Problem

We face a challenge related to disaster recovery (DR), which is a critical aspect of any robust infrastructure strategy. In the event of a disaster or a major outage, we must have a reliable and automated recovery process to minimize downtime and ensure business continuity. However, manually setting up and managing DR infrastructure is complex, time-consuming, and prone to errors.

Solution

We use Terraform to automate the provisioning and management of disaster recovery infrastructure on AWS. Terraform's declarative syntax, combined with AWS services like Amazon S3, Amazon Elastic Block Store (EBS), and AWS Elastic Disaster Recovery (DRS), allows us to define and deploy a complete DR solution. Below, we explain how we implement this solution.

Implementation

1. AWS Providers

We configure two AWS providers: one for the primary region (`us-west-2`) and another for the DR region (`us-east-1`). We use an alias for the DR provider to differentiate them.

```
provider "aws" {
  region = "us-west-2"  # Primary region
}

provider "aws" {
  alias  = "dr"
  region = "us-east-1"  # DR region
}
```

2. S3 Bucket for Backup Storage

We create an S3 bucket with versioning enabled to store backups. Additionally, we define lifecycle rules to transition data to **GLACIER** storage after 30 days and delete objects after 90

days.

```
resource "aws_s3_bucket" "backup" {
  bucket = "example-dr-backup-bucket"
  acl    = "private"

  versioning {
    enabled = true
  }

  lifecycle_rule {
    enabled = true

    transition {
      days          = 30
      storage_class = "GLACIER"
    }

    expiration {
      days = 90
    }
  }
}
```

3. EC2 Instance and EBS Volume

We set up an EC2 instance as the primary server and create an EBS volume to store data. We attach the EBS volume to the instance.

```
resource "aws_instance" "primary" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"

  tags = {
    Name = "primary-instance"
  }
}

resource "aws_ebs_volume" "primary_data" {
  availability_zone = aws_instance.primary.availability_zone
  size              = 100
  type              = "gp3"

  tags = {
    Name = "primary-data-volume"
```

```
  }
}

resource "aws_volume_attachment" "primary_data_attachment" {
  device_name = "/dev/sdh"
  volume_id   = aws_ebs_volume.primary_data.id
  instance_id = aws_instance.primary.id
}
```

4. Elastic Disaster Recovery (DRS) Configuration

We use AWS DRS to set up replication for disaster recovery. This includes creating a replication configuration template and enabling encryption with a KMS key.

```
resource "aws_drs_replication_configuration_template" "example" {
  source_server {
    instance_type = "t3.micro"

    tags = {
      Name = "DR-Replica"
    }
  }

  ebs_encryption {
    kms_key_id = aws_kms_key.dr_key.arn
  }
}

resource "aws_kms_key" "dr_key" {
  description             = "KMS key for DR encryption"
  deletion_window_in_days = 10
  enable_key_rotation     = true
}
```

5. Automating Failover with Lambda and CloudWatch

We configure a CloudWatch event rule to monitor EC2 outages in the primary region. This rule triggers a Lambda function to handle the failover process. We also create an IAM role to grant necessary permissions to the Lambda function.

```
resource "aws_cloudwatch_event_rule" "dr_failover" {
  name        = "dr-failover-trigger"
  description = "Triggers DR failover process"
```

```
  event_pattern = jsonencode({
    "source": ["aws.health"],
    "detail-type": ["AWS Health Event"],
    "detail": {
      "service": ["EC2"],
      "eventTypeCategory": ["issue"],
      "region": ["us-west-2"]
    }
  })
}

resource "aws_lambda_function" "dr_failover" {
  filename      = "dr_failover_function.zip"
  function_name = "dr-failover-handler"
  role          = aws_iam_role.dr_lambda_role.arn
  handler       = "index.handler"
  runtime       = "nodejs14.x"

  environment {
    variables = {
      DR_CONFIGURATION_ID =
aws_drs_replication_configuration_template.example.id
    }
  }
}

resource "aws_iam_role" "dr_lambda_role" {
  name = "dr-lambda-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "lambda.amazonaws.com"
        }
      }
    ]
  })
}

resource "aws_cloudwatch_event_target" "dr_failover" {
  rule      = aws_cloudwatch_event_rule.dr_failover.name
  target_id = "TriggerDRFailover"
```

```
  arn       = aws_lambda_function.dr_failover.arn
}
```

Summary

We use Terraform to automate the deployment of a disaster recovery solution on AWS. By leveraging services like S3, DRS, and Lambda, we streamline the recovery process, reduce complexity, and minimize errors compared to manual setups. This approach ensures a reliable and efficient strategy for maintaining business continuity.

Discussion

Automating disaster recovery with Terraform and AWS provides key benefits such as reproducibility, consistency, version control, automated failover, and cost optimization. By defining infrastructure as code, we ensure a reliable and efficient DR process.

Best Practices:

- Conduct regular DR drills to test failover processes.
- Ensure proper data synchronization and network configuration between regions.
- Implement robust monitoring, alerting, and least privilege IAM roles.
- Maintain clear documentation and meet compliance requirements.
- Regularly review costs and optimize resource usage.

These practices help us create a resilient and cost-effective DR solution.