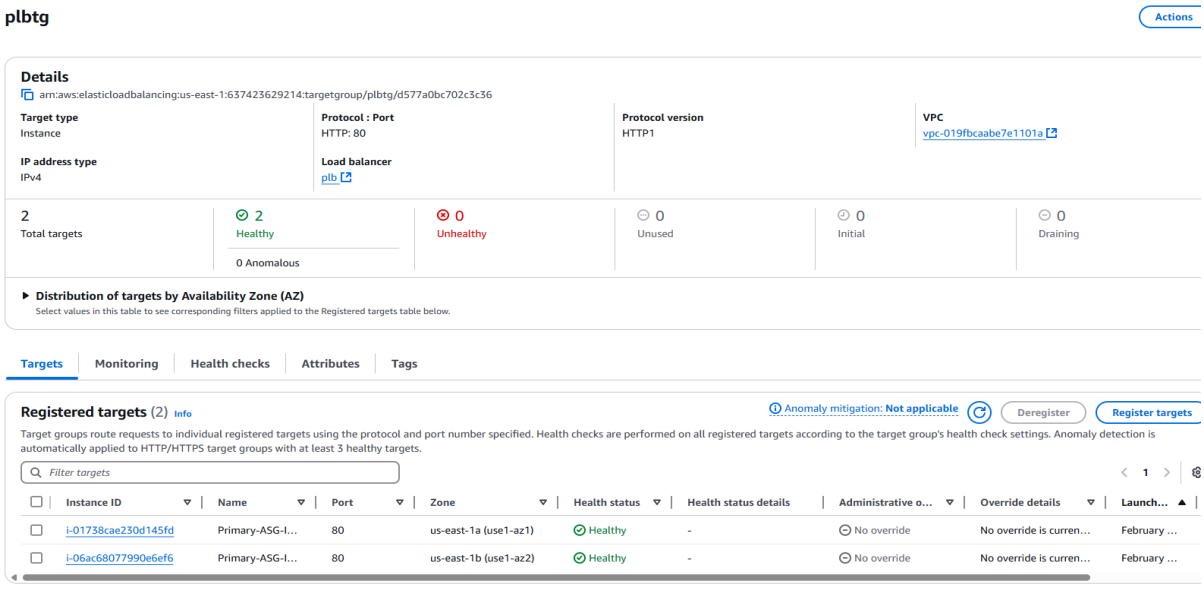
Failover Process Breakdown:

Normal Operation

- **Traffic Flow**:
    - User traffic is routed to the **Primary ALB** in the **Primary Region (us-east-1)** via **Route 53**.
- **Compute Layer**:
    - The **Primary ASG (Prim_ASG)** maintains the desired number of EC2 instances to handle traffic.
- **Database Layer**:
    - The **Primary RDS** instance (Primrds) serves as the database layer, with automatic backups enabled.
- **Monitoring**:
    - **CloudWatch** continuously monitors the health of the Primary ALB and RDS instance.

Failover In Progress

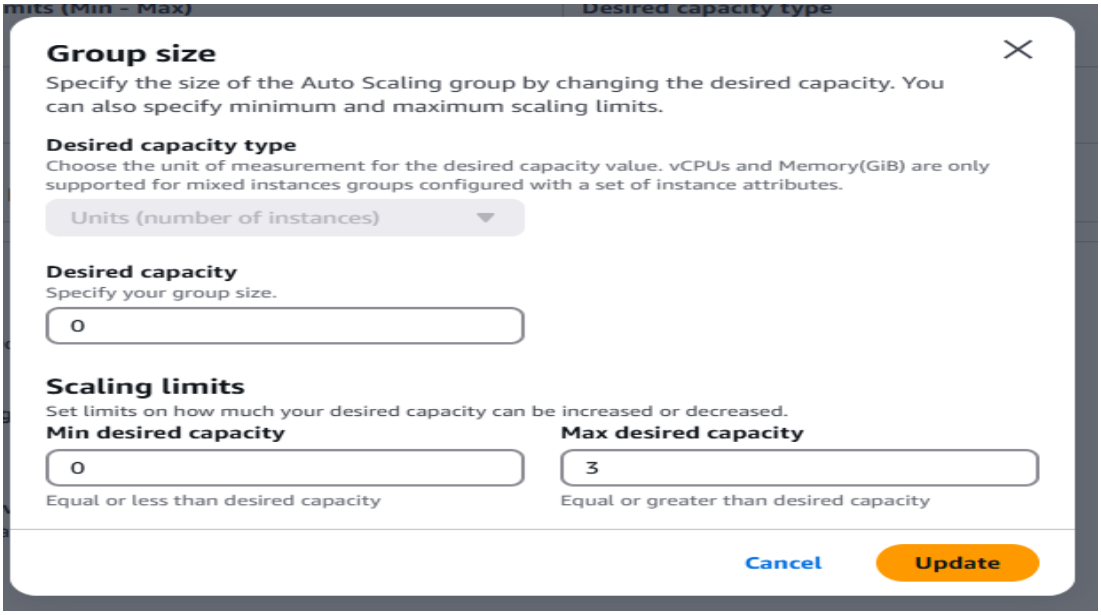
If the primary ALB target group checks fail, a CloudWatch alarm triggers. Here, the ALB TG for the primary region has two healthy instances running as normal.



ASG Capacity set to zero (Staged Failover)

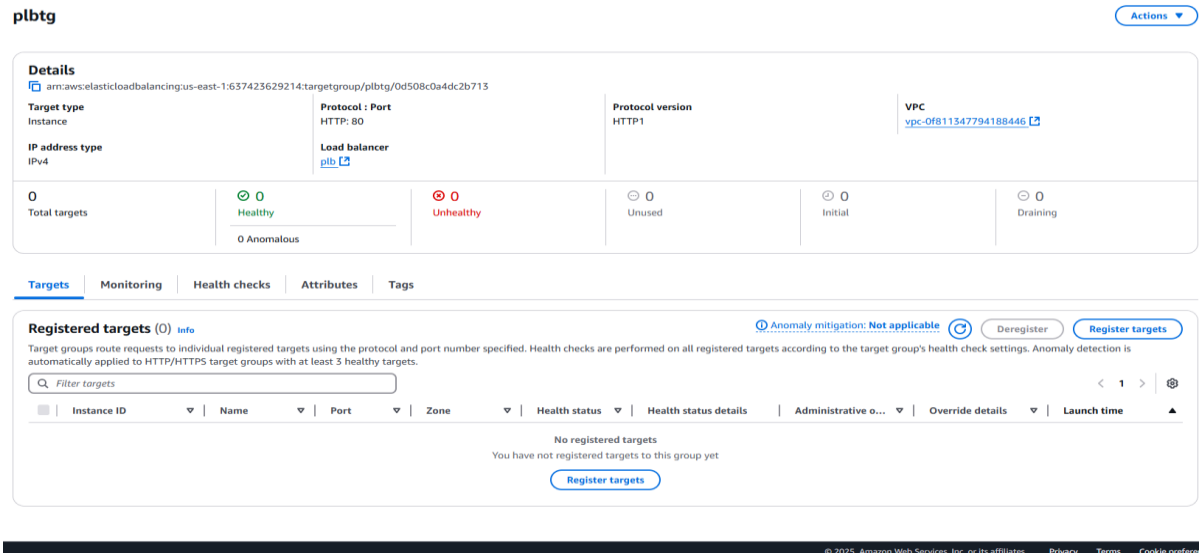
To simulate a failover scenario, I adjusted the desired capacity of the Primary Auto Scaling Group (ASG) to zero. This approach ensured that no new instances would be launched to replace the ones I terminated.

