AWS Projects

1. Web Hosting & Deployment

Traditional & Scalable Hosting

Project 1: Static Website Hosting – Host a website on S3 with Route 53 & CloudFront.

Project 2: WordPress on AWS Lightsail – Deploy a WordPress blog on AWS Lightsail.

Project 3: EC2-based Web Server – Configure Apache/Nginx on an EC2 instance.

Project 4: Scalable Web App with ALB & Auto Scaling – Use EC2, ALB, and Auto Scaling for high availability.

Project 5: Scalable Web App with NLB & Auto Scaling – Use EC2, ALB, and Auto Scaling for high availability.

Project 6: Multi-Tier Web App Deployment – Deploy an application with VPC, ALB, EC2, and RDS.

---

2. Serverless Application & Modern Web Hosting

Project 1: Serverless Static Website – Use S3, CloudFront, and Lambda@Edge for global performance.

Project 2: Multi-Tenant SaaS Application – Use Cognito, API Gateway, and RDS for a multi-tenant SaaS solution.

Project 3: Progressive Web App (PWA) Hosting – Deploy a React or Vue.js PWA on AWS Amplify.

Project 4: Jamstack Site with AWS – Use AWS Amplify, Headless CMS, and S3 for a Jamstack website.

Project 5: AWS App Runner Deployment – Deploy a containerized web application with AWS App Runner.

Project 6: Serverless API with Lambda & API Gateway – Build a RESTful API using Lambda.

Project 7: GraphQL API with AWS AppSync – Deploy a GraphQL backend using DynamoDB.

Project 8: Serverless Chatbot – Build an AI-powered chatbot with AWS Lex & Lambda.

Project 9: Event-Driven Microservices – Use SNS & SQS for asynchronous microservices communication.

Project 10: URL Shortener – Implement a Lambda-based URL shortener with DynamoDB.

Project 11: Serverless Expense Tracker – Use Lambda, DynamoDB, and API Gateway to build an expense management app.

Project 12: Serverless Image Resizer – Automatically resize images uploaded to an S3 bucket using Lambda.

Project 13: Real-time Polling App – Use WebSockets with API Gateway and DynamoDB for a real-time polling system.

Project 14: Serverless Sentiment Analysis – Analyze tweets using AWS Comprehend and store results in DynamoDB.

---

3. Infrastructure as Code (IaC) & Automation

Project 1: Terraform for AWS – Deploy VPC, EC2, and RDS using Terraform.

Project 2: CloudFormation for Infrastructure as Code – Automate AWS resource provisioning.

Project 3: AWS CDK Deployment – Use AWS Cloud Development Kit (CDK) to manage cloud infrastructure.

Project 4: Automate AWS Resource Tagging – Use AWS Lambda and Config Rules for enforcing tagging policies.

Project 5: AWS Systems Manager for Automated Patching – Manage EC2 instance patching automatically.

Project 6: Automated Cost Optimization – Use AWS Lambda to detect and stop unused EC2 instances.

Project 7: Self-Healing Infrastructure – Use Terraform to create auto-healing EC2 instances with AWS Auto Scaling.

Project 8: Event-Driven Infrastructure Provisioning – Use AWS EventBridge and CloudFormation to automate infrastructure deployment.

Project 9: Automated EC2 Scaling Based on Load – Use Auto Scaling policies with CloudWatch to optimize cost and performance.

---

4. Security & IAM Management

Project 1: IAM Role & Policy Management – Secure AWS resources using IAM policies.

Project 2: AWS Secrets Manager for Credential Storage – Secure database credentials and API keys.

Project 3: API Gateway Security – Implement authentication with Cognito and protection using AWS WAF.

Project 4: AWS Security Hub & GuardDuty – Monitor security threats and enforce compliance.

Project 5: Automated Compliance Auditing – Use AWS Config to check security best practices.

Project 6: Zero Trust Security Implementation – Enforce strict security rules using AWS IAM and VPC security groups.

Project 7: CloudTrail Logging & SIEM Integration – Integrate AWS CloudTrail logs with a SIEM tool for security monitoring.

Project 8: AWS WAF for Application Security – Protect a web application from SQL injection and XSS attacks using AWS WAF.

---

5. CI/CD Projects:

Project 1: AWS CodePipeline for Automated Deployments

Project 2: GitHub Actions + AWS ECS Deployment

Project 3: Jenkins CI/CD with Terraform on AWS

Project 4: Kubernetes GitOps with ArgoCD on AWS EKS

Project 5: Lambda-based Serverless CI/CD

Project 6: Multi-Account CI/CD Pipeline with AWS CodePipeline

Project 7: Containerized Deployment with AWS Fargate

Project 8: Blue-Green Deployment on AWS ECS

---

6. DevOps Projects:

Project 1: Infrastructure as Code (IaC) using Terraform on AWS

Project 2: AWS Auto Scaling with ALB and CloudWatch

Project 3: AWS Monitoring & Logging using ELK and CloudWatch

Project 4: Automated Patch Management with AWS SSM

Project 5: AWS Security Hardening with IAM & Config Rules

Project 6: Disaster Recovery Setup with AWS Backup & Route 53

Project 7: Cost Optimization using AWS Lambda to Stop Unused EC2s

Project 8: Self-Healing Infrastructure with AWS Auto Scaling

---

7. Database & Storage Solutions

Project 1: MySQL RDS Deployment – Set up and manage an RDS database.

Project 2: DynamoDB CRUD Operations with Lambda – Create a serverless backend with DynamoDB.

Project 3: Data Warehouse with AWS Redshift – Store and analyze structured data in Redshift.

Project 4: ETL with AWS Glue – Transform data from S3 to Redshift using AWS Glue.

Project 5: Graph Database with Amazon Neptune – Store relational data in a graph-based model.

Project 6: Automated S3 Data Archival – Use S3 lifecycle rules to move data to Glacier.

Project 7: Automated Database Migration – Use AWS Database Migration Service (DMS) to migrate a database from MySQL to Aurora.

---

8. Monitoring & Logging

Project 1: Monitor AWS Resources with CloudWatch & SNS – Set up alerts for EC2, RDS, and S3.

Project 2: Log Analysis with ELK Stack – Deploy Elasticsearch, Logstash, and Kibana on AWS.

Project 3: Serverless URL Monitor – Use Lambda & CloudWatch to check website uptime.

Project 4: Query S3 Logs with AWS Athena – Analyze stored logs without setting up a database.

Project 5: Application Performance Monitoring – Use AWS X-Ray to trace application requests and optimize performance.

---

9. Networking & Connectivity

Project 1: Route 53 DNS Management – Configure and manage DNS records for custom domains.

Project 2: AWS Transit Gateway for Multi-VPC Communication – Connect multiple AWS VPCs securely.

Project 3: Kubernetes on AWS (EKS) – Deploy a containerized workload using Amazon EKS.

Project 4: Hybrid Cloud Setup with AWS VPN – Connect an on-premises data center to AWS.

---

10. Data Processing & Analytics

Project 1: Big Data Processing with AWS EMR – Use Apache Spark to analyze large datasets.

Project 2: Real-Time Data Streaming with Kinesis – Process IoT or social media data in real time.

Project 3: AWS Quicksight for BI Dashboard – Visualize business data using Amazon QuickSight.

Project 4: AWS Athena for Log Analysis – Query CloudTrail and VPC Flow Logs using Athena.

---

11. AI/ML & IoT

Project 1: Train & Deploy ML Models with SageMaker – Use AWS SageMaker for predictive analytics.

Project 2: Fraud Detection System – Build an ML-based fraud detection model on AWS.

Project 3: AI-Powered Image Recognition – Use AWS Rekognition to analyze images.

Project 4: IoT Data Processing with AWS IoT Core – Store and analyze IoT sensor data in AWS.

---

12. Disaster Recovery & Backup

Project 1: Multi-Region Disaster Recovery Plan – Set up cross-region replication for S3 & RDS.

Project 2: AWS Backup for EBS Snapshots & Recovery – Automate backups and restore EC2 instances.

Project 3: Automated AMI Creation & Cleanup – Schedule AMI snapshots and delete old backups.

---

1. Web Hosting & Deployment

Traditional & Scalable Hosting

Project 1: Static Website Hosting – Host a website on S3 with Route 53 & CloudFront.

Static Website Hosting using S3, Route 53 & CloudFront – Complete Steps

Hosting a static website on **Amazon S3**, integrating it with **Route 53** for domain name resolution, and using **CloudFront** for content delivery enhances performance, security, and reliability.

Prerequisites:

✅ An **AWS account**
✅ A **registered domain** (either in Route 53 or another provider)

Step 1: Create an S3 Bucket for Website Hosting

1. Open the **AWS S3 Console** and click **Create bucket**.
2. **Bucket name**: Enter your domain name (e.g., example.com).
3. **Region**: Select your preferred AWS region.
4. **Disable Block Public Access**:
   - Uncheck **"Block all public access"** and confirm changes.
5. **Enable Static Website Hosting**:
   - Go to the **Properties** tab.
   - Click **Edit** under **Static website hosting**.
   - Select **Enable**.
   - Choose **"Host a static website"**.
   - Set **index document** as index.html.
   - Note the **Bucket Website Endpoint URL**.
6. Click **Create bucket**.

Step 2: Upload Website Files

1. Open your S3 bucket.
2. Click **Upload** → Add your index.html, style.css, script.js, etc.
3. Click **Upload**.

Step 3: Configure S3 Bucket Policy for Public Access

1. Go to the **Permissions** tab.

Under **Bucket Policy**, click **Edit** and paste the following JSON policy:
json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::example.com/*"
    }
  ]
}
```

2. Replace example.com with your bucket name.
3. Click **Save changes**.

Step 4: Register a Domain (if needed) in Route 53

1. Open **AWS Route 53 Console**.
2. Click **Domains → Register Domain**.
3. Search for and purchase a domain (e.g., example.com).
4. Wait for AWS to complete registration.

Step 5: Create a Hosted Zone in Route 53

1. In Route 53, go to **Hosted Zones**.
2. Click **Create hosted zone**.
3. Enter your domain name (example.com).
4. Choose **Public Hosted Zone**.
5. Click **Create**.

Step 6: Set Up CloudFront for Content Delivery

1. Open **AWS CloudFront Console**.
2. Click **Create distribution**.
3. Under **Origin**, configure:
    - **Origin domain**: Select your S3 bucket.
    - **Origin access**: Select **Public**.
    - **Viewer Protocol Policy**: Choose **Redirect HTTP to HTTPS**.
4. **Alternate Domain Names (CNAMEs)**:
    - Enter your domain name (example.com).
5. **Custom SSL Certificate**:
    - Click **Request or Import a Certificate with ACM**.
    - Request an SSL certificate for your domain.
    - Validate via email or DNS.
    - Once issued, attach it to the CloudFront distribution.

6. Click **Create distribution** and wait for deployment.

Step 7: Update Route 53 DNS Records

1. Go to **Route 53 → Hosted Zone →** Select your domain.
2. Click **Create record**.
3. Select **Simple routing → Define simple record**.
4. **Record Name**: Leave empty (for root domain).
5. **Record Type**: Select **A – IPv4 Address**.
6. **Route Traffic To**: Choose **Alias to CloudFront Distribution**.
7. Select your **CloudFront distribution** from the dropdown.
8. Click **Create record**.

Step 8: Verify and Test the Website

- Open a browser and visit your **domain (https://example.com)**.
- If it doesn't work immediately, wait for DNS propagation (~30 mins to a few hours).

Summary of What We Did:

✅ Created an **S3 bucket** for hosting static website files.
✅ Configured **public access & bucket policy** for static hosting.
✅ Registered a **domain** and set up a **hosted zone** in Route 53.
✅ Created a **CloudFront distribution** for security & performance.
✅ Configured **Route 53 DNS records** to point the domain to CloudFront.
✅ Secured the website with **SSL (HTTPS)** using **AWS ACM**.

🚀 Your static website is now hosted on AWS S3, accessible via a custom domain with CloudFront caching!

Project 2: WordPress on AWS Lightsail – Deploy a WordPress blog on AWS Lightsail.

Deploying a WordPress Blog on AWS Lightsail

AWS Lightsail is a cost-effective and beginner-friendly platform for deploying applications, including WordPress. Lightsail simplifies cloud hosting by providing pre-configured virtual private servers (VPS), making it ideal for WordPress blogs.

Steps to Deploy WordPress on AWS Lightsail

1. Sign in to AWS and Navigate to Lightsail

- Go to AWS Lightsail Console.
- Click on **"Create instance"** to set up a new server.

2. Choose an Instance Location

- Select the region closest to your target audience for better performance.

3. Select an Instance Image

- Choose **"Apps + OS"**, then select **WordPress** (which comes pre-installed).
- Alternatively, you can choose **Linux/Unix + WordPress** if you want more customization.

4. Choose an Instance Plan

- Select a pricing plan based on your expected traffic.
- For a small blog, the **$5/month plan** is a good starting point.

5. Name and Create the Instance

- Give your instance a meaningful name (e.g., my-wordpress-blog).
- Click **"Create instance"**, and wait for AWS to provision it.

6. Access Your WordPress Site

- Once the instance is running, go to the **Networking** tab in Lightsail.
- Copy the **public IP address** and paste it into your browser to open your WordPress site.

7. Retrieve WordPress Admin Credentials

- Connect to your instance using **SSH** (Click "Connect using SSH" in Lightsail).

Run the following command to get your WordPress admin password:
cat bitnami_application_password

- Note the password and use it to log in at http://your-public-ip/wp-admin/.

8. Configure Static IP (Recommended)

- Go to the **Networking** tab and create a **Static IP** to prevent your site from changing its IP on reboot.
- Attach the static IP to your instance.

9. Configure a Custom Domain (Optional)

- Register a domain from AWS Route 53 or any domain provider.
- Update your domain's **A record** to point to your static IP.

10. Enable SSL for HTTPS (Recommended)

Install a free SSL certificate using Let's Encrypt with the following commands:
sudo /opt/bitnami/bncert-tool

- Follow the prompts to enable HTTPS.

11. Secure and Optimize WordPress

- Update WordPress, themes, and plugins.
- Use security plugins like **Wordfence**.
- Optimize performance with caching plugins like **W3 Total Cache** or **LiteSpeed Cache**.

12. Backup Your Instance (Recommended)

- In Lightsail, go to the **Snapshots** tab and create a backup.
- You can also enable automatic snapshots.

Final Thoughts

Deploying WordPress on AWS Lightsail is a fast and cost-effective way to start a blog. With a few additional steps like securing your site with SSL, setting up a static IP, and using a custom domain, you can create a professional and reliable WordPress website.

Project 3: EC2-based Web Server – Configure Apache/Nginx on an EC2 instance.

In this project, you will deploy a web server using **Amazon EC2**. You will launch an **EC2 instance**, install either **Apache or Nginx**, and configure it to serve a basic website.

Steps to Set Up an EC2-based Web Server

Step 1: Launch an EC2 Instance

1. Go to the **AWS Management Console → EC2**.
2. Click **Launch Instance**.
3. Configure the instance:
    - **Name**: MyWebServer
    - **AMI**: Choose Amazon Linux 2 or Ubuntu
    - **Instance Type**: t2.micro (Free Tier eligible)
    - **Key Pair**: Create a new key pair or use an existing one.
    - **Security Group**: Allow **SSH (port 22)** and **HTTP (port 80)**.
4. Click **Launch**.

Step 2: Connect to the EC2 Instance

Open a terminal and run:
ssh -i your-key.pem ec2-user@your-instance-public-ip

1. (For Ubuntu, use ubuntu instead of ec2-user)

Step 3: Install Apache or Nginx

For Apache (httpd)

Update the package repository:
sudo yum update -y # Amazon Linux

sudo apt update -y # Ubuntu

Install Apache:
sudo yum install httpd -y # Amazon Linux

sudo apt install apache2 -y # Ubuntu

Start and enable Apache:
sudo systemctl start httpd # Amazon Linux

sudo systemctl enable httpd # Amazon Linux

sudo systemctl start apache2 # Ubuntu

sudo systemctl enable apache2 # Ubuntu

Verify installation:
sudo systemctl status httpd # Amazon Linux

sudo systemctl status apache2 # Ubuntu

For Nginx

Install Nginx:
sudo yum install nginx -y # Amazon Linux

sudo apt install nginx -y # Ubuntu

Start and enable Nginx:
sudo systemctl start nginx

sudo systemctl enable nginx

Verify installation:
sudo systemctl status nginx

Step 4: Configure Firewall

Allow HTTP traffic:

sudo firewall-cmd --permanent --add-service=http

sudo firewall-cmd --reload

(For Ubuntu, this step is not needed if using a security group in AWS)

Step 5: Deploy a Web Page

Edit the default index file:
sudo echo "<h1>Welcome to My Web Server</h1>" > /var/www/html/index.html

1. (For Nginx, use /usr/share/nginx/html/index.html)

Step 6: Access the Web Server

Open a browser and go to:
http://your-instance-public-ip

1. You should see **"Welcome to My Web Server"**.

Step 7: Configure Auto Start (Optional)

Ensure the web server starts on reboot:

sudo systemctl enable httpd # Apache

sudo systemctl enable nginx # Nginx

Conclusion

You have successfully deployed an EC2-based web server running **Apache or Nginx**. This setup is commonly used for hosting websites, web applications, or acting as a reverse proxy.

Project 4: Scalable Web App with ALB & Auto Scaling – Use EC2, ALB, and Auto Scaling for high availability.

This project involves deploying a **highly available** web application using **Amazon EC2, Application Load Balancer (ALB), and Auto Scaling**. The goal is to ensure that the web app can **scale automatically** based on traffic demand while maintaining availability.

Steps to Deploy a Scalable Web App

Step 1: Create a Launch Template

1. **Go to AWS Console → EC2 → Launch Templates → Create Launch Template**.
2. Configure:
    - **Name**: WebAppTemplate
    - **AMI**: Amazon Linux 2 or Ubuntu
    - **Instance Type**: t2.micro (Free Tier) or t3.medium
    - **Key Pair**: Select an existing key pair or create a new one.

User Data (Optional, for auto-installation of Apache/Nginx):

#!/bin/

sudo yum update -y

sudo yum install httpd -y

sudo systemctl start httpd

sudo systemctl enable httpd

echo "<h1>Welcome to Scalable Web App</h1>" | sudo tee /var/www/html/index.html

    - **Security Group**: Allow **SSH (22), HTTP (80), and HTTPS (443)**.

3. Click **Create Launch Template**.

Step 2: Create an Auto Scaling Group

1. **Go to EC2 → Auto Scaling Groups → Create Auto Scaling Group**.
2. **Choose Launch Template**: Select WebAppTemplate.
3. **Configure Group Size**:
   - **Desired Capacity**: 2
   - **Minimum Instances**: 1
   - **Maximum Instances**: 4
4. **Select Network**:
   - Choose an **existing VPC**.
   - Select at least **two subnets** across different **Availability Zones (AZs)**.
5. **Attach Load Balancer**:
   - Choose **Application Load Balancer (ALB)**.
   - **Create Target Group**:
     - Target Type: Instance
     - Protocol: HTTP
     - Health Check: /
   - Register instances later (Auto Scaling will handle this).
6. **Set Scaling Policies** (Optional):
   - Enable **Auto Scaling** based on CPU utilization.
   - Example Policy: Scale out when CPU > 60%, Scale in when CPU < 40%.
7. Click **Create Auto Scaling Group**.

Step 3: Create an Application Load Balancer (ALB)

1. **Go to EC2 → Load Balancers → Create Load Balancer**.
2. **Select Application Load Balancer**.
3. **Configure Basic Settings**:
   - **Name**: WebAppALB
   - **Scheme**: Internet-facing
   - **VPC**: Select the same VPC as Auto Scaling.
   - **Availability Zones**: Choose at least 2.
4. **Configure Listeners**:
   - Protocol: HTTP
   - Port: 80
5. **Target Group**:
   - **Select existing Target Group (from Auto Scaling Group)**.
6. **Security Group**:
   - Allow **HTTP (80)**.
7. Click **Create Load Balancer**.

Step 4: Test the Setup

1. **Get ALB DNS Name**:
   - Go to **EC2 → Load Balancers** → Copy the **ALB DNS Name**.

Open a browser and enter:
http://your-alb-dns-name

You should see:
Welcome to Scalable Web App

Step 5: Verify Auto Scaling

1. **Increase Load to Trigger Scaling**:
   - Use a tool like **Apache Benchmark** or manually increase traffic.

Run:
ab -n 1000 -c 50 http://your-alb-dns-name/

   - Check **EC2 → Auto Scaling Group** to see if new instances launch.

2. **Stop an Instance**:
   - Manually stop an instance to verify if **Auto Scaling** replaces it.

Conclusion

This setup ensures: ✅ **High Availability** using **ALB**
✅ **Scalability** using **Auto Scaling**
✅ **Redundancy** across **multiple AZs**

This architecture is commonly used for **production applications** that need reliability and cost optimization.

Project 5: Scalable Web App with NLB & Auto Scaling – Use EC2, ALB, and Auto Scaling for high availability.

Overview:
In this project, we will deploy a scalable and highly available web application using **EC2 instances**, **Network Load Balancer (NLB)**, and **Auto Scaling**. This setup ensures that our web application can handle varying traffic loads efficiently while maintaining availability and reliability.

Steps to Implement the Project

1. Set Up EC2 Instances

- Launch multiple EC2 instances with a web server installed (e.g., Apache, Nginx, or a custom application).
- Configure the security group to allow HTTP (port 80) and necessary ports for application communication.
- Ensure the web server is properly configured to serve requests.

2. Create a Network Load Balancer (NLB)

- Navigate to **EC2 > Load Balancers > Create Load Balancer**.
- Select **Network Load Balancer** as the type.
- Choose the **public subnets** where your EC2 instances are running.
- Create a **target group** and register the EC2 instances.
- Configure the **listener** on port 80 (HTTP) and forward traffic to the target group.

3. Configure Auto Scaling Group

- Create an **Auto Scaling Group** with a **Launch Template** or **Launch Configuration** that defines:
  - EC2 instance type, AMI, security group, and key pair.
  - User data script to install the web server upon launch.
- Set the desired **minimum, maximum, and desired capacity** for instances.
- Define **scaling policies** to increase/decrease instances based on CPU utilization or request load.

4. Enable Health Checks & Monitoring

- Configure **NLB health checks** to monitor instance health.
- Enable **Auto Scaling health checks** to replace unhealthy instances automatically.
- Use **Amazon CloudWatch** to track metrics like request count, CPU utilization, and instance activity.

5. Test the Setup

- Obtain the **NLB DNS name** and test the application in a browser.
- Simulate traffic using **Apache Benchmark (ab) or Siege** to verify Auto Scaling behavior.
- Monitor the Auto Scaling Group to ensure new instances launch and terminate as needed.

6. Optimize & Secure the Architecture

- **Enable HTTPS** using an AWS Certificate Manager (ACM) SSL certificate.
- **Implement IAM roles** for secure EC2 access.
- **Use AWS WAF** to protect against web threats.
- **Enable CloudWatch alarms** for proactive monitoring.

Outcome:

- A fully scalable, high-availability web application with **automatic load balancing and scaling**.
- Improved **performance, fault tolerance, and cost optimization** through Auto Scaling.
- **Seamless user experience** even under varying traffic conditions.

Project 6: Multi-Tier Web App Deployment – Deploy an application with VPC, ALB, EC2, and RDS.

A **Multi-Tier Web Application** architecture enhances scalability, security, and maintainability by separating concerns into multiple layers. This project deploys a web application on AWS using a **Virtual Private Cloud (VPC)**, **Application Load Balancer (ALB)**, **EC2 instances** for the web and application layers, and **RDS (Relational Database Service)** for the database layer.

Complete Steps for Deployment

Step 1: Set Up the VPC

1. Go to the **AWS Management Console → VPC**.
2. Create a **new VPC** (e.g., 10.0.0.0/16).
3. Create **two subnets**:
    - **Public Subnet (for ALB & Bastion Host)** (e.g., 10.0.1.0/24).
    - **Private Subnet (for EC2 App & RDS)** (e.g., 10.0.2.0/24).
4. Create an **Internet Gateway (IGW)** and attach it to the VPC.
5. Set up **Route Tables**:
    - Public subnet → Route to **IGW**.
    - Private subnet → Route to **NAT Gateway** (for internet access from private instances).

Step 2: Launch EC2 Instances

1. Go to **EC2 Dashboard → Launch Instance**.
2. Create a **Bastion Host** in the **public subnet** for SSH access.
3. Create **Web Server EC2 instances** in the **private subnet**.

Install required software on EC2 instances:
sudo yum update -y

sudo yum install -y httpd mysql

sudo systemctl start httpd

sudo systemctl enable httpd

4. Repeat for multiple instances if needed for load balancing.

Step 3: Set Up Application Load Balancer (ALB)

1. Go to **EC2 Dashboard → Load Balancer → Create Load Balancer**.
2. Choose **Application Load Balancer**.
3. Attach it to the **public subnet**.
4. Create a **target group** and register **EC2 instances** (web/app tier).
5. Configure health checks (/health endpoint).
6. Update **Security Groups** to allow HTTP/HTTPS.

Step 4: Deploy the Database (RDS)

1. Go to **RDS → Create Database**.
2. Select **MySQL/PostgreSQL**.
3. Choose **Multi-AZ Deployment** (for high availability).
4. Set up database credentials.
5. Place RDS in the **private subnet**.
6. Update security groups to allow only **EC2 instances** to connect.

Step 5: Configure Security Groups

- **ALB Security Group** → Allow HTTP (80) and HTTPS (443) from the internet.
- **Web/App EC2 Security Group** → Allow HTTP (80) only from the ALB.
- **Database Security Group** → Allow MySQL/PostgreSQL (3306) only from the Web/App layer.

Step 6: Deploy the Web Application

Upload application code to EC2:
sudo yum install -y git

git clone <your-repository>

cd <your-app>

1. Configure the database connection in your app.
2. Start the application.

Step 7: Testing & Scaling

1. **Test the Application**
    - Get the **ALB DNS name** and access the web app via a browser.
2. **Enable Auto Scaling**
    - Go to **EC2 Auto Scaling Groups**.
    - Configure a **launch template** for EC2 instances.
    - Set up scaling policies (e.g., scale up on high CPU usage).

Conclusion

This setup provides a **scalable, highly available, and secure** multi-tier web application architecture on AWS. It ensures redundancy using **ALB, Auto Scaling, and RDS Multi-AZ**, making it resilient to failures.

---

2. Serverless & Modern Web Hosting

Project 1: Serverless Static Website – Use S3, CloudFront, and Lambda@Edge for global performance.

A serverless static website on AWS eliminates the need for traditional web servers by using **Amazon S3** for hosting static content, **CloudFront** for global content delivery, and **Lambda@Edge** for dynamic content processing. This setup provides **high availability, low latency, and cost efficiency**.

Steps to Deploy a Serverless Static Website on AWS

Step 1: Create an S3 Bucket for Website Hosting

1. Open the **AWS Management Console** → Navigate to **S3**.
2. Click **Create bucket**, enter a **unique name**, and choose the **region**.
3. Uncheck **"Block all public access"** (if you want a public website).
4. Upload your **static website files** (HTML, CSS, JS, images).
5. Enable **Static website hosting** under **Properties**:
    - Choose **"Enable"**.
    - Set **index.html** as the default document.
    - Copy the **Bucket Website URL**.

Step 2: Configure Bucket Policy for Public Access

1. Go to **Permissions** → Click **Bucket Policy**.

Add the following policy to make the website publicly accessible:
json

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Principal": "*",
   "Action": "s3:GetObject",
   "Resource": "arn:aws:s3:::your-bucket-name/*"
  }
 ]
}
```

2. Save the policy.

Step 3: Create a CloudFront Distribution

1. Go to **CloudFront** in AWS Console.
2. Click **Create Distribution**.
3. Under **Origin**, select your **S3 bucket (not the website endpoint, but the bucket itself)**.
4. Enable **"Redirect HTTP to HTTPS"**.
5. Set **Caching Policy** to **CachingOptimized** (for better performance).
6. Click **Create** and note the **CloudFront URL**.

Step 4: (Optional) Use Lambda@Edge for Dynamic Processing

If you need to modify responses dynamically (e.g., adding security headers, URL rewrites), create a **Lambda@Edge function**.

1. Go to **Lambda** → Click **Create Function**.
2. Choose **Author from Scratch** → Select **Node.js or Python**.
3. Enable **Lambda@Edge** (set deployment to **us-east-1**).

Add this sample script to modify HTTP headers:
javascript

```
exports.handler = async (event) => {
   const response = event.Records[0].cf.response;
   response.headers['x-custom-header'] = [{ key: 'X-Custom-Header', value: 'MyServerlessSite' }];
   return response;
};
```

4. Deploy and associate it with CloudFront behavior.

Step 5: (Optional) Use a Custom Domain with Route 53

1. Register a **domain** using **Route 53** (or use an existing domain).
2. Create a **CNAME record** pointing to your CloudFront distribution.
3. Use **AWS Certificate Manager (ACM)** to enable **HTTPS with SSL**.

Conclusion

With this setup, your **static website is hosted on S3**, served globally using **CloudFront**, and can be enhanced with **Lambda@Edge** for additional processing. This architecture ensures **scalability, security, and cost efficiency**.

Project 2: Multi-Tenant SaaS Application – Use Cognito, API Gateway, and RDS for a multi-tenant SaaS solution.

A **Multi-Tenant SaaS Application** allows multiple customers (tenants) to use the same application while keeping their data secure and isolated. AWS provides a scalable solution using **Cognito for authentication, API Gateway for managing API requests, and RDS for structured data storage**. This setup ensures security, performance, and scalability for your SaaS platform.

Steps to Deploy a Multi-Tenant SaaS Application on AWS

1. Set Up AWS Cognito for User Authentication

- Go to the **AWS Cognito** service in the AWS Console.
- Create a **User Pool** to manage authentication for tenants.
- Enable **Multi-Tenant Support** using custom attributes (e.g., tenant_id).
- Configure authentication providers (Google, Facebook, or SAML-based authentication if needed).
- Create an **App Client** to integrate with your application.

2. Create an RDS Database for Multi-Tenant Data Storage

- Go to **Amazon RDS** and create a new database (PostgreSQL or MySQL recommended).
- Choose an **instance type** based on the expected load.
- Configure a **multi-tenant database model**:
    - **Single database, shared schema** (add tenant_id column in tables).
    - **Single database, separate schema per tenant**.
    - **Database per tenant** (useful for large enterprises).

3. Build APIs Using AWS API Gateway and Lambda

- Go to **Amazon API Gateway** and create a new **REST API**.
- Define API routes for different functionalities (e.g., /users, /orders).
- Use **Cognito Authorizer** to secure API access.
- Deploy Lambda functions to handle requests:
    - Create **Lambda functions** to interact with the RDS database.
    - Implement **tenant isolation** logic (ensure each tenant can access only
    - atus

```
  }

}
```

Use the listTasks query to retrieve all tasks:

graphql

```
query {
  listTasks {
    id
    title
    description
    status
  }
}
```

Step 6: Secure the API (Optional)

- Use **AWS Cognito** for user authentication.
- Enable **IAM** authentication for secure access.

Conclusion

You have successfully deployed a **GraphQL API using AWS AppSync** and **DynamoDB**. This setup allows seamless CRUD operations with real-time updates, making it an efficient backend solution for modern applications.

Project 3: Progressive Web App (PWA) Hosting – Deploy a React or Vue.js PWA on AWS Amplify.

Introduction

A Progressive Web App (PWA) combines the best of web and mobile applications by offering fast load times, offline support, and a native-like experience. AWS Amplify provides an easy way to deploy and host PWAs built with React or Vue.js, with automatic CI/CD integration.

Steps to Deploy a React or Vue.js PWA on AWS Amplify

Step 1: Prerequisites

- AWS account
- Node.js installed (check with node -v)
- GitHub, GitLab, or Bitbucket repository with a React or Vue.js PWA

Step 2: Create a React or Vue.js PWA (Optional)

For React:

sh

```
npx create-react-app my-pwa --template cra-template-pwa
cd my-pwa
npm start
```

For Vue.js:

sh

```
npm create vue@latest my-pwa
cd my-pwa
npm install
npm run dev
```

Step 3: Push the Project to GitHub

sh

```
git init
```

git add .

git commit -m "Initial PWA commit"

git branch -M main

git remote add origin <your-repository-url>

git push -u origin main

Step 4: Set Up AWS Amplify for Hosting

1. **Go to AWS Amplify Console**
   - Navigate to AWS Amplify Console.
   - Click **"Get Started"** under **"Deploy"**.
2. **Connect to Your Git Repository**
   - Select **GitHub/GitLab/Bitbucket**.
   - Authenticate and choose your repository.
   - Select the branch (e.g., main).
3. **Configure Build Settings**
   - AWS Amplify will detect the framework (React/Vue).
   - Modify amplify.yml if needed for custom build settings.

Step 5: Deploy the PWA

- Click **"Save and Deploy"**.
- AWS Amplify will build and deploy the PWA.
- Once deployment is complete, you will get a hosted URL (e.g., https://your-app.amplifyapp.com).

Step 6: Enable PWA Features in AWS Amplify

1. **Ensure the Web App Has a Manifest File**
   - React PWA template includes public/manifest.json.
   - Vue.js PWAs should have a manifest.json file in public/.
2. **Check Service Worker Configuration**
   - React PWA uses src/service-worker.js.
   - Vue.js PWAs can use Workbox for service workers.
3. **Set Up HTTPS (Mandatory for PWAs)**
   - AWS Amplify automatically enables HTTPS.
   - You can also connect a custom domain.

Step 7: Verify and Test the PWA

- Open the hosted URL in **Chrome**.
- Use **Lighthouse** in DevTools (Ctrl + Shift + I → "Lighthouse" tab → Generate Report).
- Check if the app is installable and supports offline usage.

Conclusion

You have successfully deployed a React or Vue.js PWA on AWS Amplify! This setup ensures seamless CI/CD, HTTPS, and scalability for your PWA

Project 4: Jamstack Site with AWS – Use AWS Amplify, Headless CMS, and S3 for a Jamstack website.

Build a modern, fast, and scalable Jamstack website using AWS Amplify, a Headless CMS, and S3 for static hosting. This approach decouples the frontend from the backend, improving performance, security, and scalability.

Steps to Build the Jamstack Site with AWS

Step 1: Set Up AWS Amplify

1. Sign in to the AWS Management Console.
2. Navigate to **AWS Amplify** and create a new project.
3. Connect a GitHub repository containing your frontend code (React, Vue, or Next.js).
4. Configure Amplify for continuous deployment.

Step 2: Choose a Headless CMS

1. Select a Headless CMS like **Strapi, Contentful, or Sanity**.
2. Create a new project and define content models (e.g., blog posts, pages).
3. Publish and retrieve content via the CMS API.

Step 3: Configure AWS S3 for Static Hosting

1. Open the **AWS S3 Console** and create a new bucket.
2. Enable **static website hosting** in the bucket settings.
3. Upload your pre-built static files if using S3 as an alternative deployment option.

Step 4: Connect CMS with Frontend

1. Fetch data from the CMS using **REST or GraphQL APIs**.
2. Display dynamic content on your website using API calls.

Step 5: Deploy and Test the Website

1. Deploy the frontend via AWS Amplify or S3.
2. Enable **CDN via AWS CloudFront** for fast global access.
3. Test your website and verify performance.

Step 6: Set Up Custom Domain and SSL

1. Configure a custom domain in AWS Route 53.
2. Use **AWS Certificate Manager (ACM)** to enable HTTPS.

Conclusion

This project helps create a fast, secure, and scalable Jamstack website using AWS services. With AWS Amplify for deployment, a Headless CMS for content management, and S3 for storage, you ensure a seamless user experience

---

Project 5: AWS App Runner Deployment – Deploy a containerized web application with AWS App Runner.

Introduction:

AWS App Runner is a fully managed service that simplifies deploying containerized web applications without managing infrastructure. It automatically builds and scales applications from container images stored in Amazon ECR or directly from source code in GitHub. This project will guide you through deploying a containerized web application using AWS App Runner.

Complete Steps:

Step 1: Prerequisites

- AWS Account
- Docker installed on your local machine
- AWS CLI configured
- A containerized web application (e.g., a simple Node.js or Python Flask app)

Step 2: Create and Build a Docker Image

1. Write a simple web application (e.g., Node.js, Flask).

Create a Dockerfile in your project directory. Example for Node.js:
dockerfile

FROM node:18

```
WORKDIR /app

COPY . .

RUN npm install

CMD ["node", "app.js"]

EXPOSE 8080
```

2.

Build the Docker image:
```
sh

docker build -t my-app-runner-app .
```

3.

Step 3: Push Image to Amazon Elastic Container Registry (ECR)

Create an ECR repository:
```
sh

aws ecr create-repository --repository-name my-app-runner-app
```

1.

Authenticate Docker with ECR:
```
sh

aws ecr get-login-password --region us-east-1 | docker login --username AWS
--password-stdin <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com
```

2.

Tag the image:
```
sh

docker tag my-app-runner-app:latest
<AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/my-app-runner-app:lat
est
```

3.

Push the image to ECR:
```
sh

docker push
<AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/my-app-runner-app:lat
est
```

4.

Step 4: Deploy to AWS App Runner

1. Go to **AWS Console > App Runner**.
2. Click **Create an App Runner Service**.
3. Select **Container Registry** and choose **Amazon ECR**.
4. Select the repository and image.
5. Configure the service:
    - Set a service name (e.g., my-app-runner-service).
    - Set port to 8080 (or as per your application).
    - Select **Auto Scaling** (default settings).
6. Click **Deploy** and wait for App Runner to set up the service.

Step 5: Access the Application

- Once deployment is complete, AWS App Runner provides a public URL.
- Open the URL in a browser to verify the application is running.

Step 6: Monitor & Update the Service

- Use AWS App Runner logs and metrics to monitor your service.
- Update your image in ECR and redeploy for changes.

Conclusion

AWS App Runner makes it easy to deploy and scale containerized applications without managing infrastructure. It's ideal for developers who want a serverless experience for their web apps.

Project 6: Serverless API with Lambda & API Gateway – Build a RESTful API using Lambda.

This project involves building a **RESTful API** using **AWS Lambda** and **API Gateway**. It eliminates the need for managing servers, providing a scalable and cost-effective solution. Lambda handles the backend logic, while API Gateway routes requests and manages security.

Steps to Build the Serverless API

Step 1: Create an AWS Lambda Function

- Go to the **AWS Lambda** console.
- Click **Create Function** → Select **Author from Scratch**.
- Enter a function name (e.g., ServerlessAPI).
- Choose **Runtime** as Python 3.x or Node.js 18.x.
- Select **Create a new role with basic Lambda permissions**.
- Click **Create Function**.

In the function editor, add this sample code:
Python Example (app.py)
python

```
import json


def lambda_handler(event, context):
    return {
        "statusCode": 200,
        "body": json.dumps({"message": "Hello from Serverless API!"})
    }
```

-
- Click **Deploy**.

Step 2: Create an API in API Gateway

- Go to **AWS API Gateway** → Click **Create API**.
- Choose **REST API** → Click **Build**.
- Select **Regional** and click **Create API**.
- Click **Actions** → **Create Resource**.
- Enter a name (e.g., hello) and click **Create Resource**.
- Select the resource and click **Create Method** → Choose **GET**.
- Select **Lambda Function** as the integration type.
- Enter the **Lambda function name** created earlier.
- Click **Save** → Confirm by clicking **OK**.

Step 3: Deploy the API

- Click **Actions** → **Deploy API**.
- Create a new stage (e.g., prod).
- Click **Deploy**.
- Copy the **Invoke URL** (e.g., https://xyz.execute-api.region.amazonaws.com/prod/hello).

Step 4: Test the API

Open a browser or use **cURL**/Postman:
sh

curl -X GET https://xyz.execute-api.region.amazonaws.com/prod/hello

-

You should receive the JSON response:
json

{"message": "Hello from Serverless API!"}

-

Enhancements (Optional)

- Use **DynamoDB** for persistent storage.
- Implement **JWT authentication** with Amazon Cognito.
- Enable **CORS** for cross-origin access.

This setup provides a **scalable, serverless API** without provisioning servers.

Project 7: GraphQL API with AWS AppSync – Deploy a GraphQL backend using DynamoDB.

Introduction

AWS AppSync is a fully managed service that simplifies the development of GraphQL APIs. It allows real-time data access and offline support. In this project, you will deploy a GraphQL API using AWS AppSync, with DynamoDB as the backend database. This setup provides a scalable, serverless, and efficient way to manage APIs with flexible querying and data fetching.

Complete Steps to Build the Project

Step 1: Set Up an AWS Account

- Sign in to your AWS account.
- Navigate to the **AWS AppSync** service in the AWS Management Console.

Step 2: Create an AppSync API

- Click **Create API** → Choose **"Build from scratch"** → Enter API name → Select **"Create"**.
- Choose **GraphQL** as the API type.

Step 3: Define a GraphQL Schema

- Go to the **Schema** section and define your GraphQL types.

Example schema for a To-Do app:
graphql

```
type Todo {
  id: ID!
  title: String!
  completed: Boolean!
}

type Query {
  getTodos: [Todo]
}
```

```
type Mutation {
  addTodo(title: String!): Todo
  updateTodo(id: ID!, completed: Boolean!): Todo
  deleteTodo(id: ID!): ID
}

schema {
  query: Query
  mutation: Mutation
}
```

-

Step 4: Set Up a DynamoDB Table

- Navigate to **AWS DynamoDB → Create Table**
- Table name: Todos
- Partition key: id (String)
- Enable **on-demand capacity mode**

Step 5: Connect DynamoDB as a Data Source

- In AWS AppSync, go to **Data Sources → Create Data Source**.
- Select **Amazon DynamoDB →** Choose the Todos table.
- Grant necessary permissions by attaching an IAM role.

Step 6: Create Resolvers for Queries & Mutations

- In AppSync, go to **Resolvers** and attach the created DynamoDB data source to GraphQL queries/mutations.
- Use VTL (Velocity Template Language) for mapping templates.

Example **createTodo** resolver request template:
vtl

```
{
  "version": "2018-05-29",
  "operation": "PutItem",
  "key": {
    "id": { "S": "$util.autoId()" }
  },
  "attributeValues": {
    "title": { "S": "$context.arguments.title" },
    "completed": { "BOOL": false }
  }
}
```

-

Step 7: Deploy & Test the API

- Use **GraphQL Explorer** in AWS AppSync to test queries and mutations.

Example mutation to add a new todo:
graphql

```
mutation {
  addTodo(title: "Learn AppSync") {
    id
```

```
  title
  completed
 }
}
```

-

Example query to fetch all todos:
graphql

```
query {
 getTodos {
  id
  title
  completed
 }
}
```

-

Step 8: Secure the API

- In **Settings**, configure authentication (API Key, Cognito, or IAM).
- Use **AWS IAM roles** for granular access control.

Step 9: Deploy a Frontend (Optional)

- Use **React.js, Vue.js, or AWS Amplify** to build a UI.
- Install AWS Amplify CLI and connect the frontend to AppSync.

Conclusion

This project provides hands-on experience with AWS AppSync, GraphQL, and DynamoDB, enabling you to build a serverless GraphQL backend efficiently. You can extend it by adding authentication, subscriptions for real-time updates, and integrating a frontend.

Project 8: Serverless Chatbot – Build an AI-powered chatbot with AWS Lex & Lambda.

Introduction

A serverless chatbot allows users to interact using natural language. AWS Lex provides automatic speech recognition (ASR) and natural language understanding (NLU) to process user queries. AWS Lambda processes user inputs and responds dynamically. This chatbot can be integrated into applications like websites or messaging platforms.

Steps to Build a Serverless Chatbot

Step 1: Create an AWS Lex Bot

1. Go to the **AWS Lex** console.
2. Click **Create bot** → Choose **Create from scratch**.
3. Enter the bot name (e.g., "CustomerSupportBot").
4. Select a language (e.g., English).
5. Configure IAM permissions for Lex to access Lambda.

Step 2: Define Intents and Utterances

1. Create an **Intent** (e.g., "CheckOrderStatus").

2. Add **Utterances** (e.g., "Where is my order?", "Track my order").
3. Set up **Slots** (e.g., "OrderID" for order tracking).
4. Configure the **Fulfillment** to call a Lambda function.

Step 3: Create an AWS Lambda Function

1. Go to the **AWS Lambda** console → Click **Create function**.
2. Select **Author from scratch** → Provide a function name (e.g., "LexOrderHandler").
3. Choose **Runtime** as Python or Node.js.
4. Add Lex permissions to Lambda (via IAM Role).
5. Write Lambda code to handle chatbot responses and logic.
6. Deploy the function.

Step 4: Integrate Lambda with AWS Lex

1. Go to your **Lex bot** → Select the Intent.
2. Under **Fulfillment**, choose **AWS Lambda Function**.
3. Select the Lambda function you created.
4. Save and build the bot.

Step 5: Test the Chatbot

1. In the Lex console, click **Test Bot**.
2. Enter sample queries and verify responses.
3. Adjust responses and logic as needed.

Step 6: Deploy and Integrate

1. Deploy the chatbot on **Amazon Connect, Facebook Messenger, Slack, or a Website**.
2. Use **AWS API Gateway** and **AWS Lambda** for a REST API interface.
3. Monitor chatbot performance using **AWS CloudWatch Logs**.

Conclusion

This project demonstrates how to build a serverless AI chatbot using AWS Lex and Lambda. It provides an interactive experience for users and can be customized for various use cases like customer support, FAQs, or order tracking.

Project 9: Event-Driven Microservices – Use SNS & SQS for asynchronous microservices communication.

Introduction

Event-driven microservices architecture enables loosely coupled services to communicate asynchronously. AWS **Simple Notification Service (SNS)** and **Simple Queue Service (SQS)** help in achieving this by allowing microservices to publish and consume messages efficiently.

In this project, we will:

- Use **SNS** for event broadcasting.
- Use **SQS** for message queuing and processing.
- Set up multiple microservices to communicate asynchronously.

Complete Steps to Implement This AWS Project

Step 1: Create an SNS Topic

1. Open **AWS Management Console** → Navigate to **SNS**.
2. Click on **Create Topic** → Choose **Standard** or **FIFO**.
3. Give a **name** (e.g., order-events-topic).
4. Click **Create Topic**.

Step 2: Create SQS Queues

1. Open **AWS SQS** → Click **Create Queue**.
2. Choose **Standard** or **FIFO** queue type.
3. Give a **name** (e.g., order-processing-queue).
4. Enable **"Subscribe to SNS topic"**.
5. Under **Access Policy**, allow SNS to send messages to SQS.
6. Click **Create Queue**.

Repeat for additional microservices (e.g., inventory-update-queue).

Step 3: Subscribe SQS Queues to the SNS Topic

1. Go to the **SNS Topic** created earlier.
2. Click **Create Subscription**.
3. Select **SQS** as the protocol.
4. Choose the **SQS Queue** created earlier.
5. Click **Create Subscription**.

Repeat this for other SQS queues.

Step 4: Deploy Producer Microservice (Publisher)

The **producer microservice** will publish events to the SNS topic.

Sample Python Code (Using Boto3)

python

```
import boto3
import json

sns = boto3.client('sns', region_name='us-east-1')
topic_arn = 'arn:aws:sns:us-east-1:123456789012:order-events-topic'

message = {
    "order_id": "12345",
    "status": "created"
}

response = sns.publish(
    TopicArn=topic_arn,
    Message=json.dumps(message),
    Subject="New Order Event"
)
print("Message published:", response)
```

Step 5: Deploy Consumer Microservice (Subscriber)

The **consumer microservice** will process messages from the SQS queue.

Sample Python Code (Using Boto3)

python

```
import boto3
```

```
sqs = boto3.client('sqs', region_name='us-east-1')


queue_url =
'https://sqs.us-east-1.amazonaws.com/123456789012/order-processing-queue'


def process_messages():
    while True:
        messages = sqs.receive_message(
            QueueUrl=queue_url,
            MaxNumberOfMessages=1,
            WaitTimeSeconds=10
        )

        if 'Messages' in messages:
            for msg in messages['Messages']:
                print("Processing order:", msg['Body'])
                sqs.delete_message(QueueUrl=queue_url,
ReceiptHandle=msg['ReceiptHandle'])


process_messages()
```

Step 6: Deploy and Test the Architecture

- **Publish an event** using the producer microservice.
- **Check SQS queues** to verify that messages are received.
- **Run the consumer service** to process messages from the queue.

Conclusion

This project demonstrates an event-driven microservices pattern using **AWS SNS & SQS**. It enables asynchronous communication between microservices, improving scalability and decoupling.

Project 10: URL Shortener – Implement a Lambda-based URL shortener with DynamoDB.

Creating a serverless URL shortener using AWS services like Lambda and DynamoDB enables efficient and scalable management of URL mappings without the need to manage underlying server infrastructure. This approach leverages AWS's fully managed services to handle the backend operations seamlessly.

Step-by-Step Implementation:

1. **Set Up a DynamoDB Table:**
    - Navigate to the DynamoDB service in the AWS Management Console.
    - Create a new table named URLShortener with a primary key short_code of type String.

2. **Develop the Lambda Function:**
    - In the AWS Lambda console, create a new function named URLShortenerFunction using Python 3.x as the runtime.
    - Assign a role to the function that grants it permission to interact with DynamoDB.

Implement the function to handle both the creation of short URLs and the redirection logic.
Sample Code:

python

```
import json
import boto3
import string
import random

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('URLShortener')

def generate_short_code():
    chars = string.ascii_letters + string.digits
    return ''.join(random.choice(chars) for _ in range(6))

def lambda_handler(event, context):
    http_method = event['httpMethod']

    if http_method == 'POST':
        # Create a short URL
        body = json.loads(event['body'])
        long_url = body['long_url']
        short_code = generate_short_code()
        table.put_item(Item={
            'short_code': short_code,
            'long_url': long_url
        })

        return {
            'statusCode': 200,
            'body': json.dumps({'short_code': short_code})
        }
    elif http_method == 'GET':
        # Redirect to the original URL
        params = event['queryStringParameters']
        short_code = params['short_code']

        response = table.get_item(Key={'short_code': short_code})
        if 'Item' in response:
            long_url = response['Item']['long_url']
            return {
                'statusCode': 301,
                'headers': {'Location': long_url}
```

```
    }
else:
    return {
        'statusCode': 404,
        'body': json.dumps({'error': 'Short URL not found'})
    }
```

3. **Configure API Gateway:**
    - Set up a new REST API in API Gateway.
    - Create two endpoints:
        - POST /shorten to handle URL shortening requests.
        - GET /redirect to manage redirection based on the short code.
    - Integrate these endpoints with the URLShortenerFunction Lambda function.
    - Deploy the API to a stage (e.g., prod) to make it accessible.

4. **Test the Application:**
    - Use tools like Postman to send a POST request to the /shorten endpoint with a JSON body containing the long_url.
    - Receive the generated short_code in the response.
    - Access the original URL by sending a GET request to the /redirect endpoint with the short_code as a query parameter.

5. **Clean Up Resources:**
    - After testing, ensure you delete the DynamoDB table, Lambda function, and API Gateway to prevent unnecessary charges.

For a comprehensive walkthrough and additional insights, refer to the detailed tutorial on building a serverless URL shortener using AWS services.

By following these steps, you can create a scalable and cost-effective URL shortening service using AWS's serverless architecture.

Project 11: Serverless Expense Tracker – Use Lambda, DynamoDB, and API Gateway to build an expense management app.

A Serverless Expense Tracker is an AWS-based application that allows users to record and manage their expenses without managing servers. This project uses AWS Lambda for backend logic, Amazon DynamoDB as the NoSQL database, and Amazon API Gateway to expose RESTful APIs. The app provides CRUD (Create, Read, Update, Delete) operations for managing expenses.

By going serverless, you eliminate infrastructure management, reduce costs, and improve scalability.

Steps to Build a Serverless Expense Tracker

1. Create a DynamoDB Table

Go to the AWS Management Console → DynamoDB → Create Table.

Set Table Name: Expenses

Set Primary Key: expense_id (String)

Enable On-Demand Capacity Mode for cost efficiency.

2. Create an AWS Lambda Function

Go to AWS Lambda → Create Function → Author from Scratch.

Set Function Name: ExpenseHandler

Choose Runtime: Python 3.x or Node.js 18.x

Set Execution Role: Create a new role with DynamoDB Full Access.

3. Write Lambda Code for CRUD Operations

Use Boto3 (Python) or AWS SDK (Node.js) to interact with DynamoDB.

Implement the following functions:

add_expense(): Add a new expense.

get_expense(): Retrieve a specific expense.

update_expense(): Modify an expense.

delete_expense(): Remove an expense.

Example (Python – Boto3):

```
import json
import boto3
import uuid

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Expenses')

def lambda_handler(event, context):
    if event['httpMethod'] == 'POST':  # Create Expense
        data = json.loads(event['body'])
        expense_id = str(uuid.uuid4())
        table.put_item(Item={"expense_id": expense_id, **data})
        return {"statusCode": 201, "body": json.dumps({"message": "Expense added", "id": expense_id})}

    if event['httpMethod'] == 'GET':  # Get Expense
        expense_id = event['queryStringParameters']['expense_id']
        response = table.get_item(Key={"expense_id": expense_id})
        return {"statusCode": 200, "body": json.dumps(response.get("Item", {}))}

    if event['httpMethod'] == 'PUT':  # Update Expense
        data = json.loads(event['body'])
        expense_id = data['expense_id']
        table.update_item(
            Key={"expense_id": expense_id},
            UpdateExpression="SET amount=:a, category=:c, date=:d",
            ExpressionAttributeValues={":a": data['amount'], ":c": data['category'], ":d": data['date']}
        )
        return {"statusCode": 200, "body": json.dumps({"message": "Expense updated"})}
```

```
    if event['httpMethod'] == 'DELETE':  # Delete Expense
        expense_id = event['queryStringParameters']['expense_id']
        table.delete_item(Key={"expense_id": expense_id})
        return {"statusCode": 200, "body": json.dumps({"message": "Expense deleted"})}
```

4. Deploy the Lambda Function

Click Deploy to save changes.

Test the function with sample JSON inputs.

5. Set Up API Gateway

Go to API Gateway → Create API → HTTP API.

Define endpoints:

POST /expense → Triggers add_expense

GET /expense → Triggers get_expense

PUT /expense → Triggers update_expense

DELETE /expense → Triggers delete_expense

Deploy the API and note the Invoke URL.