If I had simply terminated the instances without modifying the desired capacity, the ASG would have automatically launched new instances using the launch template, as the primary region itself was still fully operational. This would not have accurately simulated a regional failure, as the primary region would continue to function normally. By setting the desired capacity to zero, I effectively mimicked a complete regional outage, allowing the failover mechanisms to be tested in a realistic manner.



ALB TG Health Checks Failing

After setting the desired capacity of the Primary Auto Scaling Group (ASG) to zero, the EC2 instances began terminating. During this process, the Primary Load Balancer's Target Group continued performing health checks on the instances. These health checks ultimately failed because the instances were being terminated as part of the staged failover simulation. This failure triggered the next steps in the failover process, ensuring the architecture responded as designed.



Once the health checks failed, two things would happen simultaneously:

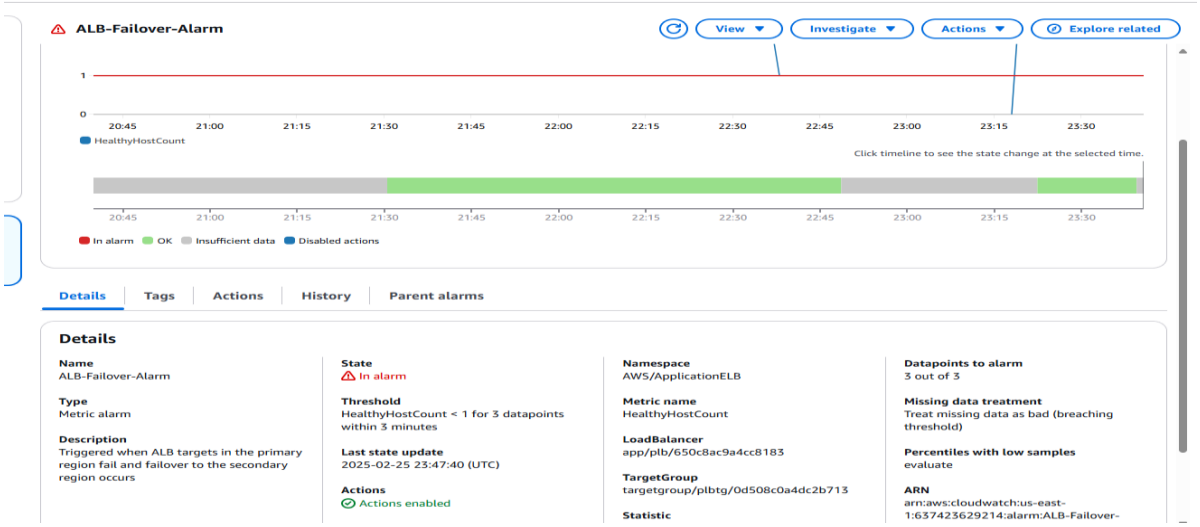
1. **ALB Becomes Unhealthy & Route 53 Triggers Failover**
    - Since Route 53 relied on the overall health of the ALB, once the ALB had no healthy targets, it was marked as unhealthy.
    - As a result, Route 53 failed over to the secondary DNS record, which directed traffic to the failover region's Load Balancer.
2. **CloudWatch Alarm Detects ALB Target Group Failure**
    - A CloudWatch Alarm monitored the "HealthyHostCount" metric for the ALB Target Group.
    - If the count dropped below 1 (meaning all targets were unhealthy), the alarm was triggered.
    - The alarm then sent an SNS notification, providing visibility into the failover event.

Disclaimer: Route 53 Failover Verification

Since this setup had no real domain or incoming traffic, Route 53 failover could not be directly verified. Normally, this could be checked with DNS queries or Route 53 health logs. However, the failover logic was correctly configured, and Route 53 should have switched traffic to the Failover ALB once the Primary ALB became unhealthy.

Cloud watch Alarm being triggered

(later, I'll explain an error I ran into whilst trying to run this configuration and how I worked around it)

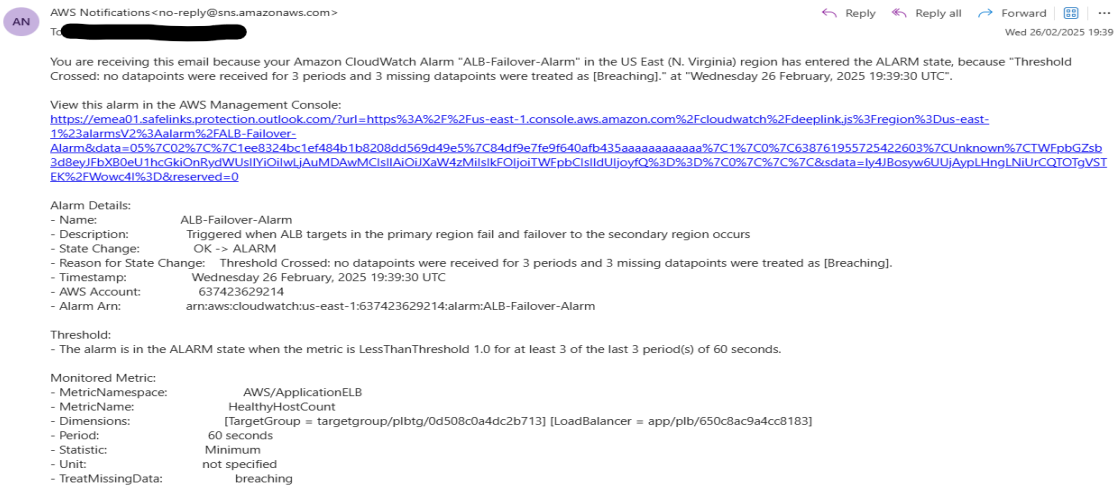


Alarm Invoking the SNS Topic

Once the CloudWatch alarm was triggered, it invoked the SNS topic to act. The SNS topic was configured to do two things:

1. Notify via email when a failover occurred. (Via SNS subscription)
2. Trigger the Lambda function to scale up the failover Auto Scaling Group (ASG) as per the failover setup.

SNS Topic configured to send a failover email alert



AN AWS Notifications<no-reply@sns.amazonaws.com>

Reply Reply all Forward

Wed 26/02/2025 19:39

You are receiving this email because your Amazon CloudWatch Alarm "ALB-Failover-Alarm" in the US East (N. Virginia) region has entered the ALARM state, because "Threshold Crossed: no datapoints were received for 3 periods and 3 missing datapoints were treated as [Breaching]." at "Wednesday 26 February, 2025 19:39:30 UTC".

View this alarm in the AWS Management Console:
https://emea01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fus-east-1.console.aws.amazon.com%2Fcloudwatch%2Fdeeplink.js%3Fregion%3Dus-east-1%23alarmsV2%3Aalarm%2FALB-Failover-Alarm&data=05%7C02%7C%7C1ee8324bc1ef484b1b8208dd569d49e5%7C84df9e7fe9f640afb435aaaaaaaaaaaa%7C1%7C0%7C638761955725422603%7CUnknown%7CTWFpbGZsb3d8eyJFbXB0eU1hcGkiOnRydWUsIlYiOiIwLjAuMDAwMCIsIlAiOiJXaW4zMiIsIkFOIjoiTWFpbCIsIldUIjoyfQ%3D%3D%7C0%7C%7C%7C&sdata=Iy4JBosyw6UUjAypLHngLNiUrCQTOTgVSTEK%2FWowc4I%3D&reserved=0

Alarm Details:
- Name: ALB-Failover-Alarm
- Description: Triggered when ALB targets in the primary region fail and failover to the secondary region occurs
- State Change: OK -> ALARM
- Reason for State Change: Threshold Crossed: no datapoints were received for 3 periods and 3 missing datapoints were treated as [Breaching].
- Timestamp: Wednesday 26 February, 2025 19:39:30 UTC
- AWS Account: 637423629214
- Alarm Arn: arn:aws:cloudwatch:us-east-1:637423629214:alarm:ALB-Failover-Alarm

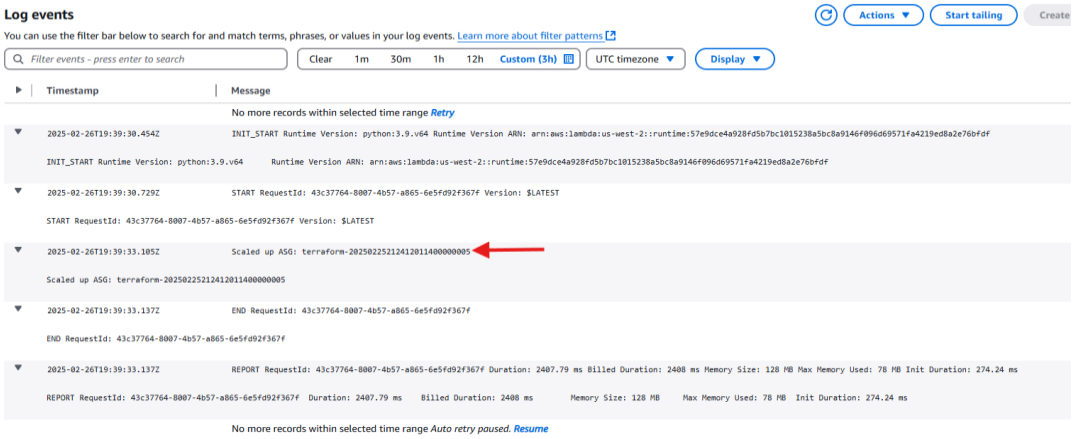
Threshold:
- The alarm is in the ALARM state when the metric is LessThanThreshold 1.0 for at least 3 of the last 3 period(s) of 60 seconds.

Monitored Metric:
- MetricNamespace: AWS/ApplicationELB
- MetricName: HealthyHostCount
- Dimensions: [TargetGroup = targetgroup/plbtg/0d508c0a4dc2b713] [LoadBalancer = app/plb/650c8ac9a4cc8183]
- Period: 60 seconds
- Statistic: Minimum
- Unit: not specified
- TreatMissingData: breaching

Lambda Function invoked via SNS topic

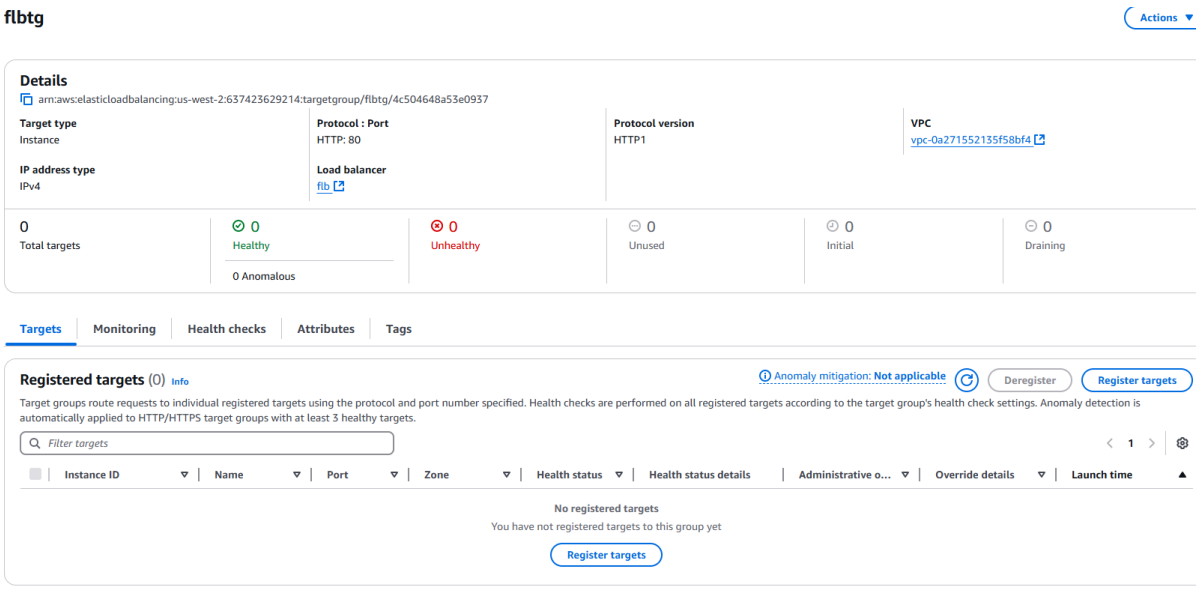
The reason I went with this set up because CloudWatch Alarms and Auto Scaling policies are **region-specific**, meaning the alarm in the primary region couldn't trigger ASG scaling policy in the failover region. To work around this, the alarm invoked an SNS topic, which triggered a Lambda function in the failover region to scale up the ASG. This ensured automated cross-region scaling despite AWS limitations.

Below are the Cloud Watch Logs for the lambda function showing it being invoked.

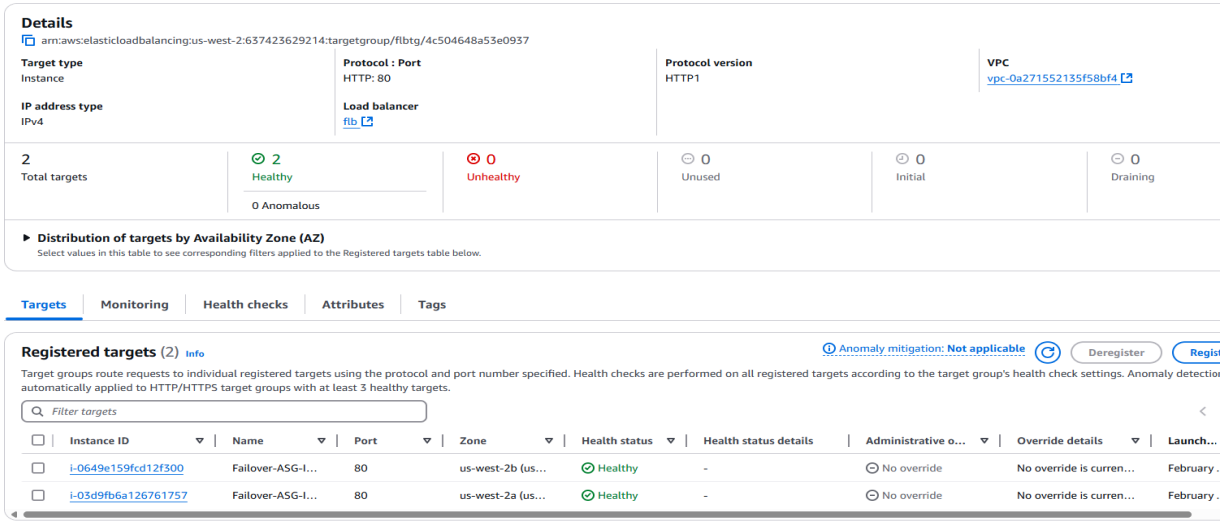


The failover ASG in the secondary region has successfully scaled up, now running two instances that are being load-balanced by the ALB.