6. Test the API

Use Postman or cURL:

```
curl -X POST https://your-api-id.execute-api.us-east-1.amazonaws.com/expense \
    -H "Content-Type: application/json" \
    -d '{"amount": 50, "category": "Food", "date": "2025-03-05"}'
```

7. Frontend (Optional)

Use React.js or Vue.js to build a frontend.

Fetch data from the API Gateway endpoints.

8. Monitor and Optimize

Use CloudWatch Logs to track errors.

Optimize with DynamoDB Indexing and Lambda Memory Settings.

Conclusion

This project provides a cost-effective, scalable expense tracker using AWS Serverless technologies. You can extend it by adding authentication (Cognito), analytics (AWS QuickSight), or deploying a frontend (React.js).

Project 12: Serverless Image Resizer – Automatically resize images uploaded to an S3 bucket using Lambda.

Implementing a serverless image resizing service using AWS involves automatically resizing images uploaded to an Amazon S3 bucket by leveraging AWS Lambda. This approach ensures scalability, cost-effectiveness, and eliminates the need for server management.

Overview:

When an image is uploaded to a designated S3 bucket, it triggers a Lambda function. This function processes the image—resizing it as specified—and then stores the resized image back into an S3 bucket.

Steps to Implement:

1. **Create S3 Buckets:**
   - Set up two S3 buckets: one to serve as the source for original images and another to store the resized images.
2. **Set Up IAM Role and Policy:**
   - In AWS Identity and Access Management (IAM), create a policy that grants necessary permissions for S3 operations and logging.
   - Create an IAM role and attach the created policy to it. This role will be assumed by the Lambda function to access S3 buckets and other necessary services.
3. **Develop the Lambda Function:**
   - Choose a runtime for your Lambda function, such as Node.js or Python.

Write the function code to handle image processing. For instance, using Node.js with the 'jimp' library:
javascript

```
const AWS = require('aws-sdk');

const Jimp = require('jimp');



const s3 = new AWS.S3();

const DEST_BUCKET = process.env.DEST_BUCKET;



exports.handler = async (event) => {

    const record = event.Records[0];

    const sourceBucket = record.s3.bucket.name;

    const objectKey = decodeURIComponent(record.s3.object.key.replace(/\+/g, ' '));



    try {

        // Retrieve the image from the source bucket

        const params = { Bucket: sourceBucket, Key: objectKey };

        const inputData = await s3.getObject(params).promise();



        // Read and resize the image

        const image = await Jimp.read(inputData.Body);

        const resizedImage = await image.resize(200, 200).getBufferAsync(Jimp.MIME_JPEG);



        // Define the destination bucket and object key

        const destParams = {

            Bucket: DEST_BUCKET,

            Key: objectKey,

            Body: resizedImage,

            ContentType: 'image/jpeg'

        };
```

```
    // Upload the resized image to the destination bucket
    await s3.putObject(destParams).promise();
    console.log(`Successfully resized and uploaded ${objectKey} to ${DEST_BUCKET}`);
  } catch (error) {
    console.error(`Error processing ${objectKey} from ${sourceBucket}:`, error);
    throw error;
  }
};
```

- In this script, the Lambda function retrieves the uploaded image from the source S3 bucket, resizes it to 200x200 pixels using the 'jimp' library, and uploads the resized image to the destination S3 bucket.

4. **Package and Deploy the Lambda Function:**
   - Install necessary dependencies locally and package them with your Lambda function code into a ZIP file.
   - Upload this ZIP file to AWS Lambda.
5. **Configure Environment Variables:**
   - Set environment variables in the Lambda function configuration, such as DEST_BUCKET, to specify the destination S3 bucket for resized images.
6. **Set Up S3 Event Trigger:**
   - Configure the source S3 bucket to trigger the Lambda function upon object creation events (e.g., when a new image is uploaded).
7. **Test the Setup:**
   - Upload an image to the source S3 bucket.
   - Verify that the Lambda function executes successfully and that the resized image appears in the destination S3 bucket.

By following these steps, you can establish a serverless architecture that automatically resizes images upon upload, optimizing storage and ensuring consistent image dimensions across your applications.

Project 13: Real-time Polling App – Use WebSockets with API Gateway and DynamoDB for a real-time polling system.

Introduction

This project involves building a real-time polling system using AWS WebSockets with API Gateway and DynamoDB. It allows users to vote on polls and see real-time updates without refreshing the page. WebSockets enable full-duplex communication between clients and the backend, making it ideal for live updates.

Steps to Build the Real-time Polling App

1. Set Up AWS API Gateway for WebSockets

- Open AWS Management Console → API Gateway.
- Create a **WebSocket API** with a **route selection expression** ($request.body.action).
- Add **routes**:
  - $connect → Handles new client connections.
  - $disconnect → Handles disconnections.
  - sendVote → Processes votes from users.
  - broadcastResults → Sends real-time poll updates to clients.
- Deploy the WebSocket API and note the **WebSocket URL**.

---

2. Create a DynamoDB Table for Polls

- Open AWS Console → DynamoDB → Create Table.
- Table name: **Polls**

- Primary Key: pollId (String)
- Add an attribute: votes (Map) to store vote counts.

---

3. Implement AWS Lambda Functions

- **Create Lambda functions to handle:**
  - **Connect & Disconnect:** Store active WebSocket connections in DynamoDB.
  - **Vote Handling:** Update poll results in DynamoDB.
  - **Broadcast Results:** Fetch poll data and send updates to connected clients.
- Attach API Gateway permissions to Lambda using AWS IAM Roles.

---

4. Configure API Gateway Integration

- Link API Gateway **routes** to corresponding **Lambda functions**.
- Use **IAM permissions** to allow API Gateway to invoke Lambda.

---

5. Deploy and Test the WebSocket API

- Deploy the API Gateway WebSocket API.
- Use **wscat** or a frontend app (React/Vue) to test connections and real-time updates.
- Send test votes and verify updates in real-time.

---

6. Deploy Frontend (Optional)

- Use React/Vue with WebSockets to connect to the API Gateway WebSocket URL.
- Display poll questions and update vote results in real-time.
- Deploy frontend on AWS Amplify or S3 + CloudFront.

Conclusion

This project demonstrates how to build a real-time polling app using WebSockets in API Gateway, Lambda, and DynamoDB. It ensures instant updates for users and scales efficiently on AWS.

Project 14: Serverless Sentiment Analysis – Analyze tweets using AWS Comprehend and store results in DynamoDB.

Introduction

This project involves building a serverless application to analyze the sentiment of tweets using AWS Comprehend. The analyzed sentiment data is then stored in DynamoDB for further analysis. This setup leverages AWS Lambda, API Gateway, and DynamoDB, making it fully serverless and scalable.

Steps to Implement the Project

Step 1: Set Up AWS Services

- Log in to your AWS account and navigate to the **AWS Management Console**.

Step 2: Create a Twitter Developer Account & API Key

- Sign up at Twitter Developer Portal.
- Create a new app to get API credentials for accessing tweets.

Step 3: Configure AWS Comprehend

- Navigate to **AWS Comprehend** in the AWS Console.
- No need for manual configuration, as we will call the sentiment analysis API through Lambda.

Step 4: Create an S3 Bucket (Optional)

- If you plan to store raw tweets, create an **S3 bucket** to store tweet data.

Step 5: Set Up a DynamoDB Table

- Go to **AWS DynamoDB** and create a new table:
  - **Table Name:** SentimentAnalysis
  - **Partition Key:** TweetID (String)
  - Add any other attributes as needed.

Step 6: Create an AWS Lambda Function

- Navigate to **AWS Lambda** and create a new function:
  - **Runtime:** Python 3.x
  - **Permissions:** Attach the AWSComprehendFullAccess and DynamoDBFullAccess policies.
  - Install dependencies (boto3 for AWS SDK, requests for API calls).
  - Use the function to:
    1. Fetch tweets from Twitter API.
    2. Pass tweets to AWS Comprehend for sentiment analysis.
    3. Store results in DynamoDB.

Example Lambda Function Code (Python)

python

```
import boto3
import json
import requests

# AWS Clients
comprehend = boto3.client('comprehend')
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('SentimentAnalysis')

# Function to fetch tweets (Replace with actual Twitter API integration)
def fetch_tweets():
    return ["I love AWS!", "This is frustrating.", "Excited about cloud computing!"]

# Lambda Handler
def lambda_handler(event, context):
    tweets = fetch_tweets()
    for tweet in tweets:
        response = comprehend.detect_sentiment(Text=tweet, LanguageCode='en')
        sentiment = response['Sentiment']

        # Store results in DynamoDB
        table.put_item(
            Item={
                'TweetID': str(hash(tweet)),
                'Tweet': tweet,
                'Sentiment': sentiment
            }
        )

    return {"message": "Sentiment analysis completed!"}
```

Step 7: Set Up API Gateway

- Create an **API Gateway** REST API.
- Create a **POST** method and link it to the Lambda function.
- Deploy the API and get the endpoint URL.

Step 8: Deploy & Test

- Trigger the API with a request.
- Check DynamoDB to confirm the sentiment data is stored.
- Monitor logs in AWS CloudWatch.

Conclusion

This serverless solution efficiently analyzes tweets using AWS Comprehend and stores the results in DynamoDB. It eliminates infrastructure management and provides a scalable approach to sentiment analysis.

---

3. Infrastructure as Code (IaC) & Automation

Project 1: Terraform for AWS – Deploy VPC, EC2, and RDS using Terraform.

Terraform is an open-source Infrastructure as Code (IaC) tool that enables you to define and provision infrastructure using a declarative configuration language. By utilizing Terraform, you can automate the deployment of AWS resources such as Virtual Private Clouds (VPCs), Elastic Compute Cloud (EC2) instances, and Relational Database Service (RDS) databases, ensuring consistent and repeatable infrastructure setups.

Below is a step-by-step guide to deploying a VPC, EC2 instance, and RDS database on AWS using Terraform:

1. Prerequisites:

- **AWS Account:** Ensure you have an active AWS account.
- **AWS CLI:** Install and configure the AWS Command Line Interface (CLI) with appropriate credentials.
- **Terraform:** Install Terraform on your local machine.

2. Set Up the Project Directory:

Create a new directory for your Terraform project and navigate into it:

mkdir terraform-aws-project

cd terraform-aws-project

3. Initialize Terraform:

Initialize the Terraform project to download necessary providers:

terraform init

4. Create a VPC:

Define a VPC resource in a main.tf file:

hcl

```
provider "aws" {
  region = "us-west-2"  # Specify your desired AWS region
}

resource "aws_vpc" "main_vpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "main_vpc"
  }
}
```

5. Create Subnets:

Define public and private subnets within the VPC:

hcl

```
resource "aws_subnet" "public_subnet" {
```

```
  vpc_id            = aws_vpc.main_vpc.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "us-west-2a"
  tags = {
    Name = "public_subnet"
  }
}

resource "aws_subnet" "private_subnet" {
  vpc_id            = aws_vpc.main_vpc.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-west-2a"
  tags = {
    Name = "private_subnet"
  }
}
```

6. Create an Internet Gateway and Route Tables:

Set up an Internet Gateway and associate it with the public subnet:

hcl

```
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main_vpc.id
  tags = {
    Name = "main_igw"
  }
}

resource "aws_route_table" "public_route_table" {
  vpc_id = aws_vpc.main_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
  tags = {
    Name = "public_route_table"
  }
}

resource "aws_route_table_association" "public_subnet_association" {
  subnet_id      = aws_subnet.public_subnet.id
  route_table_id = aws_route_table.public_route_table.id
```

```
}
```

7. Create Security Groups:

Define security groups for the EC2 instance and RDS database:

hcl

```
resource "aws_security_group" "ec2_sg" {
  vpc_id = aws_vpc.main_vpc.id
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]  # Allow SSH from anywhere; adjust as needed
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "ec2_sg"
  }
}

resource "aws_security_group" "rds_sg" {
  vpc_id = aws_vpc.main_vpc.id
  ingress {
    from_port   = 3306  # Default MySQL port; adjust for your DB engine
    to_port     = 3306
    protocol    = "tcp"
    cidr_blocks = ["10.0.0.0/16"]  # Allow access from within the VPC
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "rds_sg"
  }
}
```

8. Launch an EC2 Instance:

Configure an EC2 instance within the public subnet:

hcl

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"  # Amazon Linux 2 AMI; update as needed
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnet.id
  security_groups = [aws_security_group.ec2_sg.name]
  tags = {
    Name = "web_server"
  }
}
```

9. Deploy an RDS Instance:

Set up an RDS MySQL database in the private subnet:

hcl

```
resource "aws_db_subnet_group" "rds_subnet_group" {
  name       = "rds_subnet_group"
  subnet_ids = [aws_subnet.private_subnet.id]
  tags = {
    Name = "rds_subnet_group"
  }
}

resource "aws_db_instance" "rds_instance" {
  allocated_storage    = 20
  engine               = "mysql"
  engine_version       = "8.0"
  instance_class       = "db.t2.micro"
  name                 = "mydatabase"
  username             = "admin"
  password             = "password"  # Use a secure password
  db_subnet_group_name = aws_db_subnet_group.rds_subnet_group.name
  vpc_security_group_ids = [aws_security_group.rds_sg.id]
  skip_final_snapshot  = true
  tags = {
    Name = "rds_instance"
  }
}
```

****10. Apply the Terraform Configuration:**

Project 2: CloudFormation for Infrastructure as Code – Automate AWS resource provisioning.

CloudFormation for Infrastructure as Code – Automate AWS Resource Provisioning

Introduction

AWS CloudFormation is an Infrastructure as Code (IaC) service that allows you to define and provision AWS resources using declarative templates. With CloudFormation, you can automate the setup of AWS infrastructure, ensuring consistency, reducing manual effort, and enabling repeatable deployments.

This project will guide you through creating a CloudFormation stack to deploy a Virtual Private Cloud (VPC), an EC2 instance, and an RDS database.

Steps to Automate AWS Resource Provisioning Using CloudFormation

1. Prerequisites

- An **AWS account** with admin access.
- **AWS CLI** installed and configured (aws configure).
- **CloudFormation template** in YAML or JSON format.
- A text editor (VS Code, Sublime Text, or AWS CloudFormation Designer).

---

2. Write a CloudFormation Template

Create a cloudformation-template.yaml file defining the resources.

yaml

AWSTemplateFormatVersion: '2010-09-09'

Description: Create a VPC, EC2 instance, and RDS database.

Resources:

MyVPC:

Type: AWS::EC2::VPC

Properties:

CidrBlock: 10.0.0.0/16

EnableDnsSupport: true

EnableDnsHostnames: true

MySubnet:

Type: AWS::EC2::Subnet

Properties:

VpcId: !Ref MyVPC

CidrBlock: 10.0.1.0/24

MySecurityGroup:

Type: AWS::EC2::SecurityGroup

Properties:

```
    GroupDescription: Allow SSH and HTTP
    VpcId: !Ref MyVPC
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 80
        ToPort: 80
        CidrIp: 0.0.0.0/0

MyEC2Instance:
  Type: AWS::EC2::Instance
  Properties:
    ImageId: ami-0c55b159cbfafe1f0  # Replace with a valid AMI ID
    InstanceType: t2.micro
    SubnetId: !Ref MySubnet
    SecurityGroupIds:
      - !Ref MySecurityGroup
    Tags:
      - Key: Name
        Value: CloudFormationInstance

MyRDSInstance:
  Type: AWS::RDS::DBInstance
  Properties:
    DBInstanceIdentifier: mydatabase
    Engine: mysql
    MasterUsername: admin
    MasterUserPassword: Password123
    AllocatedStorage: 20
    DBInstanceClass: db.t2.micro
    PubliclyAccessible: true
```

---

3. Deploy the CloudFormation Stack

Use AWS CLI or the AWS Management Console to deploy the stack.

Using AWS CLI

Run the following command:

```
aws cloudformation create-stack --stack-name MyCloudFormationStack
--template-body file://cloudformation-template.yaml --capabilities
CAPABILITY_IAM
```

This command:

- Deploys the template.
- Creates the specified AWS resources.

To check the stack status:

aws cloudformation describe-stacks --stack-name MyCloudFormationStack

---

4. Verify the Deployment

- Go to **AWS Console > CloudFormation** to check the stack status.
- Navigate to **EC2, RDS, and VPC** in the AWS Console to verify the resources were created.

---

5. Update the Stack (If Needed)

Modify the cloudformation-template.yaml file and update the stack:

sh

aws cloudformation update-stack --stack-name MyCloudFormationStack --template-body file://cloudformation-template.yaml

---

6. Delete the Stack

To remove all resources:

aws cloudformation delete-stack --stack-name MyCloudFormationStack

This ensures that all deployed resources are properly deleted.

Conclusion

This project demonstrates how to use AWS CloudFormation for automating infrastructure provisioning. By defining resources in a YAML/JSON template, you can consistently deploy and manage AWS services efficiently. 🚀

Project 3: AWS CDK Deployment – Use AWS Cloud Development Kit (CDK) to manage cloud infrastructure.

AWS CDK Project: Deploying S3, Lambda, API Gateway, and RDS

This updated CDK stack will:
✅ Create an **S3 Bucket** for file storage
✅ Deploy an **AWS Lambda Function** to process requests
✅ Set up **API Gateway** to expose the Lambda function as an HTTP endpoint
✅ Deploy an **Amazon RDS (PostgreSQL) instance** for database storage

---

Step 1: Install Required AWS CDK Libraries

Inside your AWS CDK project directory, install the required AWS CDK libraries:

npm install @aws-cdk/aws-s3 @aws-cdk/aws-lambda @aws-cdk/aws-apigateway @aws-cdk/aws-rds @aws-cdk/aws-ec2

Step 2: Define AWS Resources in CDK Stack

Edit the lib/my-cdk-app-stack.ts file and replace its content with the following:

typescript

```
import * as cdk from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as apigateway from 'aws-cdk-lib/aws-apigateway';
import * as rds from 'aws-cdk-lib/aws-rds';
import * as ec2 from 'aws-cdk-lib/aws-ec2';

export class MyCdkAppStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create an S3 Bucket
    const myBucket = new s3.Bucket(this, 'MyBucket', {
      versioned: true,
      removalPolicy: cdk.RemovalPolicy.DESTROY
    });

    // Create a VPC for the RDS instance
    const vpc = new ec2.Vpc(this, 'MyVpc', {
      maxAzs: 2 // Deploy in 2 availability zones
    });

    // Create an RDS PostgreSQL database
    const database = new rds.DatabaseInstance(this, 'MyDatabase', {
      engine: rds.DatabaseInstanceEngine.postgres({
        version: rds.PostgresEngineVersion.VER_13,
      }),
      vpc,
      allocatedStorage: 20,
      instanceType: ec2.InstanceType.of(ec2.InstanceClass.T3, ec2.InstanceSize.MICRO),
      multiAz: false,
      publiclyAccessible: false,
      credentials: rds.Credentials.fromGeneratedSecret('admin'), // Auto-generate password
      removalPolicy: cdk.RemovalPolicy.DESTROY
    });

    // Create a Lambda function
    const myLambda = new lambda.Function(this, 'MyLambdaFunction', {
      runtime: lambda.Runtime.NODEJS_18_X,
      code: lambda.Code.fromAsset('lambda'), // Lambda code inside the 'lambda' directory
```

```
    handler: 'index.handler',
    environment: {
      BUCKET_NAME: myBucket.bucketName,
      DB_ENDPOINT: database.dbInstanceEndpointAddress
    }
});

// Grant the Lambda function access to S3
myBucket.grantReadWrite(myLambda);

// Grant the Lambda function access to the RDS database
database.grantConnect(myLambda);

// Create an API Gateway to expose the Lambda function
const api = new apigateway.LambdaRestApi(this, 'MyApiGateway', {
    handler: myLambda,
    proxy: true
});

new cdk.CfnOutput(this, 'API Gateway URL', {
    value: api.url
      });
    }
}
```

Step 3: Add Lambda Function Code

Create a new directory named lambda/ inside your CDK project and add a file called index.js with the following content:

javascript

```
const AWS = require('aws-sdk');
const s3 = new AWS.S3();

exports.handler = async (event) => {
    console.log("Received event:", JSON.stringify(event, null, 2));

    const response = {
        statusCode: 200,
        body: JSON.stringify({ message: "Hello from AWS Lambda via API Gateway!" })
    };

    return response;
```

};

Step 4: Bootstrap and Deploy the CDK Stack

Run the following commands to bootstrap the environment and deploy:

cdk bootstrap

cdk synth

cdk deploy

Step 5: Verify the Deployment

- Check the **S3 bucket** in the AWS Console.
- Find the **Lambda function** in AWS Lambda.
- Open the **API Gateway URL** from the terminal output and test it.
- Look for the **RDS database** under Amazon RDS.

Step 6: Clean Up (Optional)

If you want to delete all resources, run:

cdk destroy

Conclusion

You have now successfully deployed a **complete AWS infrastructure** using AWS CDK, including:

✅ An **S3 bucket** for storage
✅ A **Lambda function** for processing requests
✅ An **API Gateway** for exposing APIs
✅ An **RDS PostgreSQL database** for persistence

Project 4: Automate AWS Resource Tagging – Use AWS Lambda and Config Rules for enforcing tagging policies.

Automate AWS Resource Tagging using AWS Lambda and AWS Config Rules

Introduction

Tagging AWS resources is essential for cost allocation, security, and operational efficiency. However, ensuring consistent tagging across all resources can be challenging. AWS Lambda and AWS Config Rules can be used to **automate the enforcement of tagging policies**, ensuring that all resources comply with the organization's standards.

How it works:

- AWS Config monitors AWS resources and checks for compliance with predefined tagging policies.
- AWS Lambda automatically applies missing tags or alerts administrators when non-compliant resources are found.

Steps to Implement

1. Prerequisites

- An **AWS Account** with necessary permissions
- **AWS Lambda** execution role with permissions for EC2, S3, and other services
- **AWS Config** enabled

---

2. Create AWS Config Rule for Tag Compliance

AWS Config continuously monitors AWS resources and checks for compliance with a tagging policy.

1. **Go to AWS Config in the AWS Console**
2. Click on **Rules > Add Rule**
3. Search for **required-tags** (AWS Managed Rule)
4. Define the required tags, e.g.,
    - Environment: Production
    - Owner: DevOps
5. Click **Save**

---

3. Create an AWS Lambda Function to Auto-Tag Resources

Step 1: Create a Lambda Function

1. Open **AWS Lambda** in the AWS Console
2. Click **Create function** > Choose **Author from scratch**
3. Enter a name, e.g., AutoTaggingLambda
4. Select **Python 3.9** as the runtime
5. Choose an **IAM role** with necessary permissions (EC2, S3, Lambda execution)

Step 2: Add the Code

Use the following Python code to automatically add missing tags:

```
import boto3


required_tags = {
    "Environment": "Production",
    "Owner": "DevOps"
}


def tag_resource(resource_arn):
    client = boto3.client('resourcegroupstaggingapi')


    # Fetch current tags
    response = client.get_resources(ResourceARNList=[resource_arn])
    current_tags = {tag['Key']: tag['Value'] for tag in response['TagMappings'][0]['Tags']}


    # Identify missing tags
    missing_tags = {key: value for key, value in required_tags.items() if key not in current_tags}


    if missing_tags:
        client.tag_resources(ResourceARNList=[resource_arn], Tags=missing_tags)
        print(f"Tagged {resource_arn} with {missing_tags}")


def lambda_handler(event, context):
    for record in event['detail']['configurationItemDiff']['changedProperties']:
        resource_arn = event['detail']['configurationItem']['arn']
```

```
    tag_resource(resource_arn)

return "Tagging Completed"
```

Step 3: Add a Trigger for AWS Config

1. Go to the **Lambda function**
2. Click **Add trigger** > Select **AWS Config**
3. Choose the **Config Rule** created earlier
4. Click **Add**

---

4. Testing the Automation

- Create an **EC2 instance** or an **S3 bucket without required tags**
- AWS Config detects non-compliance
- AWS Lambda **automatically applies missing tags**

---

5. Monitoring & Alerts

- Use **AWS CloudWatch Logs** to monitor Lambda execution
- Set up **AWS SNS Notifications** to alert administrators about non-compliant resources

Conclusion

This automation ensures **consistent AWS resource tagging** by enforcing tagging policies using AWS Lambda and AWS Config Rules. It helps organizations track costs, improve security, and manage resources efficiently.

Project 5: AWS Systems Manager for Automated Patching – Manage EC2 instance patching automatically.

AWS Systems Manager for Automated Patching

Introduction

AWS Systems Manager (SSM) allows you to automate patch management for EC2 instances, ensuring they are always up-to-date with the latest security updates and patches. This project sets up **AWS Systems Manager Patch Manager** to automate OS patching on EC2 instances using Patch Baselines, Maintenance Windows, and Automation Documents.

Steps to Set Up Automated Patching

Step 1: Configure EC2 Instances for Systems Manager

1. **Create EC2 instances** (Amazon Linux, Ubuntu, or Windows).
2. **Attach IAM Role to EC2** with the following policies:
    - AmazonSSMManagedInstanceCore
    - AmazonEC2RoleforSSM

Ensure SSM Agent is installed and running (Amazon Linux 2 and Windows have it pre-installed).