Before failover:



After Failover:



As shown in the two screenshots above, the failover has been completed. The failover load balancer's target group now has two instances running, ready to receive traffic from the ALB, which in turn receives its traffic from Route 53. Route 53 failed over earlier after the primary ALB was deemed unhealthy (with no instances running).

This is showcasing the active passive set up that I went with. This approach minimizes costs while ensuring a **low Recovery Time Objective (RTO)** through automated failover and resource scaling.

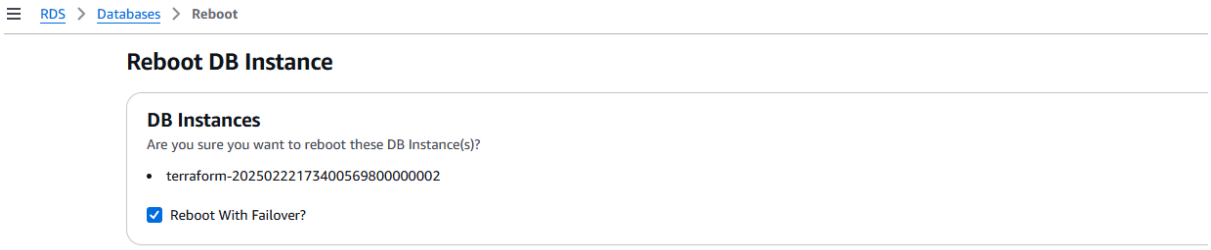
RDS Multi-AZ Failover and Automated Backups

RDS was included in this project to demonstrate how to set up a highly available database with automated backups using AWS Backup. In this simplified setup, a Multi-AZ RDS instance was deployed in the primary region to provide automatic failover within the region. Security groups and private subnets were used to securely connect EC2 instances to the RDS instance, ensuring database access is restricted to internal traffic only.

In a real-world scenario, the RDS instance would be fully integrated with the application, with the EC2 instances connecting to the database for dynamic data storage (e.g., user information, transactions). Additional configurations, such as setting up read replicas for load distribution or integrating IAM roles for fine-grained database access, would be applied to meet production requirements. Monitoring and alerting would also be set up for database health and performance.

RDS Failover event

To simulate an AZ failover I had to do a failover reboot. Now, I will state that since there has been no real AZ fail the new RDS instance will just redeploy back into the same healthy az but this is just purely to check if failover process works.



Cloud watch logs to show RDS Failover

February 25, 2025, 10:11 (UTC+00:00)	The user requested a failover of the DB instance.
February 25, 2025, 10:11 (UTC+00:00)	Multi-AZ instance failover completed
February 25, 2025, 10:11 (UTC+00:00)	DB instance restarted
February 25, 2025, 10:10 (UTC+00:00)	Multi-AZ instance failover started.

Once RDS Az Failover has occurred the CloudWatch event that waits for an RDS Instance failover is invoked

```
# CloudWatch Event Rule for RDS Failover
resource "aws_cloudwatch_event_rule" "rds_failover_rule" {
  name        = "RDS-Failover-Event"
  description = "Triggers when RDS fails over to another AZ"
  event_pattern = jsonencode({
    source      = ["aws.rds"]
    detail-type = ["RDS DB Instance Event"]
    detail = {
      EventCategories = ["failover"]
    }
  })

  tags = {
    Name        = "RDS Failover Event Rule"
    Environment = "production-primary"
  }
}
```

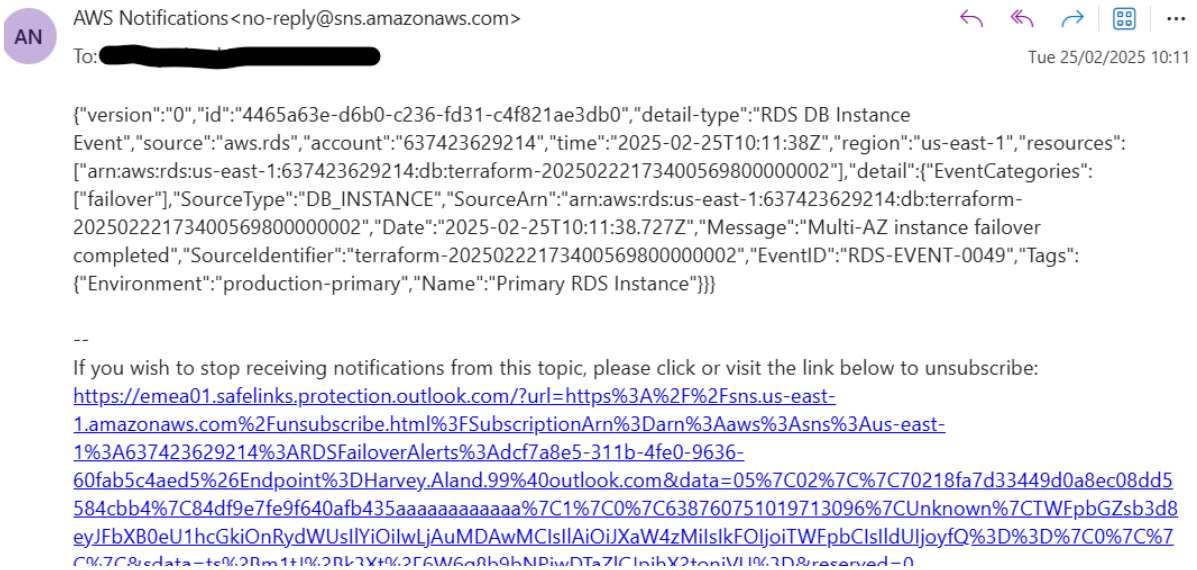
Key Arguments Enabling the Failover Detection in Terraform

The following three arguments in the Terraform code ensure that the CloudWatch Event Rule detects an RDS failover event and triggers the appropriate response:

1. **source** – Specifies the AWS Service Source
    - This defines which AWS service generates the event. In this case, it's set to aws.rds, meaning the rule listens for events from Amazon RDS.
2. **detail-type** – Specifies the Type of Event
    - The value "RDS DB Instance Event" ensures that the rule only captures events related to RDS database instances.
3. **EventCategories** – Filters for Failover Events
    - Setting EventCategories = ["failover"] ensures that only failover-related events are captured, preventing unnecessary triggers from unrelated database events.

This configuration ensures that the CloudWatch rule only listens for failover events, making it an essential component in maintaining high availability and resilience in your infrastructure.

Once the event rule is invoked, the event is published to the SNS Topic, the topic subscription is set up to alert of a RDS Failover Event via email.



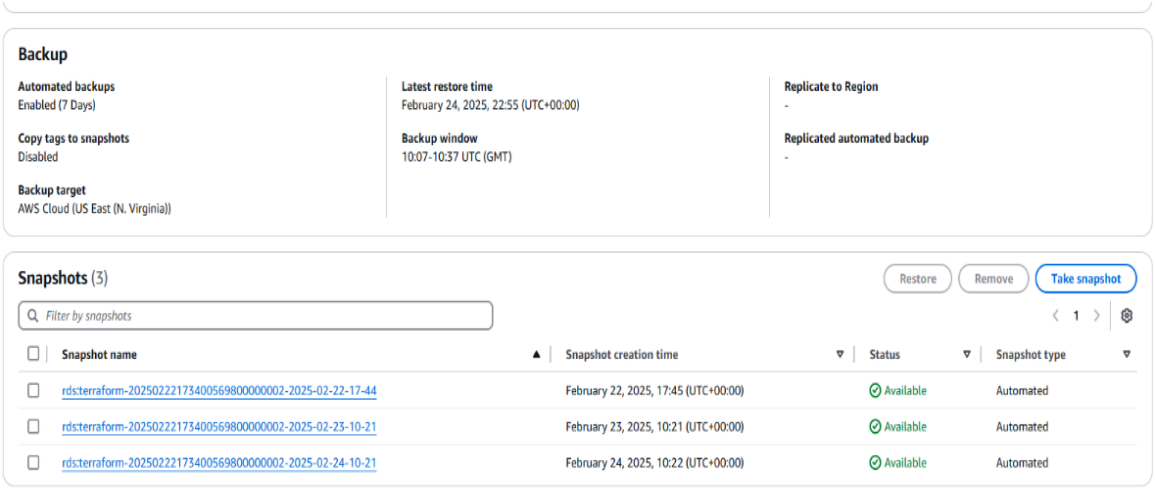
RDS Snapshots & Automated AWS BackUps

I used both RDS managed snapshots and AWS Backup in my configuration because AWS Backup does not support automatic cross-region replication for RDS backups. By using RDS-managed snapshots, I ensured that backups could be replicated across regions if needed for disaster recovery.

AWS Backup, however, provides centralized backup management and automated scheduling, making it useful for maintaining regular backups within the primary region. Combining both methods allowed me to balance automation, retention, and cross-region recovery capabilities.

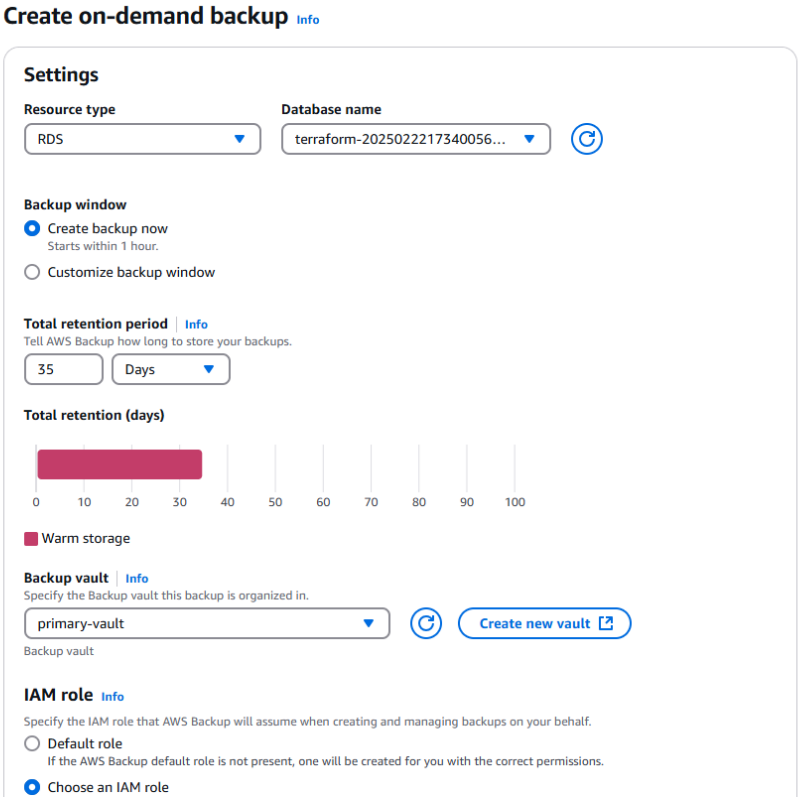
RDS Automated snapshots

As you can see below, RDS is taking daily snapshots with the retention set to 7 days as per the terraform code

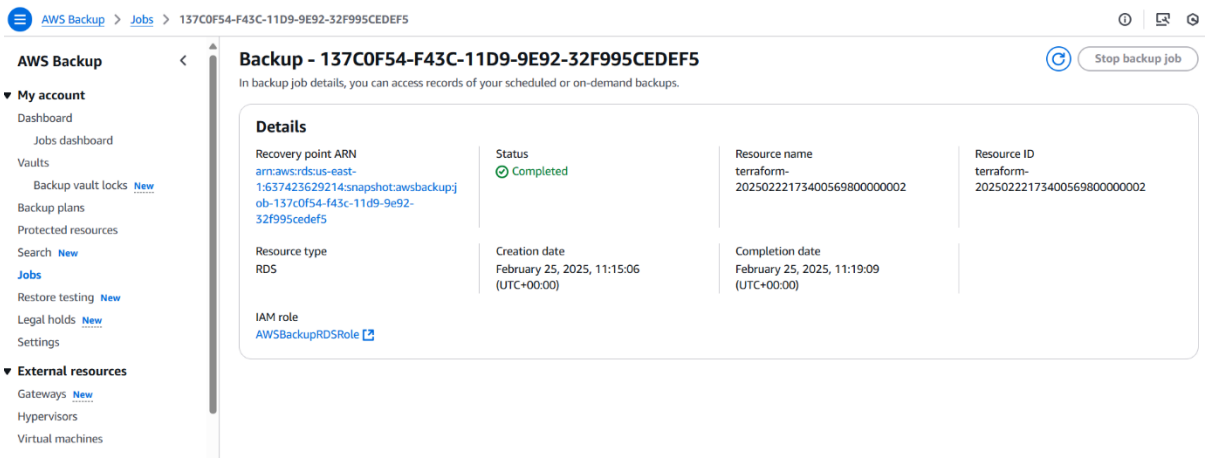


AWS Backup Jobs running and creating backups

To test the automated backup jobs were working, I created a manual job to see if the backup would pass.



Once the manual job had been set, I waited to see if it was successful



Now, while the overall RDS/Backup deployment includes additional infrastructure components, I have chosen to focus on the key working parts that are most relevant to this project, such as failover handling and backup strategies. For a more comprehensive understanding of the full setup, please refer to the RDS section of the Terraform code, which outlines all deployed configurations in detail.

Error handling whilst in deployment and testing

Surprisingly, the deployment process was not entirely smooth, as I encountered a number of errors along the way. Some were resolved with a quick search, while others required hours of troubleshooting. However, this aspect of the project turned out to be the most enjoyable, as it pushed me to dig deeper and expand my knowledge in areas I was less familiar with.

I documented the two major errors I encountered along with the solutions I applied to resolve them, providing a record of the troubleshooting process and lessons learned.

Cloud watch alarm not working when the Primary ALB TG has not healthy instances running (Failover in action)

This one was tricky because in theory everything should have worked fine, I know this because my original failover deployment had the failover occur if the ALB TG had less than two healthy instances (one healthy running). I had tested this set up and the alarm worked fine.

Once I had changed the set up to only failover when there were no healthy instances the alarm did not work. In fact, the alarm itself was in “okay” state when the infrastructure was running normally, but once a failover would occur, the alarm would go into a state

of “insufficient data” instead of “alarm”. This was holding up the rest of the failover process and no SNS topic was being notified of the failover.

Cloud Watch Alarm Stuck in Insufficient Data

The question I was asking myself whilst trouble shooting this problem is why was it working fine before and not now?