```
sudo systemctl status amazon-ssm-agent  # For Amazon Linux
```

3. **Verify connectivity** by navigating to **AWS Systems Manager > Managed Instances** and checking if the EC2 instance appears.

Step 2: Create a Patch Baseline

1. Go to **AWS Systems Manager > Patch Manager**.
2. Click **Create Patch Baseline** and configure:
    - Select **OS Type** (Amazon Linux, Ubuntu, Windows, etc.).

- Define **auto-approval rules** (e.g., approve patches after X days).
- Specify any **patch exceptions** if needed.

3. Save and note the **Patch Baseline ID**.

Step 3: Associate Patch Baseline with EC2 Instances

1. Navigate to **Patch Manager > Patch Baselines**.
2. Select your created baseline and click **Modify Instance Association**.
3. Choose **EC2 instances** that should receive patches.
4. Apply the baseline.

Step 4: Configure a Maintenance Window

1. Go to **AWS Systems Manager > Maintenance Windows**.
2. Click **Create Maintenance Window** and configure:
   - **Schedule** (e.g., Every Sunday at 2 AM UTC).
   - **Duration & Cutoff time** (e.g., 2 hours, cutoff at 30 min).
   - Allow **targets to be registered**.
3. Save and **note the Maintenance Window ID**.

Step 5: Register Targets and Tasks

1. **Register EC2 Instances** with the Maintenance Window:
   - Click **Register Targets** > Select EC2 instances.
2. **Register a Patch Task**:
   - Click **Register a Task** > Choose **AWS-RunPatchBaseline** as the SSM document.
   - Configure:
     - **Operation**: Install
     - **Patch Group**: (if using patch groups)
     - **Rate Control**: Specify concurrency and error handling.
   - Save the task.

Step 6: Test and Monitor the Patch Process

1. Manually trigger the Maintenance Window to test the setup.
2. Monitor patch execution:
   - **AWS Systems Manager > Run Command**
   - **AWS Systems Manager > Patch Manager > Reports**
3. Ensure EC2 instances are patched correctly.

Conclusion

This setup ensures EC2 instances are automatically patched based on a defined schedule and patch policies. AWS Systems Manager provides detailed logs and reports to track compliance and patching progress.

Project 6: Automated Cost Optimization – Use AWS Lambda to detect and stop unused EC2 instances.

Automated Cost Optimization using AWS Lambda

Introduction

AWS Lambda can be used to detect and stop unused EC2 instances to optimize cloud costs. By leveraging AWS services like CloudWatch, Lambda, and AWS SDK (Boto3), we can automate the process of monitoring EC2 instance usage and shutting down idle instances. This project helps reduce unnecessary cloud expenses by identifying instances with low CPU utilization over a defined period.

Steps to Implement the Project

1. Prerequisites

- AWS account with necessary IAM permissions
- Amazon EC2 instances running
- AWS CloudWatch enabled for EC2 instances
- AWS Lambda function with execution role

---

2. Create an IAM Role for Lambda

1. Go to AWS IAM Console → Roles → Create role
2. Select **AWS Service** → Choose **Lambda**
3. Attach policies:
    - AmazonEC2ReadOnlyAccess (to fetch EC2 usage details)
    - AWSLambdaBasicExecutionRole (to allow logging)
    - AmazonEC2FullAccess (to stop EC2 instances)
4. Create the role and note down the ARN

---

3. Create a CloudWatch Alarm to Monitor EC2 Usage

1. Go to **CloudWatch → Alarms →** Create Alarm
2. Select **EC2 Metrics → CPUUtilization**
3. Set a condition:
    - If **CPU Utilization < 5%** for **1 hour**
4. Choose **SNS Topic** (Create one if needed)
5. Configure the SNS topic to trigger the Lambda function

---

4. Create AWS Lambda Function to Stop Unused Instances

1. Go to **AWS Lambda** → Create function
2. Select **Author from scratch**
3. Choose **Python 3.x** as the runtime
4. Assign the IAM role created earlier
5. Add the following Python code:

python

```
import boto3

def lambda_handler(event, context):
    ec2 = boto3.client('ec2')
    cloudwatch = boto3.client('cloudwatch')

    # Get all running instances
    instances = ec2.describe_instances(Filters=[{'Name': 'instance-state-name', 'Values': ['running']}])

    for reservation in instances['Reservations']:
        for instance in reservation['Instances']:
            instance_id = instance['InstanceId']

            # Get CPU utilization metrics
            metrics = cloudwatch.get_metric_statistics(
                Period=3600,
                StartTime=datetime.utcnow() - timedelta(hours=1),
                EndTime=datetime.utcnow(),
                MetricName='CPUUtilization',
```

```
            Namespace='AWS/EC2',
            Statistics=['Average'],
            Dimensions=[{'Name': 'InstanceId', 'Value': instance_id}]
        )

        if metrics['Datapoints']:
            avg_cpu = metrics['Datapoints'][0]['Average']
            if avg_cpu < 5.0:  # Stop instance if CPU usage is low
                ec2.stop_instances(InstanceIds=[instance_id])
                print(f"Stopped instance: {instance_id}")

    return "EC2 Optimization Completed"
```

---

5. Configure Lambda Trigger

1. In **AWS Lambda**, go to the created function
2. Click **Add Trigger** → Choose **CloudWatch Events**
3. Select the SNS topic created in Step 3
4. Save and deploy the function

---

6. Testing the Setup

1. Manually lower the CPU usage of an EC2 instance
2. Wait for CloudWatch alarm to trigger the Lambda function
3. Check EC2 instance state after execution

---

7. Monitor and Improve

- Use **CloudWatch Logs** to debug errors
- Modify the **CPU threshold** based on business needs
- Extend functionality by sending notifications when instances are stopped

Conclusion

This AWS Lambda-based automation helps in cost optimization by detecting and stopping idle EC2 instances. It reduces unnecessary expenses and ensures better resource management.

Project 7: Self-Healing Infrastructure – Use Terraform to create auto-healing EC2 instances with AWS Auto Scaling.

Introduction:
Self-healing infrastructure ensures high availability and resilience by automatically recovering from failures. In AWS, this can be achieved using **EC2 instances with Auto Scaling Groups (ASG) and Load Balancers**. Terraform helps automate the provisioning and management of this infrastructure.

In this project, we will:

- Use **Terraform** to deploy an **EC2 instance** in an **Auto Scaling Group**.
- Attach a **Load Balancer** to distribute traffic.
- Use **Auto Scaling policies** to replace unhealthy instances automatically.

Steps to Implement

Step 1: Install Prerequisites

Ensure you have the following installed:

- Terraform
- AWS CLI (configured with IAM credentials)
- SSH key pair for EC2 access

Step 2: Create Terraform Configuration Files

1. Define Provider (provider.tf)

hcl

```
provider "aws" {
  region = "us-east-1"
}
```

---

2. Create a VPC and Subnets (vpc.tf)

hcl

```
resource "aws_vpc" "main_vpc" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "public_subnet" {
  vpc_id            = aws_vpc.main_vpc.id
  cidr_block        = "10.0.1.0/24"
  map_public_ip_on_launch = true
}
```

---

3. Create a Security Group (security.tf)

hcl

```
resource "aws_security_group" "web_sg" {
  vpc_id = aws_vpc.main_vpc.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
```

```
  cidr_blocks = ["0.0.0.0/0"]
 }
}
```

---

4. Create Launch Template (launch_template.tf)

hcl

```
resource "aws_launch_template" "app_template" {
 name          = "app-template"
 image_id      = "ami-0c55b159cbfafe1f0" # Replace with the latest AMI
 instance_type = "t2.micro"

 network_interfaces {
  associate_public_ip_address = true
  security_groups             = [aws_security_group.web_sg.id]
 }

 user_data = base64encode(<<-EOF
             #!/bin/
             echo "Hello, World" > /var/www/html/index.html
             sudo systemctl start httpd
             EOF
 )
}
```

---

5. Configure Auto Scaling Group (autoscaling.tf)

hcl

```
resource "aws_autoscaling_group" "app_asg" {
 vpc_zone_identifier = [aws_subnet.public_subnet.id]
 desired_capacity    = 2
 max_size            = 3
 min_size            = 1

 launch_template {
  id      = aws_launch_template.app_template.id
  version = "$Latest"
 }

 health_check_type = "EC2"
 health_check_grace_period = 300
}
```

---

6. Attach a Load Balancer (load_balancer.tf)

hcl

```
resource "aws_lb" "app_lb" {
  name               = "app-load-balancer"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.web_sg.id]
  subnets            = [aws_subnet.public_subnet.id]
}

resource "aws_lb_target_group" "app_tg" {
  name     = "app-target-group"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.main_vpc.id
}

resource "aws_lb_listener" "app_listener" {
  load_balancer_arn = aws_lb.app_lb.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.app_tg.arn
  }
}
```

Step 3: Initialize and Deploy Terraform

Run the following commands:

terraform init

terraform apply -auto-approve

Step 4: Testing Self-Healing Feature

1. Find a running EC2 instance from the **Auto Scaling Group**.
2. Manually **terminate** the instance from the AWS Console.
3. Auto Scaling should **automatically launch a new instance** to replace it.

Conclusion

With this setup, Terraform provisions an **auto-healing infrastructure** using AWS Auto Scaling and Load Balancer. If an instance fails, Auto Scaling will replace it, ensuring high availability

---

Project 8: Event-Driven Infrastructure Provisioning – Use AWS EventBridge and CloudFormation to automate infrastructure deployment.

Introduction

Event-driven infrastructure provisioning automates resource deployment in response to specific AWS events. Using **AWS EventBridge** and **AWS CloudFormation**, we can trigger infrastructure deployment dynamically based on changes in the environment. This ensures efficient resource management and scalability without manual intervention.

Steps to Implement the Project

Step 1: Create an AWS CloudFormation Template

- Define an **infrastructure-as-code (IaC)** template in YAML or JSON to provision AWS resources.
- Example: A simple EC2 instance and S3 bucket.
- Save it as cloudformation-template.yaml.

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyS3Bucket:
    Type: "AWS::S3::Bucket"
  MyEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      InstanceType: "t2.micro"
      ImageId: "ami-0abcdef1234567890"  # Replace with a valid AMI ID
```

Step 2: Create an AWS Lambda Function

- This function will be triggered by AWS EventBridge to deploy CloudFormation stacks dynamically.
- Write a simple **Python** script (lambda_function.py) to trigger CloudFormation.

python

```
import boto3

cf_client = boto3.client('cloudformation')

def lambda_handler(event, context):
    response = cf_client.create_stack(
        StackName="MyEventTriggeredStack",
        TemplateURL="https://s3.amazonaws.com/my-bucket/cloudformation-template.yaml",
        Capabilities=['CAPABILITY_NAMED_IAM']
    )
```

```
    return response
```

- Upload the script as a Lambda function and grant it **IAM permissions** for CloudFormation execution.

Step 3: Configure AWS EventBridge Rule

- Go to **AWS EventBridge** and create a new rule.
- Select an event pattern that triggers CloudFormation provisioning, such as:
  - When an S3 object is created
  - When an EC2 instance starts
- Set the **target** as the AWS Lambda function.

Example Event Pattern (S3 Object Creation):

json

```
{
  "source": ["aws.s3"],
  "detail-type": ["Object Created"],
  "detail": {
    "bucket": {
      "name": ["my-trigger-bucket"]
    }
  }
}
```

Step 4: Test the Event-Driven Deployment

- Upload a file to the specified **S3 bucket** to trigger the event.
- Verify if the **Lambda function** runs successfully.
- Check **CloudFormation Stacks** in AWS Console to ensure resources are provisioned.

Conclusion

This project automates AWS resource provisioning using an event-driven approach. **AWS EventBridge** detects changes, triggers **AWS Lambda**, and initiates infrastructure deployment using **AWS CloudFormation**. This enhances **automation, scalability, and operational efficiency** in cloud environments.

Project 9: Automated EC2 Scaling Based on Load – Use Auto Scaling policies with CloudWatch to optimize cost and performance.

Introduction

AWS Auto Scaling ensures that your EC2 instances dynamically adjust based on real-time traffic and workload demands. By integrating Auto Scaling with CloudWatch, you can optimize costs by automatically adding or removing instances based on CPU utilization or other metrics. This ensures high availability, fault tolerance, and efficient resource management.

Steps to Implement Auto Scaling Based on Load

Step 1: Create an EC2 Instance

1. Log in to your **AWS Management Console**.
2. Navigate to **EC2** → Click **Launch Instance**.
3. Choose an **Amazon Machine Image (AMI)** (e.g., Amazon Linux 2).
4. Select an instance type (e.g., t2.micro for free-tier).

5. Configure security group to allow SSH (port 22) and HTTP (port 80).
6. Add a key pair to access the instance securely.
7. Click **Launch**.

Step 2: Create a Launch Template

1. Navigate to **EC2 → Launch Templates**.
2. Click **Create Launch Template**.
3. Provide a **template name** and **description**.
4. Select the **AMI** and **instance type**.
5. Define key pair, security groups, and user data script (if needed).
6. Click **Create Launch Template**.

Step 3: Configure an Auto Scaling Group (ASG)

1. Go to **EC2 → Auto Scaling Groups**.
2. Click **Create Auto Scaling Group**.
3. Select the previously created **Launch Template**.
4. Define the **minimum**, **desired**, and **maximum instances** (e.g., Min: 1, Max: 5).
5. Select the **VPC and Subnets** where instances will be launched.
6. Attach an existing **Load Balancer** (optional).
7. Enable **Health Check** to replace unhealthy instances.
8. Click **Create Auto Scaling Group**.

Step 4: Create CloudWatch Alarms for Scaling

1. Navigate to **CloudWatch → Alarms →** Click **Create Alarm**.
2. Select a metric (e.g., **EC2 → Per-Instance Metrics → CPU Utilization**).
3. Set a threshold (e.g., **Scale-Out: CPU > 70%**, **Scale-In: CPU < 30%**).
4. Choose an **Auto Scaling Policy**:
    - **Scale Out** (Add an instance if CPU exceeds 70%).
    - **Scale In** (Remove an instance if CPU drops below 30%).
5. Click **Create Alarm**.

Step 5: Attach Scaling Policies to the Auto Scaling Group

1. Go to **EC2 → Auto Scaling Groups**.
2. Select your ASG → **Automatic Scaling** → Click **Add Policy**.
3. Choose **Target Tracking**, **Step Scaling**, or **Simple Scaling**.
4. Attach the CloudWatch alarm to **increase or decrease instances** automatically.
5. Click **Apply**.

Step 6: Testing Auto Scaling

1. Generate high CPU load on an instance (e.g., using stress tool in Linux).
2. Verify in **CloudWatch** if the alarm triggers scaling.
3. Check **EC2 Instances** to see if new instances are launched or terminated based on load.

Conclusion

This AWS project demonstrates how **Auto Scaling Groups (ASG)** and **CloudWatch Alarms** help dynamically adjust EC2 instances based on CPU utilization, optimizing performance and cost. This setup ensures your application remains **highly available** and scales automatically according to demand.

---

4. Security & IAM Management

Project 1: IAM Role & Policy Management – Secure AWS resources using IAM policies.

Introduction

AWS Identity and Access Management (IAM) allows secure control over AWS resources by defining permissions using IAM roles and policies. This project focuses on creating IAM users, roles, and policies to enforce least-privilege access, ensuring security and compliance.

Steps to Implement IAM Role & Policy Management

Step 1: Create an IAM User

1. **Login to AWS Console** → Go to **IAM**.
2. Click **Users** → **Add User**.
3. Enter a **Username** and enable **Programmatic Access** (for CLI/API).
4. Click **Next: Permissions** and **Attach existing policies directly**.
5. Choose **AdministratorAccess** (or create a custom policy).
6. Click **Next** → **Review** → **Create User**.
7. Download the **Access Key ID** and **Secret Access Key**.

Step 2: Create a Custom IAM Policy

1. In the IAM Console, navigate to **Policies** → **Create Policy**.

Select **JSON** and define a custom policy (e.g., S3 Read-Only Access):
json

```
{
 "Version": "2012-10-17",
 "Statement": [
   {
    "Effect": "Allow",
    "Action": "s3:ListBucket",
    "Resource": "arn:aws:s3:::example-bucket"
   },
   {
    "Effect": "Allow",
    "Action": "s3:GetObject",
    "Resource": "arn:aws:s3:::example-bucket/*"
   }
 ]
}
```

2.
3. Click **Next** → **Name Policy** (e.g., S3ReadOnlyPolicy) → **Create Policy**.

Step 3: Create an IAM Role

1. In the IAM Console, go to **Roles** → **Create Role**.
2. Choose **AWS Service** and select **EC2** (or another service).
3. Click **Next** and attach the **S3ReadOnlyPolicy**.
4. Name the role (e.g., S3ReadOnlyRole) and create it.

Step 4: Assign the Role to an EC2 Instance

1. Go to **EC2 Console** → Select an instance.
2. Click **Actions** → **Security** → **Modify IAM Role**.
3. Select S3ReadOnlyRole and apply changes.

Step 5: Verify IAM Role & Policy

SSH into the EC2 instance and run:
aws s3 ls s3://example-bucket

    - If successful, the IAM role is working correctly.
2. Try an unauthorized action, such as deleting an object, to confirm restricted access.

Conclusion

This project demonstrated secure AWS resource access using IAM roles and policies. By applying **least privilege principles**, we ensured controlled access to AWS services, improving security and compliance.

Project 2: AWS Secrets Manager for Credential Storage – Secure database credentials and API keys.

Introduction
AWS Secrets Manager is a secure service that helps manage sensitive information like database credentials, API keys, and other secrets. It automates secret rotation, provides fine-grained access control, and integrates seamlessly with AWS services, ensuring enhanced security.

Steps to Implement AWS Secrets Manager for Credential Storage

Step 1: Create a Secret in AWS Secrets Manager

1. Sign in to the AWS Management Console.
2. Navigate to **AWS Secrets Manager**.
3. Click **Store a new secret**.
4. Choose the secret type, e.g., **Credentials for RDS database**.
5. Enter **username** and **password** for the database.
6. Select **Amazon RDS database** (if applicable) or manually input details.
7. Click **Next**.

Step 2: Configure Secret Settings

1. Provide a **Secret name** (e.g., mydb-credentials).
2. Optionally enable **automatic rotation** and select an AWS Lambda function to rotate secrets.
3. Click **Next**, review settings, and click **Store secret**.

Step 3: Retrieve the Secret in AWS Applications

Use the AWS SDK or AWS CLI to fetch the stored secret securely in your application.

Using AWS CLI

```
aws secretsmanager get-secret-value --secret-id mydb-credentials
```

Using AWS SDK (Python Boto3 Example)

```
python


import boto3

import json


client = boto3.client('secretsmanager', region_name='us-east-1')


response = client.get_secret_value(SecretId='mydb-credentials')

secret = json.loads(response['SecretString'])


db_username = secret['username']

db_password = secret['password']

print(f"DB User: {db_username}")
```

Step 4: Grant IAM Permissions to Access Secrets

1. Attach an **IAM policy** to the application or Lambda function needing access.

2. Example IAM policy:

json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "secretsmanager:GetSecretValue",
      "Resource":
"arn:aws:secretsmanager:us-east-1:123456789012:secret:mydb-credentials"
    }
  ]
}
```

Step 5: Integrate Secrets Manager with Amazon RDS (Optional)

If using **Amazon RDS**, enable **automatic rotation** using Secrets Manager to periodically update credentials.

Step 6: Implement Secret Rotation (Optional)

1. Create a Lambda function for rotation.
2. Attach an IAM role allowing Secrets Manager and RDS access.
3. Configure Secrets Manager to use this function for rotation.

Conclusion

AWS Secrets Manager provides a **secure, automated, and scalable** way to store and manage sensitive credentials. It improves security by preventing hardcoded secrets and enabling **automatic rotation**.

Project 3: API Gateway Security – Implement authentication with Cognito and protection using AWS WAF.

Introduction

Amazon API Gateway allows developers to create, deploy, and manage APIs securely. To enhance security, AWS Cognito provides user authentication, while AWS Web Application Firewall (WAF) protects APIs from common web threats like SQL injection and cross-site scripting (XSS). This project demonstrates how to secure an API Gateway using **AWS Cognito for authentication** and **AWS WAF for protection**.

Steps to Implement

Step 1: Create an API Gateway

1. Navigate to the **AWS Management Console** → Open **API Gateway**.
2. Click **Create API** → Choose **REST API** or **HTTP API**.
3. Select **New API**, enter a name, and choose **Regional**.
4. Click **Create API**.

Step 2: Configure AWS Cognito User Pool for Authentication

1. Navigate to **AWS Cognito** → Select **Create a User Pool**.
2. Enter a name (e.g., MyUserPool) and click **Step Through Settings**.
3. Enable **Email** as the sign-in option and configure password policies.
4. Under **App Clients**, create a new client (disable secret generation).

5. Go to **App Client Settings**, enable **Cognito User Pool** and enter the **callback URL**.
6. Save changes and note the **User Pool ID** and **App Client ID**.

Step 3: Enable Cognito Authentication in API Gateway

1. Open **API Gateway** → Select the API created earlier.
2. Under **Authorization**, choose **Create a new Cognito Authorizer**.
3. Enter an **Authorizer name**, select **Cognito**, and provide the **User Pool ID**.
4. Click **Create** and attach this **authorizer** to the API methods.
5. Deploy the API and note the **invoke URL**.

Step 4: Create and Attach an AWS WAF WebACL

1. Open **AWS WAF** → Click **Create WebACL**.
2. Select **Regional** scope and choose **API Gateway** as the resource.
3. Add rules:
    - Enable **AWS Managed Rules** for SQL Injection and XSS protection.
    - Add a **Rate-Based Rule** to block excessive requests.
4. Review and create the WebACL.
5. Associate the WebACL with your API Gateway.

Step 5: Test the Secure API

1. Authenticate using **Cognito** and get an access token.
2. Send an API request with the **Bearer token** in the Authorization header.
3. Verify WAF blocks malicious requests.

Conclusion

This setup ensures **secure authentication with Cognito** and **protection against threats using AWS WAF**, making API Gateway more robust.

Project 4: AWS Security Hub & GuardDuty – Monitor security threats and enforce compliance.

Introduction

AWS Security Hub and Amazon GuardDuty are two essential AWS security services that help monitor, detect, and respond to security threats in your AWS environment.

- **AWS Security Hub** provides a comprehensive view of your security posture and compliance status across AWS accounts using automated security checks based on AWS best practices and compliance frameworks (CIS, PCI DSS, etc.).
- **Amazon GuardDuty** is a threat detection service that continuously monitors AWS resources for malicious activities, unauthorized access, and security threats using machine learning and threat intelligence.

By integrating Security Hub with GuardDuty, you can automate threat detection, centralize security findings, and improve incident response.

Complete Steps for Setting Up AWS Security Hub & GuardDuty

Step 1: Enable AWS Security Hub

1. Sign in to the AWS Management Console and navigate to **AWS Security Hub**.
2. Click **Enable Security Hub** to start monitoring your AWS account.
3. Choose the compliance standards you want to enable (CIS, PCI DSS, AWS Foundational Security Best Practices).
4. (Optional) If using multiple AWS accounts, configure Security Hub in **AWS Organizations** for centralized security management.

Step 2: Enable Amazon GuardDuty

1. Open the **GuardDuty** console in AWS.
2. Click **Enable GuardDuty** to activate continuous monitoring.
3. (Optional) If using AWS Organizations, configure **GuardDuty multi-account management** for centralized threat monitoring.
4. GuardDuty will start analyzing CloudTrail logs, VPC Flow Logs, and DNS logs for suspicious activity.

Step 3: Integrate GuardDuty Findings into Security Hub

1. Navigate to the **Security Hub Console**.
2. Under **Integrations**, find **Amazon GuardDuty** and enable integration.
3. Now, all GuardDuty threat findings (such as compromised IAM credentials or suspicious network activity) will be visible in Security Hub.

Step 4: Automate Threat Responses with AWS Lambda & EventBridge

1. Open the **Amazon EventBridge** console.
2. Create a **new rule** to trigger actions when a specific GuardDuty finding is detected.
3. Set the event source as **AWS Security Hub** or **GuardDuty** and select specific threat types (e.g., unauthorized access, malware activity).
4. Choose **AWS Lambda** as the target to trigger automatic remediation actions, such as:
    - Isolating an EC2 instance
    - Blocking an IP using AWS WAF
    - Disabling compromised IAM credentials

Step 5: Configure Security Alerts and Notifications

1. Open **Amazon SNS (Simple Notification Service)**.
2. Create a new SNS topic for security alerts.
3. Subscribe your email or Slack channel to receive notifications.
4. In **EventBridge**, create a rule that sends notifications when Security Hub detects high-severity issues.

Step 6: Monitor and Respond to Security Findings

1. Regularly review findings in **AWS Security Hub** and **GuardDuty** dashboards.
2. Use AWS Security Hub's **automated insights** to remediate security risks.
3. Continuously update security policies and compliance frameworks to meet industry standards.

Conclusion

By setting up AWS Security Hub and GuardDuty, you gain real-time visibility into security threats, automate compliance checks, and improve incident response with automated remediation. This integration helps in proactively securing your AWS environment and enforcing compliance at scale.

Project 5: Automated Compliance Auditing – Use AWS Config to check security best practices.

Introduction

AWS Config is a powerful service that continuously monitors, records, and evaluates AWS resource configurations against best practices and security policies. This project helps automate compliance auditing by setting up AWS Config rules to check security best practices, such as ensuring encryption is enabled, IAM policies follow least privilege principles, and resources are properly tagged.

Complete Steps

Step 1: Set Up AWS Config

1. **Sign in to AWS Management Console**
2. **Navigate to AWS Config**
    - Open AWS Config from the AWS Console.
3. **Choose a Recorder**

- Click **"Set up AWS Config"**
- Select the AWS **resources** you want to track (e.g., S3, EC2, IAM).

4. **Set Up an S3 Bucket for Logs**
   - Create or specify an existing S3 bucket for storing configuration history.
5. **Enable AWS Config Recorder**
   - Start recording resource configurations across your AWS account.

Step 2: Define Compliance Rules

1. **Navigate to the AWS Config Rules Section**
2. **Create AWS Config Rules**
   - Choose **predefined rules** (e.g., s3-bucket-public-read-prohibited, iam-user-no-inline-policies).
   - Or create **custom rules** using AWS Lambda.
3. **Set Rule Parameters**
   - Define required settings, such as enforcing encryption for EBS volumes.
4. **Enable Automatic Remediation (Optional)**
   - Use AWS Systems Manager Automation to fix non-compliant resources automatically.

Step 3: Evaluate Compliance & Generate Reports

1. **Monitor Compliance Status**
   - Go to **AWS Config Dashboard** and check rule evaluations.
2. **View Non-Compliant Resources**
   - Identify which AWS resources violate best practices.
3. **Generate Compliance Reports**
   - Use AWS Config's reporting feature to generate compliance summaries.
4. **Enable Amazon SNS Notifications (Optional)**
   - Get alerts for non-compliance by setting up SNS notifications.

Step 4: Automate Compliance Auditing with AWS Lambda (Optional)

1. **Create an AWS Lambda Function**
   - Write a Python function to check compliance and log results.
2. **Integrate with AWS Config**
   - Configure AWS Lambda to trigger when a rule is violated.
3. **Remediate Issues Automatically**
   - Use AWS Systems Manager to enforce compliance automatically.

Conclusion

By leveraging AWS Config, automated rules, and optional remediation actions, you can continuously audit your AWS environment for security best practices. This ensures compliance with industry standards, improves security posture, and simplifies governance.

Project 6: Zero Trust Security Implementation – Enforce strict security rules using AWS IAM and VPC security groups.

Introduction

Zero Trust Security is a security model that enforces strict access controls and continuous verification for all users, devices, and applications. It assumes that no entity—inside or outside the network—should be trusted by default. In AWS, Zero Trust can be implemented using **IAM (Identity and Access Management)** for identity-based security and **VPC Security Groups** for network-level security.

This project focuses on enforcing **least privilege access** and **network segmentation** by configuring IAM policies, roles, and VPC security groups to restrict access to AWS resources.

Steps to Implement Zero Trust Security in AWS

Step 1: Create an IAM Role with Least Privilege Access

1. Sign in to the AWS Management Console.
2. Go to **IAM > Roles** and click **Create role**.
3. Choose **AWS Service** (e.g., EC2) as the trusted entity.

Attach a custom policy with the minimum necessary permissions. Example policy:
json

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "s3:GetObject",
    "s3:ListBucket"
   ],
   "Resource": [
    "arn:aws:s3:::my-secure-bucket",
    "arn:aws:s3:::my-secure-bucket/*"
   ]
  }
 ]
}
```

Attach the role to your EC2 instances or services that need restricted access.

Step 2: Enforce Multi-Factor Authentication (MFA) for IAM Users

1. Navigate to **IAM > Users** and select the user.
2. Click **Security credentials**, then **Manage MFA**.
3. Choose **Virtual MFA device**, scan the QR code, and enter the generated codes.

Modify the user's policy to enforce MFA authentication:
json

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Deny",
   "Action": "*",
   "Resource": "*",
   "Condition": {
    "BoolIfExists": {
     "aws:MultiFactorAuthPresent": false
    }
   }
  }
 ]
```

}

Step 3: Create a VPC with Secure Security Groups

1. Go to **VPC > Security Groups** and create a new Security Group.
2. Configure **inbound rules** with strict access control:
    - Allow SSH (port 22) **only from a trusted IP**.
    - Allow HTTP/HTTPS (ports 80, 443) **only from CloudFront or ALB**.
    - Block all other traffic by default.
3. Example rules:

Type	Protocol	Port Range	Source
SSH	TCP	22	203.0.113.0/32 (Your IP)
HTTP	TCP	80	0.0.0.0/0 (Any)
HTTPS	TCP	443	0.0.0.0/0 (Any)
All other traffic	-	-	Deny by default

Step 4: Implement IAM Access Analyzer for Continuous Monitoring

1. Open **IAM > Access Analyzer**.
2. Click **Create analyzer**, select the **account scope**, and create it.
3. Review and remediate any unintended permissions detected.

Step 5: Enforce Logging and Auditing with CloudTrail and GuardDuty

1. **Enable AWS CloudTrail** to monitor API calls:
    - Go to **CloudTrail > Trails > Create Trail**.
    - Enable logging for **S3, Lambda, IAM, EC2** events.
2. **Enable GuardDuty** for threat detection:
    - Go to **GuardDuty > Enable GuardDuty**.
    - Review findings and take action on suspicious activities.

Conclusion

By implementing **strict IAM roles**, **MFA enforcement**, **VPC security groups**, and **continuous monitoring**, you can establish a **Zero Trust Security model** in AWS. This approach minimizes the risk of unauthorized access and strengthens your cloud security posture.

Project 7: CloudTrail Logging & SIEM Integration – Integrate AWS CloudTrail logs with a SIEM tool for security monitoring.

CloudTrail Logging & SIEM Integration

Introduction:

AWS CloudTrail records all API activity across AWS services, helping in security analysis and compliance auditing. Integrating CloudTrail logs with a Security Information and Event Management (SIEM) tool enables real-time monitoring, threat detection, and security incident response.

Steps to Integrate AWS CloudTrail Logs with a SIEM Tool

Step 1: Enable AWS CloudTrail Logging

1. Sign in to the **AWS Management Console**.
2. Navigate to **CloudTrail → Trails → Create trail**.
3. Provide a **Trail name** (e.g., security-trail).
4. Set **Apply trail to all regions** to **Yes** for centralized logging.
5. Choose an **S3 bucket** (Create a new one if required) for log storage.
6. Enable **CloudWatch Logs** if real-time monitoring is needed.
7. Click **Create**.

Step 2: Configure Amazon S3 for SIEM Log Collection

1. Navigate to **S3** → Select your **CloudTrail log bucket**.

Go to **Permissions → Bucket Policy** and add the following policy:
json

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Principal": {"Service": "logs.amazonaws.com"},
   "Action": "s3:PutObject",
   "Resource": "arn:aws:s3:::your-bucket-name/AWSLogs/*"
  }
 ]
}
```

2. Enable **S3 Event Notifications** to trigger SIEM log ingestion.

Step 3: Configure AWS Lambda for Log Forwarding (Optional)

1. Navigate to **AWS Lambda → Create function**.
2. Choose **Author from scratch** and provide a name (e.g., cloudtrail-to-siem).
3. Select **Python 3.x** as the runtime.
4. In the code editor, use a script to forward logs to the SIEM tool via API.
5. Set up an **S3 Trigger** to invoke Lambda when new logs are added.

Step 4: Integrate with a SIEM Tool

🔹 **For Splunk:**

- Install **Splunk AWS Add-on**.
- Configure an **S3 input** for CloudTrail logs.
- Set up a Splunk **index** for security events.

🔹 **For ELK (Elasticsearch, Logstash, Kibana):**

- Deploy **Logstash** and configure an **S3 input plugin**.
- Parse CloudTrail logs using **Grok filters**.
- Store logs in **Elasticsearch** for visualization in **Kibana**.

🔹 **For AWS Security Hub (Optional):**

- Enable **AWS Security Hub** for automated security insights.
- Use **Amazon GuardDuty** for threat detection.

Step 5: Monitor and Analyze Security Logs

1. Use SIEM dashboards to track security incidents.
2. Set up **alerting rules** for suspicious activities.
3. Automate responses using **AWS Lambda** or **AWS EventBridge**.

Conclusion

By integrating AWS CloudTrail logs with a SIEM tool, you enhance real-time security monitoring and threat detection. This setup helps organizations meet compliance requirements and improve cloud security posture.

Project 8: AWS WAF for Application Security – Protect a web application from SQL injection and XSS attacks using AWS WAF.

AWS WAF for Application Security

AWS Web Application Firewall (WAF) helps protect web applications from common threats such as SQL injection, cross-site scripting (XSS), and other OWASP Top 10 vulnerabilities. By using AWS WAF, you can create rules to filter and block malicious traffic before it reaches your application.

Project Steps

1. Prerequisites

- An AWS account
- A running web application hosted on **Amazon API Gateway, Application Load Balancer (ALB), or CloudFront**
- IAM permissions to create and manage AWS WAF resources

2. Create an AWS WAF Web ACL

1. Go to the **AWS WAF & Shield** console.
2. Click **Web ACLs → Create web ACL**.
3. Choose the **Region** where your application is hosted.
4. Under **Resource type**, select the relevant option:
    - **CloudFront** (for global protection)
    - **Application Load Balancer (ALB)** (for regional protection)
    - **API Gateway** (for API security)
5. Name the Web ACL (e.g., MyWebAppWAF).

3. Add Rules for Protection

1. Click **Add rules →** Choose **Add managed rule groups**.
2. Select **AWS Managed Rules** for common security threats. Recommended rules:
    - **SQL Database** (blocks SQL injection attempts)
    - **Cross-Site Scripting (XSS)** (prevents XSS attacks)
    - **Core Rule Set (CRS)** (provides basic protection)
3. (Optional) Add **Custom Rules** to block specific IPs or allowlist certain traffic.
4. Set **Rule action** to **Block** or **Count** for testing.

4. Associate Web ACL with Resources

1. Under **Associated AWS resources**, choose the target:
    - Select **CloudFront distribution** if using CloudFront.
    - Select **ALB** or **API Gateway** if applicable.
2. Click **Add association**.

5. Configure Default Action & Logging

1. Set default action for unmatched requests (**Allow** or **Block**).
2. Enable logging (optional) by configuring an **Amazon Kinesis Data Firehose**.

6. Review & Deploy

1. Click **Review and Create Web ACL**.
2. Confirm the configuration and click **Create Web ACL**.

7. Test & Monitor

- Use **AWS WAF Metrics** in **CloudWatch** to monitor attack attempts.
- Modify rules if needed to optimize protection.

8. Validate Security

- Perform penetration testing using tools like **OWASP ZAP** or **SQLMap**.

- Ensure AWS WAF blocks malicious traffic while allowing legitimate requests.

With this setup, your web application is protected against SQL injection, XSS, and other web attacks using AWS WAF.

---

5. CI/CD Projects:

Project 1: AWS CodePipeline for Automated Deployments

AWS CodePipeline for Automated Deployments

Introduction

AWS CodePipeline is a continuous integration and continuous delivery (CI/CD) service that automates application deployments. It allows you to build, test, and deploy applications across different AWS services such as EC2, Lambda, and ECS. CodePipeline integrates with AWS CodeBuild, AWS CodeDeploy, GitHub, and other tools for automated software delivery.

Complete Steps

Step 1: Create a Source Repository

1. Use **AWS CodeCommit**, **GitHub**, or **S3** as the source repository.
2. If using CodeCommit:
    - Go to **AWS CodeCommit** > Create a new repository.
    - Clone the repo and push your application code.

Step 2: Create a Build Stage (Optional)

1. Go to **AWS CodeBuild** > Create a new build project.
2. Choose the source provider (CodeCommit/GitHub/S3).
3. Select the environment (Managed AWS Linux or custom Docker image).

Define buildspec.yml in your repository:
yaml

```
version: 0.2

phases:
  install:
    runtime-versions:
      java: corretto11
  build:
    commands:
      - mvn clean package
artifacts:
  files:
    - target/*.jar
```

4. Save and start the build process.

Step 3: Create a Deployment Stage

1. Use **AWS CodeDeploy**, **AWS Lambda**, or **Amazon ECS** for deployment.
2. If using EC2/On-Premises deployment:
    - Install the CodeDeploy agent on EC2 instances.
    - Create an **AppSpec** file for deployment.
    - Define deployment configuration in CodeDeploy.

Step 4: Set Up AWS CodePipeline

1. Go to **AWS CodePipeline** > Click "Create pipeline".

2. Choose a pipeline name and an **IAM role** with permissions.
3. **Add Source Stage**
    - Select the source repository (CodeCommit, GitHub, or S3).
    - Configure the webhook or polling for changes.
4. **Add Build Stage** (Optional)
    - Choose AWS CodeBuild and select the build project.
5. **Add Deploy Stage**
    - Choose CodeDeploy, Lambda, or ECS for deployment.

Step 5: Trigger and Test the Pipeline

1. Push a code change to the repository.
2. AWS CodePipeline detects changes and starts the build and deployment process.
3. Verify successful deployment in the AWS console.

Conclusion

AWS CodePipeline automates software delivery by integrating source control, build, and deployment tools into a single workflow. It ensures faster, consistent, and reliable deployments with minimal manual intervention.

Project 2: GitHub Actions + AWS ECS Deployment

GitHub Actions allows you to automate your CI/CD pipeline, and AWS ECS (Elastic Container Service) is used to deploy containerized applications. Below is a step-by-step guide to deploying an application from GitHub to AWS ECS using GitHub Actions.

Step 1: Prerequisites

- An **AWS account** with ECS and IAM access
- A **GitHub repository** with your application
- AWS CLI installed and configured (aws configure)
- Docker installed for building and pushing images
- An **ECS Cluster** and **ECS Service** set up

Step 2: Create AWS Resources

Create an ECS Cluster
aws ecs create-cluster --cluster-name my-cluster

1. **Create an ECS Task Definition** (Fargate example)
    - Define a task-definition.json file with the container details.

json

```
{
  "family": "my-task",
  "networkMode": "awsvpc",
  "containerDefinitions": [
    {
      "name": "my-container",
      "image": "<AWS_ACCOUNT_ID>.dkr.ecr.<AWS_REGION>.amazonaws.com/my-repo:latest",
      "memory": 512,
      "cpu": 256,
      "essential": true
    }
```

```
  ]
}
```

2.
    - Register the task:

```
sh

aws ecs register-task-definition --cli-input-json file://task-definition.json
```

3.

Create an ECS Service

```
sh

aws ecs create-service --cluster my-cluster --service-name my-service
--task-definition my-task --desired-count 1 --launch-type FARGATE
```

Step 3: Set Up GitHub Secrets

Go to your GitHub repository → **Settings** → **Secrets and variables** → **Actions** → **New repository secret**, and add:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_REGION
- ECR_REPOSITORY_NAME
- ECS_CLUSTER
- ECS_SERVICE
- TASK_DEFINITION

Step 4: Configure GitHub Actions Workflow

Create .github/workflows/deploy.yml in your repo:

```
yaml

name: Deploy to AWS ECS


on:
  push:
    branches:
      - main


jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4


      - name: Login to AWS ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1
```

```
    - name: Build, Tag, and Push Docker Image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        IMAGE_TAG: ${{ github.sha }}
      run: |
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY_NAME:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY_NAME:$IMAGE_TAG

    - name: Update ECS Task Definition
      id: task-def
      uses: aws-actions/amazon-ecs-render-task-definition@v1
      with:
        task-definition: task-definition.json
        container-name: my-container
        image: ${{ steps.login-ecr.outputs.registry }}/${{ secrets.ECR_REPOSITORY_NAME }}:${{ github.sha }}

    - name: Deploy to ECS
      uses: aws-actions/amazon-ecs-deploy-task-definition@v1
      with:
        cluster: ${{ secrets.ECS_CLUSTER }}
        service: ${{ secrets.ECS_SERVICE }}
        task-definition: ${{ steps.task-def.outputs.task-definition }}
        wait-for-service-stability: true
```

Step 5: Commit and Push Code

```
git add .
git commit -m "Added GitHub Actions for ECS deployment"
git push origin main
```

This triggers the GitHub Actions workflow, building the image, pushing it to ECR, and updating ECS.

Conclusion

Now, your GitHub Actions pipeline will automatically deploy to AWS ECS whenever you push changes to the main branch.

Project 3: Jenkins CI/CD with Terraform on AWS

Introduction

This project demonstrates a **CI/CD pipeline using Jenkins** to provision and deploy infrastructure on **AWS using Terraform**. The pipeline automates **infrastructure provisioning**, application deployment, and management.

Steps to Set Up Jenkins CI/CD with Terraform on AWS

1. Prerequisites

- AWS Account
- EC2 instance for Jenkins (Ubuntu preferred)
- IAM role and user for Terraform with required permissions
- S3 bucket and DynamoDB table for Terraform state management
- Jenkins installed on the EC2 instance
- Terraform installed on the Jenkins server

---

2. Setup Jenkins on AWS EC2

Launch an EC2 instance and install Jenkins:

sudo apt update && sudo apt install -y openjdk-11-jdk

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt update

sudo apt install -y jenkins

sudo systemctl start jenkins

sudo systemctl enable jenkins

Open port 8080 in AWS Security Group and get the admin password:
sudo cat /var/lib/jenkins/secrets/initialAdminPassword

- **Install necessary Jenkins plugins:**
  - Terraform
  - AWS Credentials
  - Git
  - Pipeline

---

3. Configure Terraform for AWS

Install Terraform on Jenkins EC2:
wget
https://releases.hashicorp.com/terraform/1.7.0/terraform_1.7.0_linux_amd64.zip

unzip terraform_1.7.0_linux_amd64.zip

sudo mv terraform /usr/local/bin/

terraform -v

- **Create an IAM User** in AWS with the following permissions:
  - AmazonS3FullAccess (for Terraform state)
  - AmazonDynamoDBFullAccess (for state locking)
  - AdministratorAccess (for deploying resources)
- **Store AWS Credentials in Jenkins**:
  - Go to **Jenkins > Manage Jenkins > Manage Credentials**
  - Add AWS Access Key and Secret Key

---

4. Create Terraform Files

- **main.tf**: Define AWS infrastructure (e.g., EC2, S3, RDS)
- **variables.tf**: Store variable values
- **outputs.tf**: Define output values
- **provider.tf**: Configure AWS provider

Example main.tf for EC2:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "jenkins_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  key_name      = "my-key"

  tags = {
    Name = "Jenkins-Server"
  }
}
```

Initialize Terraform:
terraform init

terraform validate

---

5. Set Up Jenkins Pipeline for Terraform

- Create a **Jenkins Pipeline Job**
- Add the following Jenkinsfile in the repository:

groovy

```
pipeline {
    agent any
    environment {
        AWS_ACCESS_KEY_ID     = credentials('AWS_ACCESS_KEY_ID')
        AWS_SECRET_ACCESS_KEY = credentials('AWS_SECRET_ACCESS_KEY')
    }
    stages {
        stage('Checkout Code') {
            steps {
                git 'https://github.com/your-repo/terraform-aws.git'
            }
        }
        stage('Initialize Terraform') {
            steps {
                sh 'terraform init'
            }
```

```
        }
        stage('Plan Terraform') {
            steps {
                sh 'terraform plan'
            }
        }
        stage('Apply Terraform') {
            steps {
                sh 'terraform apply -auto-approve'
            }
        }
    }
}
```

---

6. Run the CI/CD Pipeline

- Go to **Jenkins Dashboard → Select the Pipeline Job**
- Click on **Build Now**
- Monitor logs for successful deployment

---

7. Verify AWS Resources

- Go to AWS Console
- Check if EC2, S3, RDS, or other services were provisioned

---

8. Destroy Infrastructure (Optional)

If you want to delete the deployed infrastructure:
terraform destroy -auto-approve

Conclusion

This setup automates **AWS infrastructure deployment using Jenkins and Terraform**. It ensures that infrastructure is provisioned and managed efficiently as part of a CI/CD pipeline.

Project 4: Kubernetes GitOps with ArgoCD on AWS EKS

Introduction

GitOps is a DevOps approach where Git repositories serve as the source of truth for infrastructure and application deployment. **ArgoCD** is a declarative, GitOps continuous delivery tool for Kubernetes, ensuring that the desired application state in Git matches the running state in a Kubernetes cluster.

In this project, we'll set up **ArgoCD** on **AWS EKS** to enable automated application deployment and management.

Complete Steps

1. Prerequisites

- AWS account
- AWS CLI installed & configured
- kubectl installed
- eksctl installed
- Helm installed

- GitHub repository for GitOps

---

2. Create an EKS Cluster

eksctl create cluster --name gitops-eks --region us-east-1 --nodegroup-name worker-nodes --node-type t3.medium --nodes 2

Validate the cluster:

kubectl get nodes

---

3. Install ArgoCD in Kubernetes

Create a namespace:

kubectl create namespace argocd

Install ArgoCD using Helm:

helm repo add argo https://argoproj.github.io/argo-helm

helm install argocd argo/argo-cd --namespace argocd

Verify installation:

kubectl get pods -n argocd

---

4. Expose ArgoCD API

kubectl port-forward svc/argocd-server -n argocd 8080:443

Access ArgoCD at: https://localhost:8080

Retrieve the ArgoCD admin password:

kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 --decode

Login using:

argocd login localhost:8080

---

5. Deploy a Sample Application

Add your GitHub repo as an ArgoCD application source.

Example app.yaml:

yaml

apiVersion: argoproj.io/v1alpha1

kind: Application

metadata:

  name: my-app

```
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/your-repo.git
    targetRevision: main
    path: app-manifests
  destination:
    server: https://kubernetes.default.svc
    namespace: default
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
```

Apply the app:

```
kubectl apply -f app.yaml
```

---

6. Verify Deployment

```
kubectl get pods
kubectl get svc
```

Check ArgoCD UI for app status.

---

7. Enable Auto-sync in ArgoCD

```
argocd app set my-app --sync-policy automated
```

---

8. Clean Up Resources

To delete everything:

```
eksctl delete cluster --name gitops-eks
```

Conclusion

With this setup, ArgoCD continuously syncs your application with Git, automating Kubernetes deployments on AWS EKS using GitOps principles.

Project 5: Lambda-based Serverless CI/CD

Introduction

A **Lambda-based Serverless CI/CD** pipeline automates the deployment process using **AWS Lambda**, eliminating the need for traditional servers. It leverages **AWS CodeCommit, CodeBuild, CodePipeline, and AWS Lambda** to create a fully automated, cost-efficient deployment pipeline. This setup is ideal for deploying serverless applications, including AWS Lambda functions, API Gateway, and DynamoDB.

Complete Steps

1. Setup Source Code Repository (AWS CodeCommit or GitHub)

- Create a **CodeCommit** repository or use **GitHub/GitLab** to store your application code.
- Push your application code and necessary deployment scripts.

2. Configure AWS CodeBuild for Build & Test

Create a buildspec.yml file in your repository to define build steps:
yaml

```
version: 0.2

phases:
  install:
    runtime-versions:
      nodejs: 18
  build:
    commands:
      - echo "Building the application..."
      - npm install
      - zip -r app.zip .
artifacts:
  files:
    - app.zip
```

- Configure **AWS CodeBuild** with IAM permissions, environment variables, and the buildspec file.

3. Set Up AWS Lambda Deployment using AWS CodePipeline

- In **AWS CodePipeline**, create a new pipeline with the following stages:
  1. **Source Stage**: Pulls code from **AWS CodeCommit** or **GitHub**.
  2. **Build Stage**: Uses **AWS CodeBuild** to package the application.
  3. **Deploy Stage**: Uses **AWS Lambda** for serverless deployment.

4. Create AWS Lambda Function

- Create a Lambda function with the appropriate runtime (Node.js, Python, Java, etc.).
- Assign an **IAM Role** with **AmazonS3ReadOnlyAccess** and **AWSLambdaBasicExecutionRole** permissions.
- Update Lambda code automatically via AWS CodePipeline.

5. Deploy Lambda Function via AWS CodePipeline

- Use an **S3 bucket** to store the Lambda package.

Define a Lambda deployment script in the **CodePipeline Deploy Stage**:
aws lambda update-function-code --function-name MyLambdaFunction --s3-bucket my-bucket --s3-key app.zip

6. Automate Deployment using AWS EventBridge

- Configure **AWS EventBridge** to trigger the CI/CD pipeline when code is pushed to the repository.

7. Validate Deployment

- Monitor the Lambda function logs in **Amazon CloudWatch**.
- Test the deployed function using API Gateway or direct invocation.

Summary

This Lambda-based Serverless CI/CD pipeline enables **fully automated deployments** using AWS services. It reduces infrastructure costs while ensuring rapid and scalable deployments.

Project 6: Multi-Account CI/CD Pipeline with AWS CodePipeline

Introduction

A **Multi-Account CI/CD pipeline** in AWS using **CodePipeline** enables secure and automated deployments across multiple AWS accounts. This approach follows **best practices** by separating environments like **development, staging, and production** into different AWS accounts. AWS CodePipeline automates the release process, ensuring continuous integration and continuous deployment (**CI/CD**) across these accounts.

Complete Steps to Set Up Multi-Account CI/CD Pipeline

Step 1: Prerequisites

- AWS accounts for **Dev, Staging, and Prod**
- An IAM user with admin access
- A **GitHub/CodeCommit repository** for storing application code
- A **S3 bucket** for storing pipeline artifacts
- AWS CLI configured with access to all accounts

Step 2: Set Up IAM Roles for Cross-Account Access

Since AWS CodePipeline runs in the **Management Account**, it needs permissions to deploy to other AWS accounts.

1. Create an IAM Role in Each Target Account (Staging/Prod)

- Go to **IAM → Roles** in the **Staging/Prod** AWS accounts.
- Click **Create Role → Another AWS Account**.
- Enter the **Account ID** of the management account.
- Attach the **AWSCodeDeployRole** policy.
- Trust policy should allow the management account to assume this role.

json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": { "AWS": "arn:aws:iam::<MANAGEMENT_ACCOUNT_ID>:root" },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

- Save the **Role ARN**, as it will be used in the pipeline.

Step 3: Set Up AWS CodePipeline in Management Account

1. Go to **AWS CodePipeline** in the Management Account.
2. Click **Create Pipeline**.
3. Choose a name and select **New Service Role**.
4. **Source Stage:**
    - Select **GitHub** or **AWS CodeCommit**.
    - Choose the repository and branch.
5. **Build Stage (Optional but Recommended):**
    - Select **AWS CodeBuild**.
    - Define a buildspec.yml file for the build process.

Example buildspec.yml:

```
yaml

version: 0.2

phases:

 install:

  runtime-versions:

   nodejs: 14

 build:

  commands:

   - echo "Building the application..."

   - npm install

 post_build:

  commands:

   - echo "Build completed!"

artifacts:

 files: '**/*'
```

Step 4: Set Up Deployment to Multiple Accounts

1. **Create a Deploy Stage for Staging Account**
    - Add a new **Deploy Stage** in CodePipeline.
    - Choose **AWS CodeDeploy** or **CloudFormation**.
    - Specify the **Role ARN** created in the staging account.
2. **Create a Deploy Stage for Production Account**
    - Repeat the same process, using the **Production Role ARN**.

Step 5: Test the Pipeline

1. Push code changes to the repository.
2. Check if the pipeline triggers the build and deploys across accounts.
3. Validate the deployment in each environment.

Conclusion

This **multi-account AWS CodePipeline** ensures **secure** and **automated CI/CD deployments** across AWS accounts. It follows **AWS best practices**, using IAM roles for cross-account access and AWS CodeDeploy or CloudFormation for safe deployments.

- **Containerized Deployment with AWS Fargate**

Introduction

AWS Fargate is a serverless compute engine for containers that allows you to run and manage Docker containers without managing the underlying infrastructure. It integrates with Amazon ECS (Elastic Container Service) and Amazon EKS (Elastic Kubernetes Service), but here we will focus on ECS with Fargate.

This project will demonstrate deploying a containerized application using AWS Fargate with Amazon ECS.

Steps for Containerized Deployment with AWS Fargate

1. Prerequisites

- AWS Account
- AWS CLI installed and configured (aws configure)
- Docker installed
- An ECS cluster

---

2. Build and Push Docker Image to Amazon ECR

 # Create a repository in Amazon ECR

aws ecr create-repository --repository-name my-fargate-app

 # Authenticate Docker with ECR

aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com

 # Build Docker image

docker build -t my-fargate-app .

 # Tag the image

docker tag my-fargate-app:latest <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/my-fargate-app:latest

 # Push the image to ECR

docker push <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/my-fargate-app:latest

---

3. Create an ECS Cluster

aws ecs create-cluster --cluster-name my-fargate-cluster

---

4. Create Task Definition

Create a JSON file (task-def.json) with the following content:

json

```
{
  "family": "my-fargate-task",
  "networkMode": "awsvpc",
  "requiresCompatibilities": ["FARGATE"],
  "cpu": "256",
```

```
  "memory": "512",
  "executionRoleArn": "arn:aws:iam::AWS_ACCOUNT_ID:role/ecsTaskExecutionRole",
  "containerDefinitions": [
    {
      "name": "my-fargate-app",
      "image": "<AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/my-fargate-app:latest",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80,
          "protocol": "tcp"
        }
      ]
    }
  ]
}
```

Register the task definition:

```
aws ecs register-task-definition --cli-input-json file://task-def.json
```

---

5. Create an ECS Service and Run the Task

```
aws ecs create-service \
  --cluster my-fargate-cluster \
  --service-name my-fargate-service \
  --task-definition my-fargate-task \
  --launch-type FARGATE \
  --network-configuration "awsvpcConfiguration={subnets=[subnet-abc123],securityGroups=[sg-abc123],assignPublicIp=ENABLED}" \
  --desired-count 1
```

---

6. Verify Deployment

Check running tasks:

```
aws ecs list-tasks --cluster my-fargate-cluster
```

Get logs (if using CloudWatch):

```
aws logs describe-log-streams --log-group-name /ecs/my-fargate-app
```

---

7. Clean Up Resources

```
aws ecs delete-service --cluster my-fargate-cluster --service my-fargate-service --force
```

```
aws ecs delete-cluster --cluster my-fargate-cluster
```

aws ecr delete-repository --repository-name my-fargate-app --force

---

Project 8: Blue-Green Deployment on AWS ECS

Introduction:
Blue-Green Deployment is a release management strategy that reduces downtime and risk by running two identical environments, Blue (current version) and Green (new version). In AWS ECS, this ensures smooth application updates without downtime.

Steps to Implement Blue-Green Deployment on AWS ECS

1. Setup AWS Infrastructure

- Create an **ECS Cluster** with Fargate or EC2 launch type.
- Define an **ECS Service** with a Load Balancer.
- Use **AWS CodeDeploy** for automated deployments.
- Configure an **Application Load Balancer (ALB)** to route traffic.

2. Create IAM Roles and Policies

- Ensure ECS, CodeDeploy, and EC2 have appropriate IAM roles.
- Attach AWS-managed policies for ECS, CodeDeploy, and ALB access.

3. Setup Application Load Balancer (ALB)

- Create an ALB with two Target Groups (**Blue** and **Green**).
- Define a listener to forward traffic to the active target group.

4. Deploy the First Version (Blue) on ECS

- Register the **Blue Task Definition** in ECS.
- Deploy the service using the Blue Target Group.
- Ensure the application is working before deploying updates.

5. Configure AWS CodeDeploy for Blue-Green Deployment

- Create an **AWS CodeDeploy Application**.
- Define a **Deployment Group** linked to ECS and ALB.
- Set up CodeDeploy to shift traffic between Blue and Green.

6. Deploy the New Version (Green) Using CodePipeline

- Push updated application code to a GitHub repo.
- Use **AWS CodePipeline** to trigger deployment.
- CodeDeploy updates the ECS service and shifts traffic from Blue to Green.

7. Monitor Deployment & Rollback If Needed

- Use **CloudWatch Logs & Alarms** to monitor health.
- If issues arise, rollback by shifting traffic back to Blue.

Summary of AWS Services Used:

- **Amazon ECS** – Runs containerized applications.
- **AWS CodeDeploy** – Automates Blue-Green deployments.
- **AWS CodePipeline** – Automates deployment workflow.
- **Application Load Balancer (ALB)** – Directs traffic to the right version.
- **Amazon CloudWatch** – Monitors logs and performance.

---

6. DevOps Projects:

Project 1: Infrastructure as Code (IaC) using Terraform on AWS

Introduction

Infrastructure as Code (IaC) is a method of managing and provisioning cloud infrastructure using code. Terraform, developed by HashiCorp, is a popular IaC tool that allows you to define AWS infrastructure in a declarative way. With Terraform, you can create, modify, and delete AWS resources efficiently and consistently.

Project Steps: Deploy an EC2 Instance Using Terraform

Prerequisites

- An **AWS account** with IAM user permissions
- **Terraform installed** on your local system (Install Terraform)
- **AWS CLI installed** and configured (aws configure)

Step 1: Create the Project Directory

sh