The issue occurred because the CloudWatch alarm was set to trigger only when there were no healthy instances in the ALB target group (TG). However, when all instances became unhealthy, CloudWatch lacked enough data points to evaluate the metric, causing the alarm to enter an "insufficient data" state instead of "alarm." This prevented the SNS notification from being triggered, delaying the failover process. To fix this, I adjusted the alarm's evaluation settings to handle missing data more effectively, ensuring it would transition to "alarm" when no healthy instances were detected.

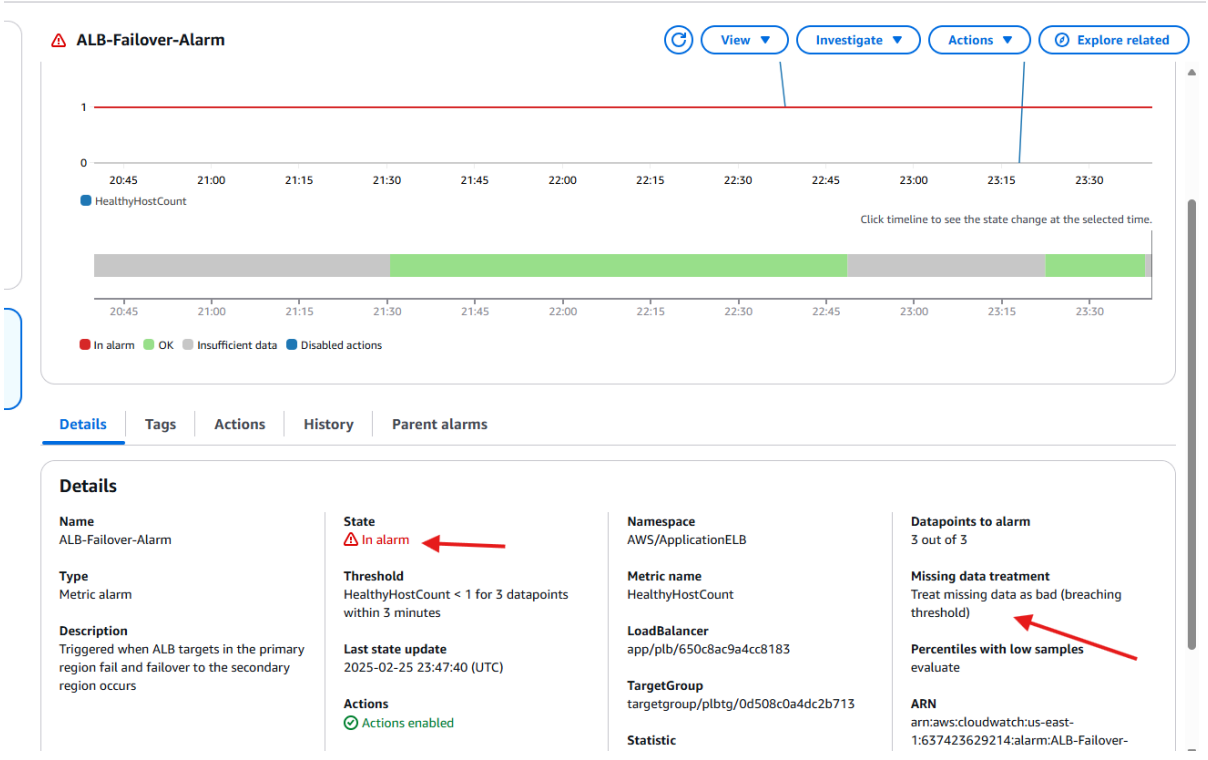
What changes did I make to the CloudWatch alarm metric?

```
# CloudWatch Metric Alarm for ALB Failover
resource "aws_cloudwatch_metric_alarm" "alb_failover_alarm" {
  provider = aws
  alarm_name                = "ALB-Failover-Alarm"
  alarm_description         = "Triggered when ALB targets in the primary region fail and failover to the secondary region occurs"
  metric_name               = "HealthyHostCount"
  namespace                 = "AWS/ApplicationELB"
  statistic                 = "Minimum"
  period                    = 60   # 1-minute period
  evaluation_periods        = 3    # Evaluate for 3 minutes
  threshold                 = 1    # # Triggers if < 1 (so at 0)
  comparison_operator       = "LessThanThreshold"
  dimensions = {
    LoadBalancer = aws_lb.Prim_lb.arn_suffix
    TargetGroup  = aws_lb_target_group.Prim_lb_tg.arn_suffix
  }

  alarm_actions = [aws_sns_topic.ALB_Fail_alerts.arn]

  tags = {
    Name        = "ALB Failover Alarm"
    Environment = "production-primary"
  }
  treat_missing_data = "breaching"  # Fix for handling missing data
}
```

“Treat_missing_data” is set to "**breaching**" to ensure CloudWatch treats missing data (when no healthy instances are available) as breaching the threshold, causing the alarm to enter the "**ALARM**" state instead of "**INSUFFICIENT DATA**" and trigger SNS notifications during failovers.

Once the new code was deployed, I put it to the test.

I simulated a failover by dropping the Primary ASG Capacity as shown earlier, the failover was then set in motion and the alarm was operating as it should.



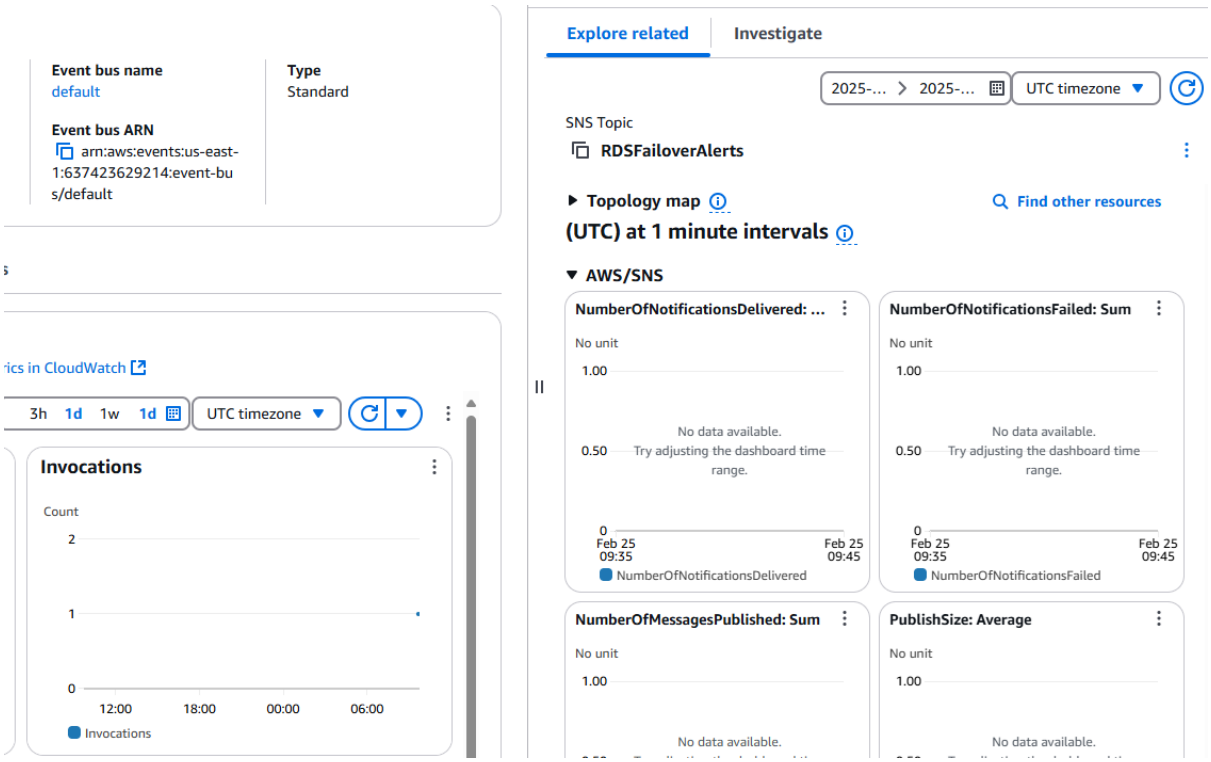
Things to consider:

One this I'd like to point out with the Cloud Watch alarm metric set up I decided to go with is that this is useful in failover situations, but it can possibly be triggered by errors or temporary loss of data. It's a trade-off between ensuring the alarm triggers during failover and avoiding it being triggered by transient issues that might not be actual breaches. Later in the **Final thoughts** section I touch on how the alarm metric I configured can still be utilized if manual review of events in implemented, better aligning with a production focused architecture.