```
mkdir terraform-aws-ec2 && cd terraform-aws-ec2
```

Step 2: Create the Terraform Configuration File

Create a file named main.tf and add the following Terraform script:

hcl

```
provider "aws" {
  region = "us-east-1"  # Change as per your requirement
}

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0" # Amazon Linux 2 AMI (Change as per region)
  instance_type = "t2.micro"

  tags = {
    Name = "Terraform-EC2"
  }
}
```

Step 3: Initialize Terraform

Run the following command to initialize Terraform in the project directory:

terraform init

Step 4: Validate Terraform Configuration

Check for any syntax errors before applying changes:

terraform validate

Step 5: Plan the Infrastructure Deployment

See what changes Terraform will make before applying them:

terraform plan

Step 6: Apply Terraform Configuration

Deploy the resources to AWS:

terraform apply -auto-approve

This will create an EC2 instance in AWS.

Step 7: Verify the Deployment

Check the AWS Console (EC2 Dashboard) to confirm that the instance has been created.

Step 8: Destroy the Infrastructure

To delete all the resources created by Terraform:

terraform destroy -auto-approve

Conclusion

This project demonstrated how to use Terraform to deploy an EC2 instance on AWS. Terraform allows automation, consistency, and scalability in cloud infrastructure management

---

Project 2: AWS Auto Scaling with ALB and CloudWatch

Introduction

AWS Auto Scaling enables applications to automatically adjust their compute capacity based on demand, ensuring optimal performance and cost efficiency. Application Load Balancer (ALB) distributes incoming traffic across multiple instances, while Amazon CloudWatch monitors system performance and triggers scaling actions when required.

Project Overview

In this project, we will:

- Deploy an EC2-based application behind an **Application Load Balancer (ALB)**
- Configure an **Auto Scaling Group (ASG)** to dynamically add or remove EC2 instances
- Use **CloudWatch Alarms** to trigger scaling events based on CPU utilization
- Ensure high availability and fault tolerance

Steps to Implement AWS Auto Scaling with ALB and CloudWatch

Step 1: Create a Key Pair

Generate a key pair for SSH access to EC2 instances:

aws ec2 create-key-pair --key-name MyKeyPair --query 'KeyMaterial' --output text > MyKeyPair.pem

chmod 400 MyKeyPair.pem

Step 2: Launch a Base EC2 Instance

Launch an Amazon Linux 2 instance and install a sample web server:

sudo yum update -y

sudo yum install -y httpd

echo "Welcome to Auto Scaling Demo!" | sudo tee /var/www/html/index.html

sudo systemctl start httpd

sudo systemctl enable httpd

1. Create an **Amazon Machine Image (AMI)** from this instance for scaling.

Step 3: Create a Load Balancer (ALB)

1. Go to **EC2 Dashboard → Load Balancers → Create Load Balancer**
2. Choose **Application Load Balancer**
3. Configure:
    - **Listeners:** HTTP (Port 80)
    - **Target Group:** Create a new target group for EC2 instances
    - **Health Check Path:** /index.html
4. **Register Targets:** Leave empty as ASG will add instances dynamically

Step 4: Create a Launch Template

1. Navigate to **EC2 → Launch Templates → Create Launch Template**
2. Configure:
    - **AMI:** Select the AMI created earlier
    - **Instance Type:** t2.micro (free-tier)
    - **Security Group:** Allow HTTP (80) and SSH (22)
    - **Key Pair:** Select the key pair created in Step 1

User Data (Optional): To auto-start the web server, add:

#!/bin/

sudo yum update -y

sudo yum install -y httpd

echo "Welcome to Auto Scaling Demo!" | sudo tee /var/www/html/index.html

sudo systemctl start httpd

sudo systemctl enable httpd

Step 5: Create an Auto Scaling Group (ASG)

1. Navigate to **EC2 → Auto Scaling Groups → Create Auto Scaling Group**
2. Configure:
    - **Launch Template:** Choose the one created earlier
    - **VPC & Subnets:** Select public subnets
    - **Attach to Load Balancer:** Select the target group from Step 3
    - **Desired Capacity:** 2 (initial number of instances)
    - **Min/Max Instances:** Set min = 1, max = 3
    - **Scaling Policies:** Add scaling based on **CPU utilization**
        - **Scale Out Policy:** Add an instance when CPU > 70%
        - **Scale In Policy:** Remove an instance when CPU < 30%

Step 6: Configure CloudWatch Alarms

1. Go to **CloudWatch → Alarms → Create Alarm**
2. Choose **EC2 → Auto Scaling Group → CPU Utilization**
3. Set thresholds:
    - **Scale Out:** CPU > 70% for 5 minutes → Add 1 instance
    - **Scale In:** CPU < 30% for 5 minutes → Remove 1 instance
4. Set notifications (optional) for email alerts

Step 7: Test Auto Scaling

1. **Check Initial Setup:**
    - Visit the ALB DNS in a browser (http://ALB-DNS-NAME)
    - You should see: Welcome to Auto Scaling Demo!

Simulate High Load to Trigger Scaling:
SSH into an instance and run:
yes > /dev/null &

2. This will increase CPU usage and trigger a scale-out event.

Stop Load Test and Observe Scale-In:
pkill yes

3. CloudWatch will detect lower CPU usage and remove extra instances.

Summary

- **ALB** distributes traffic across instances
- **Auto Scaling Group (ASG)** dynamically adjusts instance count
- **CloudWatch** monitors and triggers scaling actions
- **High availability and cost efficiency** achieved

Project 3: AWS Monitoring & Logging using ELK and CloudWatch

Introduction

Monitoring and logging are critical for maintaining system reliability, troubleshooting issues, and optimizing performance. This project integrates **AWS CloudWatch** for infrastructure monitoring and **ELK Stack (Elasticsearch, Logstash, Kibana)** for centralized logging.

AWS CloudWatch collects logs, metrics, and events from AWS services, while ELK Stack provides log aggregation, visualization, and analysis.

Project Steps

1. Setup AWS CloudWatch Monitoring

1. **Enable CloudWatch Logs**
    - Go to **AWS CloudWatch → Logs**
    - Configure log groups and streams for AWS services (e.g., EC2, Lambda, ECS).

Install and configure the CloudWatch agent on EC2 instances:
sudo yum install -y amazon-cloudwatch-agent

sudo amazon-cloudwatch-agent-config-wizard

sudo systemctl start amazon-cloudwatch-agent

    - Verify logs in **CloudWatch Logs**.
2. **Set Up CloudWatch Alarms**
    - Define **metric filters** for logs to track specific patterns (e.g., error messages).
    - Configure **CloudWatch Alarms** to trigger SNS notifications when thresholds are breached.

---

2. Deploy ELK Stack on AWS

Option 1: Using AWS Managed OpenSearch (formerly Elasticsearch)

1. **Create an OpenSearch Domain** in AWS.
2. **Enable log ingestion from AWS services** (e.g., CloudWatch, S3).
3. **Access OpenSearch Dashboards** for visualization.

Option 2: Self-Hosted ELK on EC2

1. **Launch an EC2 instance** (Ubuntu 20.04).

Install Java & Download ELK Stack
sudo apt update && sudo apt install -y openjdk-11-jdk

wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.17.9-linux-x86_64.tar.gz

wget https://artifacts.elastic.co/downloads/logstash/logstash-7.17.9-linux-x86_64.tar.gz

wget https://artifacts.elastic.co/downloads/kibana/kibana-7.17.9-linux-x86_64.tar.gz

Configure Elasticsearch (/etc/elasticsearch/elasticsearch.yml)
yaml

network.host: 0.0.0.0

discovery.type: single-node

Start Elasticsearch:
sudo systemctl start elasticsearch

Configure Logstash (/etc/logstash/conf.d/logstash.conf)

```
input {
  file {
    path => "/var/log/syslog"
    start_position => "beginning"
  }
}
output {
 elasticsearch {
   hosts => ["http://localhost:9200"]
   index => "logs"
 }
}
```

Start Logstash:
sudo systemctl start logstash

Configure Kibana (/etc/kibana/kibana.yml)

server.host: "0.0.0.0"

Start Kibana:
sudo systemctl start kibana

---

3. Integrate CloudWatch with ELK

1. **Export CloudWatch Logs to S3**
    - In CloudWatch, create a log subscription filter to send logs to an **S3 bucket**.
2. **Use Logstash to Ingest S3 Logs**
    - Configure Logstash S3 input plugin to fetch logs from S3.

```
input {
 s3 {
   bucket => "your-bucket-name"
```

```
  access_key_id => "AWS_ACCESS_KEY"
  secret_access_key => "AWS_SECRET_KEY"
 }
}
```

---

4. Data Visualization in Kibana

1. **Access Kibana** at http://<EC2-IP>:5601.
2. **Create an index pattern** (logs-*) to visualize logs.
3. **Build dashboards** for log analysis and monitoring.

---

5. Set Up Alerts & Notifications

1. **Configure OpenSearch/Kibana Alerts** for log-based alerts.
2. **Integrate with AWS SNS** to send alerts via email or SMS.

Final Steps

- Validate log ingestion from CloudWatch to ELK.
- Set up **Grafana** (optional) for advanced visualization.
- Ensure **security best practices** (IAM roles, VPC, encryption).

This setup provides **real-time monitoring, centralized logging, and alerting** for AWS infrastructure.

Project 4: Automated Patch Management with AWS SSM

Introduction

Automated Patch Management is crucial for maintaining system security and compliance by keeping servers up to date with the latest patches. AWS Systems Manager (SSM) provides **Patch Manager**, which automates patching for Amazon EC2 instances and on-premises servers. This ensures that critical security updates are applied efficiently without manual intervention.

Steps to Set Up Automated Patch Management with AWS SSM

1. Prerequisites

- An **AWS account** with the necessary permissions
- Amazon **EC2 instances** (Windows/Linux)
- **IAM roles** for SSM
- **AWS Systems Manager Agent (SSM Agent)** installed on EC2 instances

---

2. Attach an IAM Role to EC2

1. **Create an IAM Role** with the following policies:
    - AmazonSSMManagedInstanceCore
    - AmazonSSMPatchAssociation
2. Attach this IAM role to your EC2 instances.

---

3. Verify SSM Agent Installation

For Amazon Linux and Ubuntu:
sudo systemctl status amazon-ssm-agent

For Windows:
Get-Service AmazonSSMAgent

If not installed, install using:
sudo yum install -y amazon-ssm-agent

---

4. Configure Patch Baselines

1. Go to **AWS Systems Manager → Patch Manager**.
2. Choose **Patch Baselines**.
3. Create a **custom baseline** or use the **default** one.
4. Define:
    - **OS Type** (Amazon Linux, Ubuntu, Windows, etc.)
    - **Approved patches** and compliance rules.
    - **Rejection criteria** (e.g., exclude specific patches).

---

5. Create a Patch Group

1. Tag EC2 instances with:
    - Key: PatchGroup
    - Value: Production (or any group name)
2. Assign the **patch baseline** to this Patch Group.

---

6. Schedule Patch Deployment Using Maintenance Windows

1. In **AWS Systems Manager**, go to **Maintenance Windows**.
2. Click **Create Maintenance Window** and configure:
    - **Name** (e.g., Patch-Window)
    - **Schedule** (e.g., every Sunday at 2 AM UTC)
    - **Duration & Stop Time** settings.
3. Add **targets** (EC2 instances tagged with PatchGroup=Production).
4. Define **patching tasks**:
    - **Register a task** → Choose **AWS-RunPatchBaseline**.
    - Set Operation as **Scan & Install**.
    - Define **Rate Control** (Concurrency, Error Threshold).

---

7. Monitor Patch Compliance

1. Go to **AWS Systems Manager → Patch Manager → Compliance**.
2. Review **patched/non-patched instances**.
3. Use **AWS CloudWatch Logs** to track patch execution.

---

8. Automate Notifications (Optional)

- Use **Amazon SNS** to send patching alerts.
- Configure **AWS Lambda** to trigger notifications for failed patches.

Conclusion

AWS SSM Patch Manager simplifies patching by automating security updates for EC2 instances, ensuring compliance with security policies. By following these steps, you can efficiently manage patching operations with minimal manual effort.

Project 5: AWS Security Hardening with IAM & Config Rules

AWS security hardening ensures a robust security posture by implementing least privilege access with **IAM** (Identity and Access Management) and continuous compliance monitoring with **AWS Config Rules**. This approach helps enforce security policies, detect misconfigurations, and prevent unauthorized access.

Project Steps

1. Set Up IAM Best Practices

- **Create IAM Users & Groups**: Avoid using root user; create separate users with specific permissions.
- **Use IAM Roles**: Assign temporary credentials instead of static access keys.
- **Enable MFA**: Require Multi-Factor Authentication for privileged users.
- **Implement Least Privilege**: Use IAM policies with only required permissions.
- **Restrict Root User Access**: Disable root API access keys if present.
- **Set IAM Password Policy**: Enforce strong passwords with expiration and reuse restrictions.

2. Define AWS Config Rules for Compliance

- **Enable AWS Config**: Turn on AWS Config to track changes in AWS resources.
- **Create Managed Config Rules**:
  - iam-user-mfa-enabled: Ensures IAM users have MFA enabled.
  - restricted-ssh: Blocks unrestricted SSH (0.0.0.0/0) access.
  - s3-bucket-public-read-prohibited: Prevents public access to S3 buckets.
  - cloudtrail-enabled: Ensures AWS CloudTrail is active for auditing.
  - ec2-instance-no-public-ip: Blocks EC2 instances from having public IPs.
- **Set Up Custom Config Rules** (Optional): Use AWS Lambda for custom checks.

3. Enforce Security with AWS Config & IAM

- **Auto-Remediation with AWS Lambda**: Create a Lambda function to remediate security violations automatically.
- **Enable AWS Security Hub**: Get security insights and recommendations.
- **Monitor Logs with CloudTrail & CloudWatch**:
  - Enable AWS CloudTrail for audit logs.
  - Set up CloudWatch Alarms for suspicious activities.
- **Use AWS Organizations SCPs**: Define Service Control Policies to restrict access globally.

4. Testing & Validation

- Simulate IAM policy changes using **IAM Policy Simulator**.
- Trigger AWS Config evaluations manually to check compliance.
- Validate security policies using AWS **Access Analyzer**.
- Review CloudTrail logs for unauthorized access attempts.

Outcome

By implementing **IAM security best practices** and **AWS Config Rules**, this project ensures a **secure AWS environment**, reduces risks of **misconfigurations**, and enhances compliance with **security standards** like **CIS AWS Benchmark**.

Project 6: Disaster Recovery Setup with AWS Backup & Route 53

Introduction

Disaster Recovery (DR) on AWS ensures business continuity in case of failures, cyber-attacks, or natural disasters. This project focuses on using AWS Backup to automate backup processes and Route 53 for DNS failover to redirect traffic in case of failure.

Steps to Implement Disaster Recovery on AWS

Step 1: Set Up AWS Backup

1. **Create a Backup Plan:**
   - Go to AWS Backup → Backup Plans → Create a Backup Plan
   - Choose "Build a new plan" → Define the backup frequency (e.g., daily, weekly)
   - Select retention policy (e.g., 30 days)
2. **Add Resources to Backup Plan:**
   - Attach AWS services like EC2, RDS, EBS, DynamoDB, etc.

    - Configure IAM roles for backup access
3. **Create an On-Demand Backup (Optional):**
    - Select the resource → Click "Create Backup" → Set retention and storage
4. **Configure Backup Vault:**
    - AWS Backup automatically stores backups in a vault
    - Encrypt backups for additional security

Step 2: Set Up Route 53 for Failover

1. **Create a Route 53 Hosted Zone:**
    - Navigate to Route 53 → Hosted Zones → Create a new hosted zone
2. **Set Up Health Checks:**
    - Go to Route 53 → Health Checks → Create Health Check
    - Configure health checks for the primary application server (e.g., HTTP, TCP)
3. **Configure DNS Failover Policy:**
    - Create two record sets:
        - Primary record: Points to the main application (e.g., ALB or EC2 instance)
        - Secondary record: Points to a DR instance in another region
    - Enable Failover Routing Policy
    - Attach the health check to the primary record

Step 3: Restore & Validate Backup in Disaster Scenario

1. Simulate a failure by stopping the primary instance
2. **Restore the latest backup from AWS Backup:**
    - Go to AWS Backup → Recovery Points → Restore Resource
3. **Check if Route 53 failover works:**
    - Route 53 should automatically redirect traffic to the secondary DR instance

Conclusion

By automating backups with AWS Backup and using Route 53 for DNS failover, you ensure quick recovery in case of disasters. This setup minimizes downtime and ensures business continuity.

Project 7: Cost Optimization using AWS Lambda to Stop Unused EC2s

Introduction

AWS Lambda can help reduce costs by automatically stopping idle or unused EC2 instances. This solution involves using AWS Lambda with CloudWatch and IAM to detect and shut down instances that are not in use, ensuring cost savings.

Steps to Implement

1. Create an IAM Role for Lambda

- Go to **IAM > Roles > Create Role**
- Choose **AWS service** and select **Lambda**
- Attach the **AmazonEC2FullAccess** policy
- Attach **CloudWatchLogsFullAccess** (for logging)
- Name the role **Lambda-StopEC2-Role** and create it

---

2. Create an AWS Lambda Function

- Go to **AWS Lambda > Create Function**
- Select **Author from scratch**
- Set:
    - **Function name**: StopUnusedEC2Instances
    - **Runtime**: Python 3.x
    - **Role**: Select **Lambda-StopEC2-Role**
- Click **Create Function**

---

3. Add Python Code to Stop Unused EC2s

- In the **Function Code** section, replace the default code with:

```
python

import boto3


def lambda_handler(event, context):
    ec2 = boto3.client('ec2')


    # Get all running instances
    response = ec2.describe_instances(
        Filters=[{'Name': 'instance-state-name', 'Values': ['running']}]
    )


    instances_to_stop = []


    for reservation in response['Reservations']:
        for instance in reservation['Instances']:
            instances_to_stop.append(instance['InstanceId'])


    if instances_to_stop:
        ec2.stop_instances(InstanceIds=instances_to_stop)
        print(f"Stopping instances: {instances_to_stop}")
    else:
        print("No instances to stop.")
```

- Click **Deploy**

---

4. Create a CloudWatch Event Rule (Trigger for Lambda)

- Go to **Amazon EventBridge (CloudWatch Events) > Rules > Create Rule**
- Name it **StopUnusedEC2Rule**
- Select **Schedule** and set it to run **every 1 hour** (or as required)
- Choose **Lambda Function** as the target and select **StopUnusedEC2Instances**
- Click **Create**

Testing the Lambda Function

- Manually start some EC2 instances
- Go to Lambda and **Test** the function
- Check **EC2 console** to see if instances are stopped

Conclusion

This solution helps optimize costs by automatically stopping unused EC2 instances. You can modify it to exclude specific instances, send notifications via SNS, or extend it to start instances at scheduled times.

Project 8: Self-Healing Infrastructure with AWS Auto Scaling

Introduction

Self-healing infrastructure in AWS ensures high availability and fault tolerance by automatically detecting and recovering from failures without manual intervention. AWS Auto Scaling helps maintain optimal performance by adjusting the number of EC2 instances in response to demand. It works alongside Elastic Load Balancer (ELB) and Amazon CloudWatch to monitor instance health and automatically replace unhealthy instances.

Steps to Implement Self-Healing Infrastructure with AWS Auto Scaling

1. Set Up an AWS EC2 Auto Scaling Group

- Launch an EC2 instance and configure it with the required application setup.
- Create an Amazon Machine Image (AMI) of this instance for auto-scaling.

2. Create a Launch Template or Launch Configuration

- Go to the **EC2 Dashboard → Auto Scaling → Launch Templates**.
- Create a launch template using the previously created AMI and define instance type, security group, and key pair.

3. Configure an Auto Scaling Group (ASG)

- Go to **Auto Scaling Groups** and create a new group.
- Attach the **Launch Template** to the ASG.
- Define the desired number of instances, minimum and maximum scaling limits.
- Select the **VPC and subnets** for instance deployment.

4. Configure Elastic Load Balancer (ELB) (Optional but Recommended)

- Go to **EC2 Dashboard → Load Balancers → Create Load Balancer**.
- Choose **Application Load Balancer (ALB)** and configure the target group.
- Attach the Auto Scaling Group to the Load Balancer.

5. Set Up Health Checks for Self-Healing

- In the Auto Scaling Group, define **EC2 health checks** and **ELB health checks** to detect unhealthy instances.
- Enable the **Replace Unhealthy Instances** option to automatically replace failed instances.

6. Configure Auto Scaling Policies with CloudWatch Metrics

- Navigate to **Auto Scaling Group → Scaling Policies**.
- Choose a scaling policy (e.g., **Target Tracking**, **Step Scaling**, or **Scheduled Scaling**).
- Set up CloudWatch alarms to trigger scaling actions based on metrics like **CPU utilization**, **network traffic**, or **request count**.

7. Test the Self-Healing Mechanism

- Manually terminate an instance in the Auto Scaling Group and observe how AWS automatically launches a replacement.
- Stress test the application to see how Auto Scaling responds to demand changes.

8. Monitor and Optimize Scaling

- Use **AWS CloudWatch** to monitor performance and adjust scaling policies as needed.
- Enable **AWS Auto Scaling Notifications** using **Amazon SNS** for alerts on scaling events.

Conclusion

This setup ensures that application instances are automatically replaced if they fail, and resources scale dynamically based on demand. By leveraging **Auto Scaling,**

ELB, and CloudWatch, AWS enables a highly available and resilient self-healing infrastructure.

---

7. Database & Storage Solutions

Project 1: MySQL RDS Deployment – Set up and manage an RDS database.

Introduction:

Amazon RDS (Relational Database Service) is a managed database service that simplifies the deployment, scaling, and maintenance of relational databases in the cloud. In this project, we will set up a MySQL database on Amazon RDS, configure security settings, and connect it to an application or local system.

Steps to Deploy MySQL on AWS RDS:

Step 1: Log in to AWS Console

- Navigate to AWS Management Console and log in.

Step 2: Create an RDS Instance

1. Open the **RDS** service.
2. Click **Create database**.
3. Select **Standard create**.
4. Choose **MySQL** as the database engine.
5. Select the **Free Tier** (for testing) or a suitable instance type.
6. Set the **DB instance identifier** (e.g., my-mysql-db).
7. Enter **Master username** and **password**.
8. Choose the **db.t3.micro** instance type (or as per requirements).
9. Set **Storage** (e.g., 20 GB, enable auto-scaling if needed).
10. Select **Multi-AZ deployment** (if high availability is required).
11. Set **Public access** to **Yes** (if you want to connect from external sources).
12. Choose the **VPC, Subnet, and Security Group**.
13. Enable automatic backups and monitoring if needed.
14. Click **Create database**.

Step 3: Configure Security Group

1. Navigate to **EC2 > Security Groups**.
2. Select the **Security Group** associated with the RDS instance.
3. Click **Inbound rules > Edit inbound rules**.
4. Add a new rule:
    - **Type:** MySQL/Aurora
    - **Port:** 3306
    - **Source:** Your IP (My IP) or 0.0.0.0/0 (for open access, not recommended for production).
5. Save the rules.

Step 4: Get the RDS Endpoint

1. Go to **RDS > Databases**.
2. Click on your database.
3. Copy the **Endpoint** (e.g., mydb.abcdefg1234.us-**east**-1.rds.amazonaws.com).

Step 5: Connect to MySQL RDS

From your local machine or EC2 instance, install MySQL client:
sudo apt update && sudo apt install mysql-client -y

Connect to the RDS instance using:
mysql -h mydb.abcdefg1234.us-east-1.rds.amazonaws.com -u admin -p

Step 6: Create a Database and Table

sql

```
CREATE DATABASE myappdb;

USE myappdb;


CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    email VARCHAR(100) UNIQUE
);
```

Step 7: Integrate with an Application

Update the database connection string in your application:
mysql://admin:password@mydb.abcdefg1234.us-east-1.rds.amazonaws.com:3306/myappdb

- Test the connection from your application.

Step 8: Enable Monitoring & Backups

- Enable **Enhanced Monitoring** in **RDS > Monitoring**.
- Configure **Automated Backups** for retention and snapshots.

Step 9: Scaling & Performance Optimization

- Modify the instance size if needed (Modify option in RDS).
- Enable **Read Replicas** for scaling reads.

Step 10: Clean Up Resources (Optional)

If no longer needed, delete the database to avoid charges:

- Go to **RDS > Databases** > Select your DB > Click **Delete**.
- Take a final snapshot if required.

Conclusion:

This project demonstrates how to deploy, secure, and manage a MySQL database on AWS RDS. With monitoring, scaling, and backup strategies, your database remains highly available and optimized for production use.

Project 2: DynamoDB CRUD Operations with Lambda – Create a serverless backend with DynamoDB.

Introduction:

In this project, you will create a **serverless backend** using AWS **Lambda** to perform **CRUD (Create, Read, Update, Delete) operations** on an **Amazon DynamoDB** table. AWS API Gateway will expose RESTful APIs, triggering Lambda functions to interact with DynamoDB. This setup ensures a **fully serverless and scalable** backend without managing any servers.

Steps to Implement:

1. Create a DynamoDB Table

- Go to the AWS Management Console → DynamoDB → Create Table
- Set **Table Name**: UsersTable
- Set **Primary Key**: userId (String)
- Enable On-Demand Capacity
- Click **Create Table**

---

2. Create an IAM Role for Lambda

- Go to AWS IAM → Roles → Create Role
- Select **AWS Service** → Choose **Lambda**
- Attach policies:
  - **AmazonDynamoDBFullAccess**
  - **AWSLambdaBasicExecutionRole**
- Click **Create Role**

---

3. Create AWS Lambda Functions for CRUD Operations

Create Lambda for "Create User" (Insert Data)

- Go to **AWS Lambda** → Create Function
- Choose **Author from scratch**
- Set **Function Name**: CreateUserFunction
- Choose **Runtime**: Python 3.x / Node.js
- Assign the IAM Role created earlier
- Add this code:

Python Code for Create User (create_user.py):