SNS not being triggered when RDS Failover is staged

The issue was that while the RDS event was being triggered, no notifications were being sent to SNS. To troubleshoot this, I had to verify a few things:

- The **CloudWatch Event Rule** was correctly filtering for RDS failover events by checking the EventCategories in the event pattern.
- That the **CloudWatch Event Rule** was actually being invoked
- The **CloudWatch Event Target** was properly linked to the SNS topic.



As you can see above the cloud watch event was being invoked but the SNS topic was not being sent over the event notification.

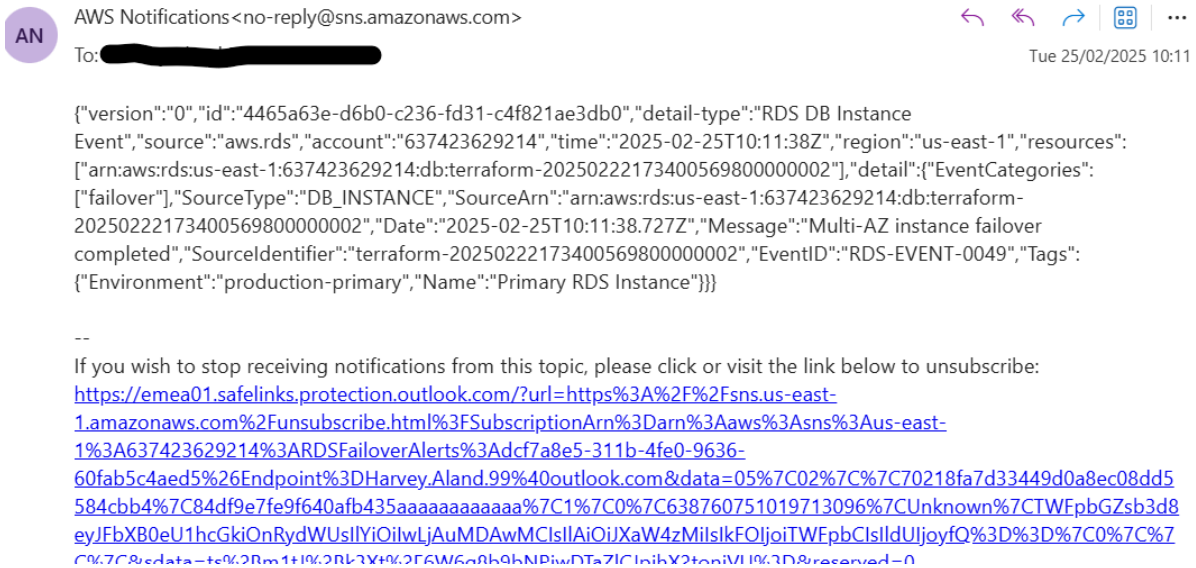
After troubleshooting, I realized the issue was due to a missing SNS topic policy. My CloudWatch Event Rule lacked the necessary permissions to publish notifications to the SNS topic.

Once I updated the policy to allow CloudWatch to publish messages, the SNS topic successfully sent notifications, and I received alerts for the RDS AZ failover via email from the SNS topic subscription.

Updated SNS code

```
# CloudWatch Event Target for RDS Failover SNS
resource "aws_cloudwatch_event_target" "rds_failover_sns" {
  rule      = aws_cloudwatch_event_rule.rds_failover_rule.name
  target_id = "SendToSNS"
  arn       = aws_sns_topic.RDS_Fail_alerts.arn

}

resource "aws_sns_topic_policy" "sns_policy" {
  arn    = aws_sns_topic.RDS_Fail_alerts.arn
  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Effect    = "Allow"
        Action    = "sns:Publish"
        Principal = {
          Service = "events.amazonaws.com"
        }
        Resource  = aws_sns_topic.RDS_Fail_alerts.arn
      }
    ]
  })
}
```

SNS topic now receives the alert of the failover which notify me by email via subscription



AN

AWS Notifications<no-reply@sns.amazonaws.com>

To: [redacted]

Tue 25/02/2025 10:11

{"version":"0","id":"4465a63e-d6b0-c236-fd31-c4f821ae3db0","detail-type":"RDS DB Instance Event","source":"aws.rds","account":"637423629214","time":"2025-02-25T10:11:38Z","region":"us-east-1","resources":["arn:aws:rds:us-east-1:637423629214:db:terraform-20250222173400569800000002"],"detail":{"EventCategories":["failover"],"SourceType":"DB_INSTANCE","SourceArn":"arn:aws:rds:us-east-1:637423629214:db:terraform-20250222173400569800000002","Date":"2025-02-25T10:11:38.727Z","Message":"Multi-AZ instance failover completed","SourceIdentifier":"terraform-20250222173400569800000002","EventID":"RDS-EVENT-0049","Tags":{"Environment":"production-primary","Name":"Primary RDS Instance"}}}

--

If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:

https://emea01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fsns.us-east-1.amazonaws.com%2Funsubscribe.html%3FSubscriptionArn%3Darn%3Aaws%3Asns%3Aus-east-1%3A637423629214%3ARDSFailoverAlerts%3Adcf7a8e5-311b-4fe0-9636-60fab5c4aed5%26Endpoint%3DHarvey.Aland.99%40outlook.com&data=05%7C02%7C%7C70218fa7d33449d0a8ec08dd584cbb4%7C84df9e7fe9f640afb435aaaaaaaaaaaa%7C1%7C0%7C638760751019713096%7CUnknown%7CTWFpbGZsb3d8eyJFbXB0eU1hcGkiOnRydWUsIlYiOiIwLjAuMDAwMCIsIlAiOiJXaW4zMiIsIkFOIjoiTWFpbCIsIldUIjoyfQ%3D%3D%7C0%7C%7C%7C&sdata=ts%2Bm1tU%2Bk3Xt%2F6W6q8b9bNPiwDTaZlCUpjbX2tonjVU%3D&reserved=0

What I learnt from handling this error

Although this error seems straight forward, at the time I couldn't get my head around why my cloud watch alarm set up for the ALB failover was working but now im using cloud watch events for RDS it was not, which led me to discover the following:

What I Learned from Handling This Error

At first, I couldn't understand why my CloudWatch Alarm for ALB failover was successfully triggering SNS notifications, while my CloudWatch Event Rule for RDS failover was not. This led me to discover a key difference in how these services interact with SNS.