```
python

import json

import boto3

import uuid


dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('UsersTable')


def lambda_handler(event, context):

    data = json.loads(event['body'])

    user_id = str(uuid.uuid4())

    item = {

        'userId': user_id,

        'name': data['name'],

        'email': data['email']

    }

    table.put_item(Item=item)

    return {

        'statusCode': 201,

        'body': json.dumps({'message': 'User created', 'userId': user_id})

    }
```

- Click **Deploy**

Create Lambda for "Read User" (Fetch Data)

Python Code for Read User (get_user.py):

```
python


import json

import boto3
```

```
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('UsersTable')

def lambda_handler(event, context):
    user_id = event['pathParameters']['userId']
    response = table.get_item(Key={'userId': user_id})
    if 'Item' in response:
        return {'statusCode': 200, 'body': json.dumps(response['Item'])}
    return {'statusCode': 404, 'body': json.dumps({'message': 'User not found'})}
```

Create Lambda for "Update User"

Python Code for Update User (update_user.py):

python

```
import json
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('UsersTable')

def lambda_handler(event, context):
    user_id = event['pathParameters']['userId']
    data = json.loads(event['body'])

    table.update_item(
        Key={'userId': user_id},
        UpdateExpression='SET #n = :name, email = :email',
        ExpressionAttributeNames={'#n': 'name'},
        ExpressionAttributeValues={':name': data['name'], ':email': data['email']}
    )

    return {'statusCode': 200, 'body': json.dumps({'message': 'User updated'})}
```

Create Lambda for "Delete User"

Python Code for Delete User (delete_user.py):

python

```
import json
import boto3
```

```
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('UsersTable')


def lambda_handler(event, context):

    user_id = event['pathParameters']['userId']

    table.delete_item(Key={'userId': user_id})

    return {'statusCode': 200, 'body': json.dumps({'message': 'User deleted'})}
```

---

4. Configure API Gateway for REST API

- Go to **API Gateway** → Create **REST API**
- Create resources: /users and /users/{userId}
- Link each API method (GET, POST, PUT, DELETE) to the respective Lambda function
- Deploy API

---

5. Test Your APIs

Use **Postman** or **curl** to test API endpoints:

- **Create User:** POST /users
- **Get User:** GET /users/{userId}
- **Update User:** PUT /users/{userId}
- **Delete User:** DELETE /users/{userId}

Conclusion:

You have successfully built a **serverless backend** using **AWS Lambda, DynamoDB, and API Gateway** for CRUD operations. This architecture is **highly scalable, cost-effective, and maintenance-free**.

Project 3: Data Warehouse with AWS Redshift – Store and analyze structured data in Redshift.

1. Prerequisites

- An AWS account
- IAM role with required permissions
- Basic knowledge of SQL and AWS services

---

2. Create an AWS Redshift Cluster

1. **Sign in to AWS Console** and go to **Amazon Redshift**.
2. Click on **Create Cluster** and configure the following:
    - **Cluster Identifier**: Give a unique name.
    - **Node Type**: Choose an instance type (e.g., dc2.large for testing).
    - **Number of Nodes**: Start with 1 node for demo purposes.
    - **Database Name**: Define a name for the Redshift database.
    - **Master Username & Password**: Set credentials for database access.
3. Choose **Public or Private access** based on network configuration.
4. Enable **Enhanced VPC Routing** if needed.
5. Click **Create Cluster** and wait for the cluster to be available.

---

3. Configure Security Group & IAM Role

1. **Modify Security Group** to allow access from your IP.
2. **Attach IAM Role** with AmazonS3ReadOnlyAccess if loading data from S3.

---

4. Connect to Redshift Using SQL Client

1. Download **SQL Workbench/J** or **DBeaver**.
2. Use **JDBC/ODBC driver** to connect to the Redshift endpoint.

Run the following SQL command to test the connection:
sql

```
SELECT version();
```

---

5. Load Data into Redshift

Using Amazon S3 (Recommended)

1. Upload a CSV file to an **S3 bucket**.

Use the COPY command to load data into Redshift:
sql

```
COPY my_table
FROM 's3://your-bucket-name/data.csv'
IAM_ROLE 'arn:aws:iam::account-id:role/your-redshift-role'
CSV
IGNOREHEADER 1;
```

---

6. Run Queries & Analyze Data

Create a table:
sql

```
CREATE TABLE sales_data (
    id INT,
    product_name VARCHAR(255),
    price DECIMAL(10,2),
    quantity INT,
    sale_date DATE
);
```

Run analytical queries:
sql

```
SELECT product_name, SUM(price * quantity) AS total_sales
FROM sales_data
GROUP BY product_name
ORDER BY total_sales DESC;
```

---

7. Visualize Data in Amazon QuickSight *(Optional)*

1. Sign in to **Amazon QuickSight**.
2. Connect Redshift as a **new data source**.
3. Create **dashboards and reports** for analysis.

8. Clean Up Resources ***(To avoid costs)***

1. Delete the Redshift cluster.
2. Remove IAM roles and policies.
3. Delete the S3 bucket if no longer needed.

Conclusion

By following these steps, you have successfully created an AWS Redshift data warehouse, loaded structured data, and analyzed it using SQL queries. This setup is scalable for enterprise data analytics.

Project 4: ETL with AWS Glue – Transform data from S3 to Redshift using AWS Glue.

Introduction

AWS Glue is a fully managed Extract, Transform, Load (ETL) service that helps prepare and move data from various sources to a data warehouse like Amazon Redshift. In this project, we will use AWS Glue to extract data stored in Amazon S3, transform it using AWS Glue ETL scripts, and load it into an Amazon Redshift table.

Steps to Set Up the ETL Pipeline

1. Prerequisites

- AWS Account
- IAM roles with necessary permissions for AWS Glue, S3, and Redshift
- Amazon S3 bucket with sample data (CSV or JSON)
- Amazon Redshift cluster

---

2. Create and Configure Amazon Redshift

1. Go to the **AWS Redshift Console** → Click **Create Cluster**.
2. Choose a cluster name, node type, and database name.
3. In **Cluster Permissions**, attach an IAM role that allows S3 access.

Create a **Schema & Table** in Redshift:
sql

```
CREATE TABLE sales_data (
    id INT,
    customer_name VARCHAR(100),
    amount FLOAT,
    date DATE
);
```

4. Save the **JDBC URL** of the Redshift cluster for AWS Glue connection.

---

3. Upload Sample Data to S3

1. Create an S3 bucket (e.g., s3://my-glue-etl-bucket/).
2. Upload a sample CSV file (e.g., sales_data.csv).

---

4. Create an AWS Glue Crawler

1. Navigate to the **AWS Glue Console → Crawlers → Create Crawler**.
2. Add the S3 path (s3://my-glue-etl-bucket/).

3. Choose a database to store metadata.
4. Run the crawler to populate AWS Glue Data Catalog.

---

5. Create an AWS Glue Connection to Redshift

1. Go to **AWS Glue → Connections → Add Connection**.
2. Choose **Amazon Redshift** and provide the JDBC URL.
3. Enter the **database name**, **user**, and **password**.
4. Test the connection.

---

6. Create an AWS Glue ETL Job

1. Go to **AWS Glue → Jobs → Create Job**.
2. Choose a **Glue IAM Role** with permissions for S3 and Redshift.
3. Select **Source: AWS Glue Data Catalog (S3 table)**.
4. Select **Target: Amazon Redshift Table**.
5. Write a PySpark transformation script (example below).

---

7. AWS Glue ETL Script (PySpark Example)

python

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job

args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

# Read data from S3
s3_df = glueContext.create_dynamic_frame.from_catalog(
    database="my_glue_database",
    table_name="sales_data"
)

# Convert DynamicFrame to DataFrame for transformation
df = s3_df.toDF()
df = df.withColumnRenamed("id", "customer_id")

# Convert back to DynamicFrame
transformed_df = DynamicFrame.fromDF(df, glueContext)
```

 # Write data to Redshift

```
glueContext.write_dynamic_frame.from_jdbc_conf(
    frame=transformed_df,
    catalog_connection="redshift_connection",
    connection_options={
        "dbtable": "sales_data",
        "database": "dev"
    }
)

job.commit()
```

---

8. Run and Monitor the ETL Job

1. Click **Run Job** in AWS Glue Console.
2. Monitor logs in **CloudWatch Logs**.

---

9. Verify Data in Redshift

Connect to Redshift using **Query Editor** and run:

sql

```
SELECT * FROM sales_data;
```

Conclusion

You have successfully set up an **ETL pipeline** using AWS Glue to extract data from **Amazon S3**, transform it using **PySpark**, and load it into **Amazon Redshift**.

Project 5: Graph Database with Amazon Neptune – Store relational data in a graph-based model.

Introduction

Amazon Neptune is a managed graph database service that supports both **property graph (Gremlin)** and **RDF graph (SPARQL)** models. It enables efficient storage and querying of highly connected data, making it suitable for applications like recommendation engines, fraud detection, and social networks.

In this project, we will:

- Set up an **Amazon Neptune** cluster
- Upload **relational data** to Neptune in a **graph-based model**
- Query the data using **Gremlin**

Steps to Set Up the Graph Database

1. Prerequisites

- AWS Account
- IAM role with Amazon Neptune permissions
- Amazon VPC with a private subnet
- Amazon Neptune Cluster

---

2. Create an Amazon Neptune Cluster

1. Go to **AWS Console → Neptune → Create Database**
2. Select **Engine Version** (latest recommended)
3. Choose **Instance Type** (e.g., db.r5.large)
4. Ensure the cluster is in a **private VPC subnet**
5. Click **Create Cluster**

---

3. Configure IAM Role for Neptune Access

1. Go to **IAM Console → Roles → Create Role**
2. Attach **AmazonNeptuneFullAccess** policy
3. Attach the role to your **Neptune Cluster**

---

4. Upload Relational Data in Graph Format

1. Convert relational data to **Gremlin CSV Format**:

Nodes (vertices) file (nodes.csv):

~id,~label,name,age

1,Person,Alice,30

2,Person,Bob,35

3,Person,Charlie,25

Edges (relationships) file (edges.csv):
css

~id,~from,~to,~label,since

101,1,2,knows,2015

102,2,3,knows,2018

2. Upload these files to an **Amazon S3 bucket**.

---

5. Load Data into Neptune

Use the following command to load data into Neptune:

aws neptune-db load \

--source s3://my-neptune-data/nodes.csv \

--format csv \

--iam-role arn:aws:iam::123456789012:role/NeptuneLoadRole

Repeat for the edges file:

aws neptune-db load \

--source s3://my-neptune-data/edges.csv \

--format csv \

--iam-role arn:aws:iam::123456789012:role/NeptuneLoadRole

---

6. Connect to Amazon Neptune

Use **Gremlin Console** to connect to Neptune:

gremlin> :remote connect tinkerpop.server conf/neptune-remote.yaml

gremlin> :remote console

---

7. Query the Graph Using Gremlin

Get all persons:

g.V().hasLabel('Person').values('name')

Find people Alice knows:
g.V().has('name', 'Alice').out('knows').values('name')

Find shortest path between Alice and Charlie:
g.V().has('name', 'Alice').repeat(out()).until(has('name', 'Charlie')).path()

Conclusion

You have successfully set up **Amazon Neptune** as a graph database, imported relational data, and queried it using **Gremlin**. This approach allows for **faster and more complex relationship-based queries** compared to traditional relational databases.

Project 6: Automated S3 Data Archival – Use S3 lifecycle rules to move data to Glacier.

Introduction

Amazon S3 (Simple Storage Service) provides lifecycle policies that allow automatic data archival and deletion. This project sets up an automated process where data stored in an S3 bucket is automatically moved to Amazon S3 Glacier for cost-effective long-term storage.

Steps to Implement

Step 1: Create an S3 Bucket

1. Sign in to the AWS Management Console.
2. Navigate to **Amazon S3 → Create bucket**.
3. Provide a unique bucket name (e.g., my-archival-bucket).
4. Choose a region and configure other settings as needed.
5. Click **Create bucket**.

Step 2: Upload Sample Data

1. Open the created S3 bucket.
2. Click **Upload**, select some files, and confirm the upload.

Step 3: Create an S3 Lifecycle Rule

1. In the S3 bucket, go to the **Management** tab.
2. Click **Create lifecycle rule**.
3. Enter a rule name (e.g., move-to-glacier).
4. Choose **Apply to all objects in the bucket** or select a specific prefix.
5. Under **Lifecycle rule actions**, select **Move current versions of objects between storage classes**.
6. Configure the transition:
    - **Move to Glacier Instant Retrieval after X days** (e.g., 30 days).
    - **Move to Glacier Deep Archive after Y days** (e.g., 90 days) for further cost optimization.
7. Click **Create rule**.

Step 4: Verify the Lifecycle Rule

1. Wait for the configured time period (or adjust settings to test quickly).

2. Check object storage class changes in the **Objects** tab (should show **Glacier** after transition).

Step 5: Restore Archived Data (Optional)

1. Select an object stored in Glacier.
2. Click **Initiate restore** → Choose retrieval option (e.g., Standard, Expedited, or Bulk).
3. Set the duration for temporary access and confirm the restore request.

Conclusion

This project automates data archival using S3 lifecycle rules, reducing storage costs by moving older data to Amazon Glacier while keeping it retrievable when needed.

Project 7: Automated Database Migration – Use AWS Database Migration Service (DMS) to migrate a database from MySQL to Aurora.

Automated Database Migration with AWS DMS

AWS Database Migration Service (AWS DMS) helps migrate databases to AWS quickly and securely with minimal downtime. In this project, we will migrate a **MySQL** database to **Amazon Aurora** using AWS DMS.

Steps to Migrate MySQL to Aurora Using AWS DMS

1. Set Up the Source MySQL Database

- Ensure your MySQL database is running on **Amazon RDS** or an **on-premises** instance.

Enable **binary logging** on MySQL by adding the following parameters to my.cnf or my.ini:
ini

[mysqld]

log_bin=mysql-bin

binlog_format=row

expire_logs_days=1

Restart MySQL and create a user for DMS with necessary privileges:

CREATE USER 'dms_user'@'%' IDENTIFIED BY 'password';

GRANT SELECT, RELOAD, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'dms_user'@'%';

FLUSH PRIVILEGES;

2. Create an Amazon Aurora Target Database

- Launch an **Amazon Aurora MySQL-Compatible** instance in the same region as your MySQL database.
- Note the **endpoint** and database **credentials** for later use.

3. Create an AWS DMS Replication Instance

- Navigate to **AWS DMS Console → Replication Instances → Create replication instance**.
- Choose a **suitable instance class** (e.g., dms.r5.large).
- Ensure it has access to both MySQL and Aurora via **VPC, Security Groups, and Subnet Groups**.

4. Configure Source and Target Endpoints

- **Source Endpoint (MySQL)**
  - Select **MySQL** as the database engine.
  - Provide MySQL **server endpoint, port, database name, username, and password**.
  - Test the connection.
- **Target Endpoint (Aurora)**
  - Select **Amazon Aurora (MySQL-Compatible)** as the engine.
  - Provide the **Aurora endpoint, port, username, and password**.
  - Test the connection.

5. Create a Database Migration Task

- Go to **Database Migration Tasks → Create Task**.
- Choose the **Replication Instance** and the **source & target endpoints**.
- Select **Migration Type**:
  - **Full Load**: For one-time migration.
  - **Full Load + CDC (Change Data Capture)**: For real-time migration with ongoing changes.
- Enable **Mapping Rules** if needed (e.g., schema mapping, column transformations).
- Start the task and monitor progress.

6. Monitor Migration and Validate Data

- Check the migration status in **AWS DMS Console** under **Tasks**.
- Validate data consistency between **MySQL and Aurora** using SQL queries.
- If using **CDC**, ensure changes are replicated in real-time.

7. Cut Over and Final Validation

- Stop writes to the MySQL source database.
- Ensure the **final synchronization** is completed.
- Redirect application traffic to **Aurora**.

8. Clean Up Resources (Optional)

- Delete the **DMS replication instance** if no longer needed.
- Terminate **MySQL RDS instance** if you are fully migrated to Aurora.

Key Benefits of Using AWS DMS

✔ **Minimal Downtime** – Supports CDC for live migration.
✔ **Automated Schema Conversion** – Converts database objects automatically.
✔ **Supports Heterogeneous Migrations** – Can migrate between different database engines.
✔ **Scalability & Reliability** – Fully managed and highly available.

---

8. Monitoring & Logging

Project 1: Monitor AWS Resources with CloudWatch & SNS – Set up alerts for EC2, RDS, and S3.

Introduction

Amazon CloudWatch is a monitoring and observability service that helps track AWS resource performance and operational health. By integrating CloudWatch with Amazon SNS (Simple Notification Service), you can set up alerts for AWS services like **EC2, RDS, and S3**, ensuring proactive monitoring and incident response.

Complete Steps

Step 1: Set Up CloudWatch Alarms for EC2, RDS, and S3

1. Create a CloudWatch Alarm for EC2

1. Go to **AWS Console > CloudWatch > Alarms**
2. Click **Create Alarm**
3. Select **EC2 metric** (e.g., CPU Utilization)

4. Set a threshold (e.g., **CPU Utilization > 80% for 5 minutes**)
5. Choose **Amazon SNS topic** for notifications
6. Click **Create Alarm**

2. Create a CloudWatch Alarm for RDS

1. Go to **CloudWatch > Alarms**
2. Click **Create Alarm**
3. Select **RDS Metrics** (e.g., FreeStorageSpace, CPUUtilization)
4. Set a threshold (e.g., **Free Storage < 1GB**)
5. Attach the **SNS topic**
6. Click **Create Alarm**

3. Create a CloudWatch Alarm for S3

1. Go to **S3 > Select Bucket > Properties**
2. Enable **CloudTrail** to track events
3. In **CloudWatch > Logs**, create a metric filter
4. Define conditions (e.g., **Unusual API calls or Object deletions**)
5. Create an alarm and attach **SNS topic**

Step 2: Configure SNS for Notifications

1. Go to **AWS SNS Console**
2. Click **Create Topic** → Select **Standard**
3. Name the topic (e.g., **CloudWatch-Alerts**)
4. Click **Create Subscription**
5. Choose **Email/SMS/HTTP endpoint** as a protocol
6. Confirm the subscription via email/SMS

Step 3: Test & Validate Alerts

1. Trigger an alert (e.g., **Increase CPU usage in EC2** using a stress test)
2. Check **CloudWatch Alarms** dashboard
3. Verify if SNS sends a notification to the configured channel

---

Conclusion

By integrating **Amazon CloudWatch with SNS**, you can proactively monitor AWS services, receive alerts for performance issues, and take corrective actions quickly.

Project 2: Log Analysis with ELK Stack – Deploy Elasticsearch, Logstash, and Kibana on AWS.

The **ELK Stack** (Elasticsearch, Logstash, and Kibana) is a powerful toolset for log management and analysis. This project demonstrates how to deploy the ELK stack on AWS to collect, process, and visualize logs in real time.

Steps to Deploy ELK Stack on AWS

Step 1: Set Up an AWS EC2 Instance

1. Go to the **AWS Management Console** → Navigate to **EC2**.
2. Launch a new EC2 instance (Ubuntu 22.04 recommended).
3. Choose an instance type (t2.medium or higher for better performance).
4. Configure security groups:
    - Allow **SSH (port 22)** for remote access.
    - Allow **Elasticsearch (port 9200)**, **Logstash (port 5044)**, and **Kibana (port 5601)**.
5. Attach a key pair for SSH access.

Step 2: Install and Configure Elasticsearch

SSH into the EC2 instance:
ssh -i your-key.pem ubuntu@your-ec2-public-ip

Update the package list:
sudo apt update && sudo apt upgrade -y

Install Java (required for Elasticsearch and Logstash):
sudo apt install openjdk-11-jdk -y

Download and install Elasticsearch:
wget
https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-8.7.0-amd64.deb

sudo dpkg -i elasticsearch-8.7.0-amd64.deb

Configure Elasticsearch (/etc/elasticsearch/elasticsearch.yml):
yaml

network.host: 0.0.0.0

discovery.type: single-node

Start and enable Elasticsearch:
sudo systemctl enable elasticsearch

sudo systemctl start elasticsearch

Verify Elasticsearch is running:
curl -X GET "http://localhost:9200"

Step 3: Install and Configure Logstash

Download and install Logstash:
wget https://artifacts.elastic.co/downloads/logstash/logstash-8.7.0-amd64.deb

sudo dpkg -i logstash-8.7.0-amd64.deb

Create a Logstash configuration file (/etc/logstash/conf.d/logstash.conf):
yaml

```
input {
  beats {
    port => 5044
  }
}
filter {
  grok {
    match => { "message" => "%{COMMONAPACHELOG}" }
  }
}
output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "logs-%{+YYYY.MM.dd}"
  }
```

}

Start and enable Logstash:
sudo systemctl enable logstash

sudo systemctl start logstash

Step 4: Install and Configure Kibana

Download and install Kibana:
wget https://artifacts.elastic.co/downloads/kibana/kibana-8.7.0-amd64.deb

sudo dpkg -i kibana-8.7.0-amd64.deb

Configure Kibana (/etc/kibana/kibana.yml):
yaml

server.host: "0.0.0.0"

elasticsearch.hosts: ["http://localhost:9200"]

Start and enable Kibana:
sudo systemctl enable kibana

sudo systemctl start kibana

Access Kibana via browser:
http://your-ec2-public-ip:5601

Step 5: Install Filebeat to Forward Logs

Download and install Filebeat:
wget https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-8.7.0-amd64.deb

sudo dpkg -i filebeat-8.7.0-amd64.deb

Configure Filebeat (/etc/filebeat/filebeat.yml):
yaml

output.logstash:

hosts: ["localhost:5044"]

Enable and start Filebeat:
sudo systemctl enable filebeat

sudo systemctl start filebeat

1. Verify Filebeat logs in Kibana.

Final Validation

- Check **Elasticsearch**: curl -X GET "http://localhost:9200/_cat/indices?v"
- Check **Logstash** logs: sudo journalctl -u logstash --no-pager
- Verify **Kibana UI** at http://your-ec2-public-ip:5601
- View real-time logs in **Kibana Discover** section.

Conclusion

This setup enables real-time log collection, processing, and visualization using AWS EC2 and the ELK stack. You can extend this project by integrating AWS services like CloudWatch Logs, Lambda, and S3 for enhanced log storage and monitoring.

Project 3: Serverless URL Monitor – Use Lambda & CloudWatch to check website uptime.

Introduction

A serverless URL monitor helps check website uptime using AWS Lambda and CloudWatch. AWS Lambda functions execute on a scheduled interval to send HTTP requests to a specified URL and log the response. CloudWatch Alarms notify users if the website is down. This eliminates the need for a dedicated server, reducing costs and management overhead.

Steps to Implement

1. Set Up an IAM Role for Lambda

- Go to the **AWS IAM Console → Roles → Create Role**.
- Select **AWS Service → Lambda → Next**.
- Attach policies:
  - AWSLambdaBasicExecutionRole (for logging).
  - CloudWatchFullAccess (for monitoring).
- Name the role **LambdaMonitoringRole** and create it.

2. Create a Lambda Function

- Navigate to **AWS Lambda → Create function**.
- Select **Author from scratch**.
- Name it URLMonitorLambda.
- Choose **Runtime**: Python 3.9 (or Node.js).
- Assign the IAM role **LambdaMonitoringRole**.
- Click **Create function**.

3. Add Code to Lambda Function

- In the Lambda function editor, replace the default code with the following (Python example):

python

```
import json
import requests
import boto3
import os

def lambda_handler(event, context):
    url = os.getenv('MONITOR_URL', 'https://example.com')  # Set URL in environment variables
    try:
        response = requests.get(url, timeout=5)
        status = response.status_code
        if status == 200:
            return {"statusCode": 200, "body": "Website is Up"}
        else:
            send_alert(f"Website {url} is DOWN. Status Code: {status}")
            return {"statusCode": status, "body": "Website is Down"}
```

```
    except requests.RequestException:
        send_alert(f"Website {url} is NOT REACHABLE")
        return {"statusCode": 500, "body": "Website is Not Reachable"}

def send_alert(message):
    sns_client = boto3.client("sns")
    sns_client.publish(
        TopicArn=os.getenv('SNS_TOPIC_ARN'),
        Message=message,
        Subject="Website Downtime Alert"
    )
```

- **Set Environment Variables:**
  - MONITOR_URL → Website URL to check.
  - SNS_TOPIC_ARN → ARN of an SNS topic for notifications.

4. Create a CloudWatch Event Rule

- Go to **AWS CloudWatch → Rules → Create Rule**.
- Select **Event Source**: **Schedule Expression** (rate(5 minutes)).
- Select **Target** → Choose **Lambda Function** → Select URLMonitorLambda.
- Click **Create Rule**.

5. Set Up SNS Notifications (Optional)

- Navigate to **Amazon SNS → Create Topic**.
- Choose **Standard Topic** → Name it WebsiteAlerts.
- Copy the **Topic ARN** and add it as an environment variable in the Lambda function.
- Subscribe to the topic with an **email** or **SMS** for alerts.

6. Deploy & Test the Lambda Function

- Click **Deploy** in AWS Lambda.
- Trigger manually or wait for CloudWatch to invoke it.
- Check **CloudWatch Logs** for execution results.

Outcome

The Lambda function will periodically check the website's uptime and send an alert via SNS if the site is down. CloudWatch Logs help track availability metrics.

Enhancements

- Integrate with **AWS DynamoDB** to log website uptime history.
- Add a **Retry Mechanism** before sending an alert.
- Use **AWS Step Functions** for advanced monitoring workflows.

Project 4: Query S3 Logs with AWS Athena – Analyze stored logs without setting up a database.

Introduction

A serverless URL monitor helps check website uptime using AWS Lambda and CloudWatch. AWS Lambda functions execute on a scheduled interval to send HTTP requests to a specified URL and log the response. CloudWatch Alarms notify users if the website is down. This eliminates the need for a dedicated server, reducing costs and management overhead.

Steps to Implement

1. Set Up an IAM Role for Lambda

- Go to the **AWS IAM Console → Roles → Create Role**.
- Select **AWS Service → Lambda → Next**.
- Attach policies:
  - AWSLambdaBasicExecutionRole (for logging).
  - CloudWatchFullAccess (for monitoring).
- Name the role **LambdaMonitoringRole** and create it.

2. Create a Lambda Function

- Navigate to **AWS Lambda → Create function**.
- Select **Author from scratch**.
- Name it URLMonitorLambda.
- Choose **Runtime**: Python 3.9 (or Node.js).
- Assign the IAM role **LambdaMonitoringRole**.
- Click **Create function**.

3. Add Code to Lambda Function

- In the Lambda function editor, replace the default code with the following (Python example):

python

```
import json
import requests
import boto3
import os

def lambda_handler(event, context):
    url = os.getenv('MONITOR_URL', 'https://example.com')  # Set URL in environment variables
    try:
        response = requests.get(url, timeout=5)
        status = response.status_code
        if status == 200:
            return {"statusCode": 200, "body": "Website is Up"}
        else:
            send_alert(f"Website {url} is DOWN. Status Code: {status}")
            return {"statusCode": status, "body": "Website is Down"}
    except requests.RequestException:
        send_alert(f"Website {url} is NOT REACHABLE")
        return {"statusCode": 500, "body": "Website is Not Reachable"}

def send_alert(message):
    sns_client = boto3.client("sns")
    sns_client.publish(
        TopicArn=os.getenv('SNS_TOPIC_ARN'),
        Message=message,
        Subject="Website Downtime Alert"
    )
```

- **Set Environment Variables:**
  - MONITOR_URL → Website URL to check.
  - SNS_TOPIC_ARN → ARN of an SNS topic for notifications.