- **CloudWatch Alarms:**
    - AWS automatically grants the necessary permissions for alarms to publish to SNS.
    - Since alarms are a service-integrated feature, they don't require a separate SNS topic policy.
- **CloudWatch Event Rules (EventBridge):**
    - Unlike alarms, Event Rules do not have automatic permissions to publish to SNS.

- Because EventBridge operates independently, an explicit SNS topic policy is required to allow it to send notifications.
- Without this policy, the Event Rule was triggering but failing to send messages to SNS.

Summary:

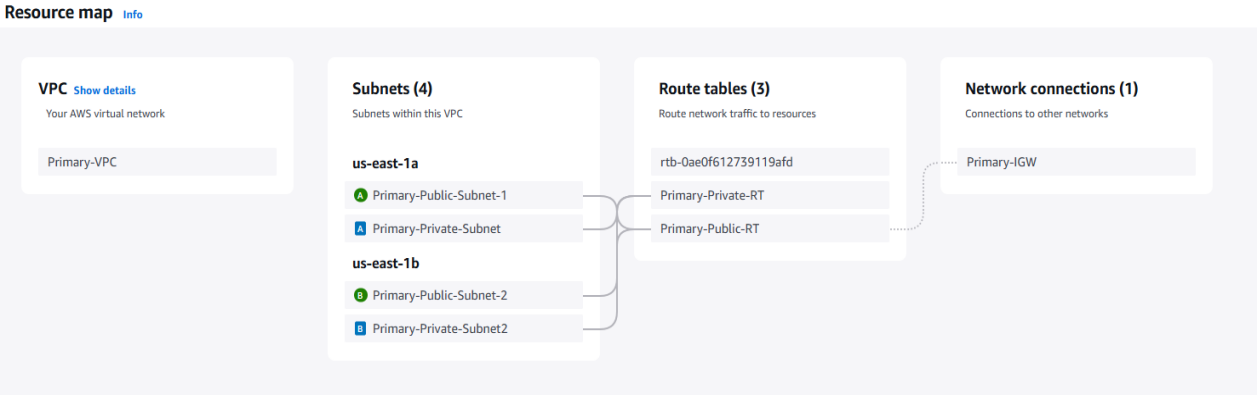
- **Alarms don't need a topic policy** because AWS manages permissions automatically.
- **Event Rules need a topic policy** because they rely on manual permission configuration to send messages to SNS.

Networking

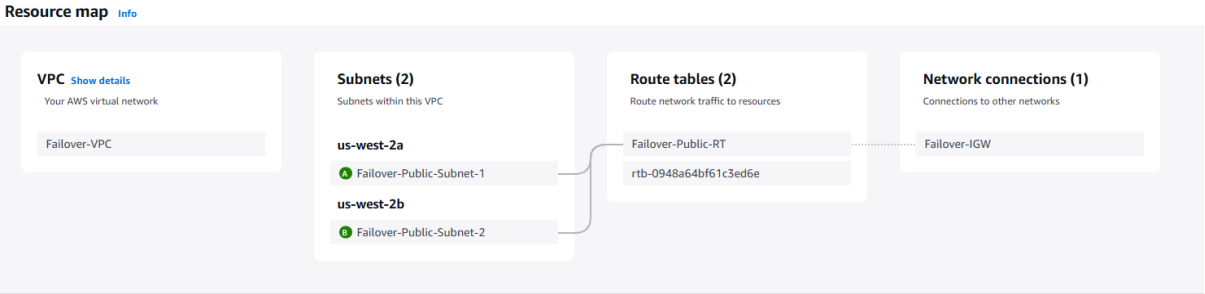
My setup is a multi-region, highly available architecture with a primary region and a failover region. Each region has its own VPC, with public subnets for web servers and private subnets for isolating critical resources like RDS databases. The private subnets are completely isolated from the internet, ensuring that the RDS instances are only accessible from within the VPC, such as from the EC2 instances in the public subnets or other trusted resources. Both VPCs are configured with internet gateways and route tables to allow internet access for the public subnets while keeping the private subnets secure.

For compute, I've set up Auto Scaling Groups (ASGs) in both regions, each using a launch template to deploy EC2 instances. These instances are load-balanced using Application Load Balancers (ALBs) with target groups and listeners to handle HTTP traffic. Security groups are in place to restrict access to the instances and ALBs, allowing only necessary traffic (e.g., HTTP on port 80). Additionally, IAM roles are configured to grant the EC2 instances the minimum permissions required to function, such as logging and interacting with the load balancers.

Primary Region Network Map



Failover Region Network Map



No private subnets in this region as the RDS DB was deployed in the Primary region

---

HTTPS Listener Consideration

One aspect not implemented in this deployment but essential for a fully functioning, production-ready application is an HTTPS listener for the Application Load Balancer (ALB).

For this project, HTTP (port 80) was intentionally used instead of HTTPS to simplify development and testing. Since this is a learning exercise without production traffic or sensitive data, HTTP reduces complexity by:

1. Avoiding certificate management – No need to provision and validate ACM certificates.
2. Reducing costs – No expenses for SSL/TLS certificates or domain registration.

While HTTPS is critical for real-world application ensuring security, SEO benefits, and compliance with browser requirement. This project prioritizes core AWS infrastructure components focusing on a failover architecture deployment.

Just to note that I have utilised HTTPs listener and ACM in my Cloud Resume challenge.

Future Enhancements and Improvements for the Project

Moving forward I would like to implement the following to make the architecture more in line with a real-world deployment:

Cross-Region Replication for RDS:
Set up cross-region replication for the RDS databases.

Automated Detection of Primary Region Recovery

Create an AWS Lambda function that is triggered by a CloudWatch Alarm monitoring the healthy instances in the ALB target group. Once the alarm detects that the required number of instances in the primary region are healthy, Lambda (being triggered by SNS) can initiate the failback process. Route 53 should automatically fail back based on the existing deployment, but testing would be required to verify that traffic correctly returns to the primary region.

Project Summery

The architecture for this infrastructure is an ideal representation of my understanding of AWS resources, designed to closely mimic a real-world deployment. This project has been an incredible opportunity to apply the knowledge and experience I gained from earning my **AWS Certified Cloud Practitioner (CCP)** and **AWS Solutions Architect Associate (SSA)** certifications. By fully deploying this infrastructure as **Infrastructure as Code (IaC)** using Terraform, I've not only showcased my ability to navigate the AWS console for testing and documentation but also demonstrated my proficiency in leveraging Terraform to its full potential.

This document was created for the purpose of breaking down the functional aspects of the project, how the design works and some error handling. I chose not to touch on all the deployed infrastructure as I thought it would take away from the focus of the project which is creating an Active passive failover networking architecture.

Attached in the Github is a file with the full code annotated with for easy reading, please refer to this when reviewing all resources used.

Final Thoughts

Through additional research on existing regional outage failover deployments, I've found that most companies in production environments avoid automatic failover. The primary reason is that regional outages are often resolved quickly, and deploying a full-scale production infrastructure for failover can be unnecessarily costly.

Outages are inevitable and should be factored into AWS architecture design. Application owners should also be made aware of this reality so they can determine the

best approach for their business need, whether automated or manual failover is more appropriate.

Based on my current experience as a technology specialist, any outage or failure event is typically escalated to the relevant team for review before action is taken. With this in mind, we can optimize the current deployment by modifying the Lambda function's role. Once the SNS topic sends an email alert, we can remove the Lambda function as an SNS subscriber and reconfigure it for manual invocation. This way, a Major Incident Manager can trigger the Lambda function only after investigating the outage, ensuring a more controlled and cost-effective response.