4. Create a CloudWatch Event Rule

- Go to **AWS CloudWatch → Rules → Create Rule**.
- Select **Event Source**: **Schedule Expression** (rate(5 minutes)).
- Select **Target** → Choose **Lambda Function** → Select URLMonitorLambda.
- Click **Create Rule**.

5. Set Up SNS Notifications (Optional)

- Navigate to **Amazon SNS → Create Topic**.
- Choose **Standard Topic** → Name it WebsiteAlerts.
- Copy the **Topic ARN** and add it as an environment variable in the Lambda function.
- Subscribe to the topic with an **email** or **SMS** for alerts.

6. Deploy & Test the Lambda Function

- Click **Deploy** in AWS Lambda.
- Trigger manually or wait for CloudWatch to invoke it.
- Check **CloudWatch Logs** for execution results.

Outcome

The Lambda function will periodically check the website's uptime and send an alert via SNS if the site is down. CloudWatch Logs help track availability metrics.

Enhancements

- Integrate with **AWS DynamoDB** to log website uptime history.
- Add a **Retry Mechanism** before sending an alert.
- Use **AWS Step Functions** for advanced monitoring workflows.

Project 5: Application Performance Monitoring – Use AWS X-Ray to trace application requests and optimize performance.

Introduction

AWS X-Ray is a distributed tracing service that helps developers analyze and debug applications running in production or development. It provides insights into application requests, latency, and dependencies across microservices, improving performance and troubleshooting issues efficiently. X-Ray is particularly useful for AWS Lambda, API Gateway, EC2, ECS, and containerized applications.

Steps to Implement AWS X-Ray for Performance Monitoring

1. Prerequisites

- AWS account
- An application running on EC2, ECS, Lambda, or API Gateway
- AWS CLI and IAM permissions

2. Enable AWS X-Ray

For EC2 and ECS:

Install the X-Ray Daemon:
curl -o /tmp/aws-xray-daemon.rpm https://s3.amazonaws.com/aws-xray-assets.us-east-1/xray-daemon/aws-xray-daemon-3.x86_64.rpm

sudo yum install -y /tmp/aws-xray-daemon.rpm

Start the X-Ray daemon:
sudo systemctl start xray

For AWS Lambda:

- Enable Active Tracing in Lambda function settings.

Add the X-Ray SDK to your Lambda function:
pip install aws-xray-sdk

3. Integrate X-Ray with Your Application

For Python Application:

Install the X-Ray SDK:
pip install aws-xray-sdk

Modify your application code:
python

```
from aws_xray_sdk.core import xray_recorder

from aws_xray_sdk.ext.flask.middleware import XRayMiddleware

from flask import Flask


app = Flask(__name__)

xray_recorder.configure(service='MyApp')

XRayMiddleware(app, xray_recorder)


@app.route('/')

def index():

    return "Hello, X-Ray!"


if __name__ == '__main__':

    app.run()
```

-

For Node.js Application:

Install the SDK:
npm install aws-xray-sdk

Modify your application code:
javascript

```
const AWSXRay = require('aws-xray-sdk');

const express = require('express');


const app = express();

AWSXRay.captureHTTPsGlobal(require('http'));


app.use(AWSXRay.express.openSegment('MyApp'));


app.get('/', (req, res) => {

    res.send('Hello, X-Ray!');

});


app.use(AWSXRay.express.closeSegment());
```

```
app.listen(3000, () => console.log('Server running on port 3000'));
```

4. Configure IAM Permissions

Create an IAM role with the following permissions:
json

```
{
  "Effect": "Allow",
  "Action": [
    "xray:PutTraceSegments",
    "xray:PutTelemetryRecords",
    "xray:GetSamplingRules"
  ],
  "Resource": "*"
}
```

5. View Traces in AWS X-Ray

- Open **AWS Console → X-Ray**
- Select your application and analyze request traces
- Identify latency issues, bottlenecks, and dependencies

Conclusion

AWS X-Ray provides deep visibility into application performance, helping developers trace requests, analyze bottlenecks, and optimize their architecture. By integrating X-Ray with EC2, Lambda, or API Gateway, you can monitor distributed applications efficiently.

---

9. Networking & Connectivity

Project 1: Route 53 DNS Management – Configure and manage DNS records for custom domains.

Introduction

Amazon Route 53 is a scalable and highly available Domain Name System (DNS) web service. It helps route end users to applications by translating domain names (e.g., example.com) into IP addresses. In this project, you will configure and manage DNS records for a custom domain using Route 53.

Steps to Complete the Project

1. Register a Domain (Optional)

- If you don't have a domain, register one using **Route 53** or any domain registrar like **GoDaddy, Namecheap, etc.**
- If using Route 53:
    1. Navigate to **AWS Route 53 Console**.
    2. Click **"Registered domains"** > **"Register Domain"**.
    3. Choose a domain, provide details, and complete the purchase.

2. Create a Hosted Zone

- A Hosted Zone manages DNS settings for your domain.
- Steps:
    1. Open **Route 53** in the AWS Console.
    2. Click **"Hosted Zones"** > **"Create Hosted Zone"**.

3. Enter your domain name and choose **Public Hosted Zone**.
4. Click **"Create"**.
5. Note the **Name Server (NS) records** displayed.

3. Update Name Servers (If Not Using Route 53 Domain)

- If your domain is registered with another provider, update the **NS records** in the provider's dashboard.
- Copy the NS records from Route 53 and update them in your domain registrar's settings.

4. Configure DNS Records

- Click on your Hosted Zone and **add records** based on your needs:
  - **A Record (IPv4 Address)** – Maps domain to an IP.
  - **CNAME Record** – Maps domain to another domain (e.g., www.example.com → example.com).
  - **MX Record** – Routes email for your domain.
  - **TXT Record** – Used for domain verification (e.g., AWS SES, Google Workspace).

Steps to add a record:

1. Click **"Create Record"**.
2. Choose the **Record Type** (A, CNAME, MX, etc.).
3. Enter the domain/subdomain (e.g., www.example.com).
4. Provide the value (IP, another domain, or text value).
5. Click **"Create Records"**.

5. Configure Health Checks (Optional)

- Health checks monitor endpoint availability.
- Steps:
  1. Open **Route 53 Console** > Click **Health Checks**.
  2. Click **Create Health Check**.
  3. Provide the IP or domain of the endpoint to monitor.
  4. Set failure thresholds and notifications.
  5. Save the health check.

6. Enable Route 53 Traffic Routing (Optional)

- Route 53 allows **Geolocation, Latency, Weighted, and Failover Routing**.
- Example: **Latency-based routing** directs users to the closest AWS region.

7. Test DNS Configuration

- Use tools like:
  - **nslookup** (nslookup example.com)
  - **dig** (dig example.com)
  - **Route 53 Test Record Set**

8. Use Route 53 with AWS Services (Optional)

- **Connect Route 53 with an S3 Static Website**
- **Set up Route 53 with an Elastic Load Balancer (ELB)**

Conclusion

With Route 53, you can efficiently manage DNS for your custom domain, ensuring high availability, traffic control, and integration with AWS services.

Project 2: AWS Transit Gateway for Multi-VPC Communication – Connect multiple AWS VPCs securely.

Introduction

AWS Transit Gateway is a networking service that enables secure and scalable communication between multiple Amazon Virtual Private Clouds (VPCs) and on-premises networks. It acts as a central hub, simplifying network management by reducing the number of direct VPC-to-VPC peering connections. This is ideal for large-scale deployments where multiple VPCs need to communicate securely without complex peering configurations.

Steps to Set Up AWS Transit Gateway for Multi-VPC Communication

Step 1: Create a Transit Gateway

1. Sign in to the AWS Management Console.
2. Navigate to **VPC → Transit Gateways → Create Transit Gateway**.
3. Provide a **Name** and **Description**.
4. Set the **Amazon ASN (Autonomous System Number)** or use the default.
5. Enable **DNS support**, **VPN ECMP support**, and **Default route table association** if required.
6. Click **Create Transit Gateway** and note the **Transit Gateway ID**.

Step 2: Attach VPCs to the Transit Gateway

1. Go to **VPC → Transit Gateway Attachments → Create Transit Gateway Attachment**.
2. Select **Attachment Type** as **VPC**.
3. Choose the **Transit Gateway** created earlier.
4. Select the **VPC** you want to attach and choose the **Subnets** for the attachment.
5. Click **Create Attachment**.
6. Repeat the process for other VPCs you want to connect.

Step 3: Configure Route Tables for Communication

1. Navigate to **Transit Gateway Route Tables → Create Route Table**.
2. Provide a **Name** and **Description**.
3. Associate the **VPC Attachments** to this route table.
4. Add **routes** for traffic between VPCs.
5. Click **Create Route Table**.

Step 4: Update VPC Route Tables

1. Go to **VPC → Route Tables**.
2. Select the **route table** associated with each VPC.
3. Click **Edit Routes → Add Route**.
4. Set the **Destination CIDR** (other VPC's CIDR block).
5. Choose the **Transit Gateway ID** as the Target.
6. Click **Save Routes**.
7. Repeat for all VPCs to enable bidirectional communication.

Step 5: Test Connectivity Between VPCs

1. Launch **EC2 instances** in different VPCs.
2. Assign security group rules to allow inbound ICMP (ping) or necessary ports.
3. SSH into one instance and try pinging another instance in a different VPC.
4. If successful, the **Transit Gateway** is working correctly.

Conclusion

AWS Transit Gateway simplifies multi-VPC communication by acting as a centralized hub, eliminating the need for multiple peering connections. It enhances security, reduces management overhead, and provides scalability for growing cloud architectures.

Project 3: Kubernetes on AWS (EKS) – Deploy a containerized workload using Amazon EKS.

Introduction

Amazon Elastic Kubernetes Service (EKS) is a managed Kubernetes service that simplifies running Kubernetes on AWS. It automates cluster provisioning, scaling,

and maintenance. In this project, you will deploy a containerized workload on EKS, ensuring high availability and scalability.

Steps to Deploy a Containerized Workload on EKS

Step 1: Prerequisites

Ensure you have the following:

- AWS CLI installed and configured
- kubectl installed
- eksctl installed
- Docker installed

Step 2: Create an EKS Cluster

eksctl create cluster --name my-eks-cluster --region us-east-1 --nodegroup-name my-nodes --node-type t3.medium --nodes 2

This command creates an EKS cluster with a managed node group.

Step 3: Configure kubectl to Connect to EKS

aws eks update-kubeconfig --region us-east-1 --name my-eks-cluster

Verify cluster access:

kubectl get nodes

Step 4: Build and Push Docker Image

1. Create a Dockerfile for your application.

Build and tag the image:
docker build -t my-app .

Push the image to Amazon ECR (Elastic Container Registry):
aws ecr create-repository --repository-name my-app

aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com

docker tag my-app <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/my-app

docker push <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/my-app

Step 5: Deploy Application to EKS

Create a Kubernetes deployment YAML (deployment.yaml):

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
    app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/my-app
          ports:
            - containerPort: 80
```

Apply the deployment:

sh

```
kubectl apply -f deployment.yaml
```

Step 6: Expose the Application

Create a service (service.yaml):

yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Apply the service:

sh

```
kubectl apply -f service.yaml
```

Step 7: Verify Deployment

Check running pods:

```
kubectl get pods
```

Get the LoadBalancer URL:

kubectl get svc my-app-service

Access your application in a browser using the external URL.

Step 8: Clean Up Resources

Delete the cluster:

eksctl delete cluster --name my-eks-cluster

Conclusion

You have successfully deployed a containerized application on Amazon EKS. This setup allows you to scale and manage workloads efficiently using Kubernetes.

Project 4: Hybrid Cloud Setup with AWS VPN – Connect an on-premises data center to AWS.

Connect an On-Premises Data Center to AWS

A **Hybrid Cloud** setup allows organizations to extend their on-premises infrastructure to AWS, enabling seamless communication between local and cloud environments. AWS Site-to-Site VPN creates a secure, encrypted connection between your on-premises network and AWS VPC, providing low-latency, high-availability access to cloud resources.

Steps to Set Up Hybrid Cloud with AWS VPN

Step 1: Create a VPC in AWS

1. Go to the **AWS Management Console → VPC Dashboard**.
2. Click **Create VPC** and provide the following details:
    - Name: Hybrid-VPC
    - IPv4 CIDR Block: 10.0.0.0/16
3. Click **Create**.

Step 2: Create Subnets and Route Tables

1. **Create Public and Private Subnets** in the VPC.
2. Assign **route tables** to each subnet.

Step 3: Create a Virtual Private Gateway (VGW)

1. Navigate to **VPC Dashboard → Virtual Private Gateways**.
2. Click **Create Virtual Private Gateway**.
    - Name: Hybrid-VGW
    - ASN: Use default or specify one.
3. Click **Attach to VPC** and select your **Hybrid-VPC**.

Step 4: Create a Customer Gateway (CGW)

1. Go to **VPC Dashboard → Customer Gateways**.
2. Click **Create Customer Gateway**.
    - Name: OnPrem-CGW
    - IP Address: Public IP of your **on-premises router/firewall**.
    - Routing Type: **Static or Dynamic (BGP)**.

Step 5: Create a Site-to-Site VPN Connection

1. Navigate to **VPC Dashboard → VPN Connections**.
2. Click **Create VPN Connection**.
    - Name: Hybrid-VPN
    - Virtual Private Gateway: Select Hybrid-VGW.
    - Customer Gateway: Select OnPrem-CGW.
    - Routing Options: **Dynamic (BGP) or Static**.

3. Click **Create VPN Connection**.

Step 6: Configure VPN on On-Premises Router

1. **Download the VPN Configuration** file from AWS.
2. **Apply the configuration** to your on-premises router/firewall (Cisco, Juniper, etc.).
3. Establish **IKE Phase 1 & 2** for encryption and key exchange.

Step 7: Verify VPN Connectivity

1. Go to **VPC Dashboard → VPN Connections →** Check **Tunnel Status**.
2. On your on-premises router, verify **IPsec session status**.

Test connectivity using:
ping 10.0.1.10 # AWS Private IP from On-Prem

traceroute 10.0.1.10

Step 8: Configure Route Tables in AWS

1. Go to **VPC Dashboard → Route Tables**.
2. Edit the **Hybrid-VPC** route table:
    - **Destination:** Your On-Prem Network CIDR
    - **Target:** Virtual Private Gateway (VGW)
3. Click **Save Routes**.

Step 9: Secure the Traffic

1. Update **Security Groups** and **NACLs** to allow traffic from on-premises IP ranges.
2. Use **AWS Network Firewall** or **VPN Monitoring** for security.

Step 10: Test End-to-End Communication

- Ping an **AWS instance** from the **on-premises server**.
- Access **AWS services** from on-prem via private IP.
- Use **AWS CloudWatch** for VPN tunnel monitoring.

Conclusion

This setup ensures secure and **seamless communication** between your on-premises data center and AWS. You can now extend workloads, enable disaster recovery, or securely access cloud resources from your private network.

---

10. Data Processing & Analytics

Project 1: Big Data Processing with AWS EMR – Use Apache Spark to analyze large datasets.

Introduction

Amazon EMR (Elastic MapReduce) is a managed big data framework that allows you to process large datasets efficiently using Apache Spark, Hadoop, and other frameworks. EMR simplifies data processing, making it scalable and cost-effective for various analytical workloads.

In this project, you will use **Apache Spark** on AWS EMR to analyze large datasets stored in **Amazon S3**.

Steps to Complete the Project

1. Set Up Prerequisites

- Ensure you have an **AWS account**.

Install and configure the **AWS CLI**:
aws configure

Create an **S3 bucket** to store datasets.
aws s3 mb s3://your-emr-bucket

---

2. Launch an EMR Cluster

You can launch an EMR cluster using the AWS Console or AWS CLI.

Using AWS CLI:
aws emr create-cluster \

--name "Spark-Cluster" \

--release-label emr-6.7.0 \

--applications Name=Spark \

--instance-type m5.xlarge \

--instance-count 3 \

--use-default-roles \

--ec2-attributes KeyName=your-key

- **Using AWS Console**:
  - Navigate to **EMR Service**.
  - Click **Create Cluster** → Choose **Apache Spark**.
  - Select instance types and **Start the Cluster**.

---

3. Upload Data to S3

Prepare your dataset and upload it to the S3 bucket.

aws s3 cp dataset.csv s3://your-emr-bucket/

---

4. Run a Spark Job on EMR

Once the cluster is running, SSH into the master node:

ssh -i your-key.pem hadoop@master-node-dns

Start the **Spark shell**:

spark-shell

Run a simple Spark job to analyze data:

scala

val data = spark.read.option("header",
"true").csv("s3://your-emr-bucket/dataset.csv")

data.groupBy("column_name").count().show()

---

5. Stop or Terminate the EMR Cluster

After processing, you can stop or terminate the cluster to save costs:

aws emr terminate-clusters --cluster-ids j-XXXXXXXXXXXXX

Conclusion

This project demonstrates how to process large datasets using **Apache Spark on AWS EMR**. By leveraging EMR's managed cluster, you can efficiently run big data analytics without managing infrastructure.

Project 2: Real-Time Data Streaming with Kinesis – Process IoT or social media data in real time.

Introduction

AWS Kinesis is a cloud-based service that enables real-time data processing from IoT devices, social media, application logs, and other data sources. It allows organizations to ingest, analyze, and process streaming data efficiently. Kinesis provides different services like **Kinesis Data Streams, Kinesis Data Firehose, and Kinesis Data Analytics** for real-time data handling.

In this project, we will use **Kinesis Data Streams** to collect real-time data, **Lambda** to process it, and **DynamoDB** to store the processed data.

Steps to Implement the Project

1. Set Up AWS Kinesis Data Stream

- Sign in to AWS Console.
- Navigate to **Kinesis → Create a Data Stream**.
- Provide a name (e.g., IoT-Data-Stream).
- Select the number of shards based on expected data throughput.
- Click **Create Stream**.

2. Create an AWS Lambda Function to Process Data

- Go to **AWS Lambda → Create function**.
- Choose **Author from scratch** and enter a function name (ProcessKinesisData).
- Select **Runtime** as Python 3.x.

In the function code, process Kinesis records:
python

```
import json

import boto3


dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('IoTData')


def lambda_handler(event, context):

  for record in event['Records']:

    payload = json.loads(record['kinesis']['data'])

    table.put_item(Item={

      'device_id': payload['device_id'],

      'timestamp': payload['timestamp'],

      'temperature': payload['temperature']

    })

  return {'statusCode': 200, 'body': 'Data Processed'}
```

- Deploy the function.
- Add a **Kinesis trigger**, selecting the stream IoT-Data-Stream.

3. Create a DynamoDB Table for Storing Data

- Go to **DynamoDB → Create Table**.
- Name the table IoTData.
- Set device_id as the primary key.

- Click **Create Table**.

4. Send Real-Time Data to Kinesis

Use the AWS SDK (Python Boto3) to send data:

python

```
import boto3
import json
import time
import random

kinesis_client = boto3.client('kinesis')

stream_name = 'IoT-Data-Stream'

while True:
    data = {
        'device_id': 'sensor-1',
        'timestamp': int(time.time()),
        'temperature': random.uniform(20.0, 30.0)
    }

    response = kinesis_client.put_record(
        StreamName=stream_name,
        Data=json.dumps(data),
        PartitionKey='sensor-1'
    )

    print(f"Sent data: {data}")
    time.sleep(5)
```

5. Verify Data Processing

- Check **CloudWatch Logs** for Lambda execution.
- Open DynamoDB table and verify that processed data is stored.

Conclusion

This project demonstrates how to set up a real-time data streaming pipeline with AWS Kinesis, Lambda, and DynamoDB. It allows continuous data ingestion, processing, and storage, making it suitable for IoT telemetry and social media data analysis.

Project 3: AWS Quicksight for BI Dashboard – Visualize business data using Amazon QuickSight.

Introduction

Amazon QuickSight is a cloud-based Business Intelligence (BI) tool that enables organizations to visualize data and gain insights in real-time. In this project, we

will build a **Sales Performance Dashboard** using QuickSight to track key sales metrics like revenue, orders, customer trends, and product performance.

Steps to Set Up an AWS QuickSight Sales Dashboard

1. Set Up the Sales Data Source

- Prepare sales data in **Amazon S3**, **RDS (MySQL/PostgreSQL)**, **Amazon Redshift**, or **Athena**.
- Ensure your dataset includes key fields such as:
  - **Order ID**, **Customer Name**, **Product Category**, **Sales Amount**, **Profit Margin**, **Date of Sale**, **Region**, **Sales Representative**.

2. Enable Amazon QuickSight

- Navigate to the AWS **Management Console**.
- Search for **QuickSight** and **sign up**.
- Choose **Enterprise Edition** if you need **Active Directory integration** and **Row-Level Security**.
- Assign **IAM permissions** for QuickSight to access data sources.

3. Connect Data Source to QuickSight

- Go to **Manage Data** > **New Data Set**.
- Select the data source (Amazon S3, RDS, Redshift, Athena).
- Provide necessary **IAM permissions** or database credentials.
- **Load the sales data** into QuickSight.

4. Data Preparation in QuickSight

- Use **QuickSight's Data Prep tool** to clean and transform data.
- Create **calculated fields**, such as:
  - **Total Revenue** = SUM(Sales Amount)
  - **Profit Percentage** = (Profit Margin / Sales Amount) * 100
  - **Monthly Sales Growth** = (Current Month Sales - Previous Month Sales) / Previous Month Sales * 100
- Set up **filters** and **aggregations** (e.g., grouping by product category or region).

5. Build Sales Visualizations

- Go to **Create Analysis** and select your dataset.
- Drag & drop the following key **charts & visualizations**:
  - **Total Sales & Profit** → KPI Widget
  - **Monthly Sales Trend** → Line Chart
  - **Top Selling Products** → Bar Chart
  - **Sales by Region** → Geo Map
  - **Sales Rep Performance** → Table with conditional formatting
  - **New vs. Returning Customers** → Pie Chart
- Configure **dynamic filters** to allow users to filter by **date range, product category, or region**.

6. Publish and Share the Dashboard

- Save the dashboard and **publish it**.
- Set up **Scheduled Refresh** to keep data up to date.
- Share it with sales teams via **IAM role-based access**.
- Embed the dashboard into a web portal if needed.

7. Monitor and Optimize

- Use **QuickSight ML Insights** for:
  - **Anomaly detection** (e.g., sudden drop in sales).
  - **Forecasting** (e.g., next quarter sales projection).
- Optimize data queries for better performance.

Conclusion

With AWS QuickSight, sales teams can track real-time revenue, identify top-performing products, and analyze customer trends to make **data-driven decisions**.

Project 4: AWS Athena for Log Analysis – Query CloudTrail and VPC Flow Logs using Athena.

Introduction

AWS Athena is a serverless query service that allows you to analyze data in Amazon S3 using standard SQL. It is commonly used for log analysis, including CloudTrail logs (which track API activity in AWS) and VPC Flow Logs (which capture network traffic metadata). By leveraging Athena, you can quickly search, filter, and gain insights from your AWS logs without needing a traditional database.

Steps to Implement AWS Athena for Log Analysis

1. Enable CloudTrail and VPC Flow Logs

- **CloudTrail Logs:**
  1. Go to the **AWS Management Console → CloudTrail**
  2. Click **Create trail** and configure:
     - Enable **Management events**
     - Store logs in an **S3 bucket**
  3. Click **Create**
- **VPC Flow Logs:**
  1. Navigate to **EC2 Dashboard → Network Interfaces**
  2. Select a VPC, click **Flow Logs**, then **Create Flow Log**
  3. Choose **Amazon S3** as the destination
  4. Click **Create**

---

2. Create an S3 Bucket for Log Storage

1. Go to **Amazon S3**
2. Click **Create bucket**
3. Set a unique name (e.g., my-log-bucket)
4. Ensure the bucket has public access blocked
5. Click **Create bucket**

---

3. Configure Athena to Query Logs

- **Set Up Athena:**
  1. Open **AWS Athena**
  2. Click **Settings →** Set query results location to the S3 bucket (e.g., s3://my-log-bucket/athena-results/)
- **Create a Database in Athena:**

Run the following SQL in Athena Query Editor:
sql

```
CREATE DATABASE log_analysis;
```

1. Select **log_analysis** as the active database

---

4. Create Tables for Log Analysis

CloudTrail Table:
sql

```
CREATE EXTERNAL TABLE cloudtrail_logs (
  eventVersion STRING,
  eventTime STRING,
  eventSource STRING,
  eventName STRING,
  awsRegion STRING,
```

```
    sourceIPAddress STRING,
    userAgent STRING
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION 's3://my-log-bucket/AWSLogs/{account-id}/CloudTrail/'
```

- *(Replace {account-id} with your AWS Account ID.)*

VPC Flow Logs Table:
sql

```
CREATE EXTERNAL TABLE vpc_flow_logs (
    version INT,
    account_id STRING,
    interface_id STRING,
    srcaddr STRING,
    dstaddr STRING,
    srcport INT,
    dstport INT,
    protocol INT,
    packets INT,
    bytes INT,
    start_time INT,
    end_time INT,
    action STRING,
    log_status STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ' '
LOCATION 's3://my-log-bucket/vpc-flow-logs/';
```

---

5. Run Queries to Analyze Logs

Find Unauthorized Access Attempts in CloudTrail:
sql

```
SELECT eventTime, eventSource, eventName, sourceIPAddress
FROM cloudtrail_logs
WHERE eventName = 'ConsoleLogin' AND userAgent LIKE '%Failed%';
```

Monitor Traffic to a Specific IP in VPC Flow Logs:
sql

```
SELECT srcaddr, dstaddr, bytes, action
FROM vpc_flow_logs
WHERE dstaddr = '192.168.1.10';
```

---

6. Automate Log Queries with AWS Lambda (Optional)

- Use AWS Lambda with Athena to schedule periodic log queries and send alerts to Amazon SNS.

---

7. Visualize Data in Amazon QuickSight (Optional)

- Connect Athena to QuickSight for dashboards and reports.

Conclusion

This project helps monitor AWS security and network activity using Athena's SQL queries on CloudTrail and VPC Flow Logs. Since Athena is serverless, you only pay for the queries you run, making it an efficient solution for log analysis.

---

11. AI/ML & IoT

Project 1: Train & Deploy ML Models with SageMaker – Use AWS SageMaker for predictive analytics.

Introduction

AWS SageMaker is a fully managed machine learning (ML) service that simplifies the process of building, training, and deploying ML models at scale. It supports various ML frameworks such as TensorFlow, PyTorch, and Scikit-Learn, along with built-in algorithms for common ML tasks. In this project, we will use AWS SageMaker to train an ML model for predictive analytics and deploy it as an endpoint for real-time predictions.

Complete Steps

1. Set Up AWS SageMaker Environment

1. **Sign in to AWS Console** and navigate to **SageMaker**.
2. Go to **Notebook instances → Create notebook instance**.
3. Set a name (e.g., sagemaker-ml-project).
4. Choose an instance type (ml.t2.medium for basic workloads).
5. Under **IAM role**, create a new role with AmazonSageMakerFullAccess.
6. Click **Create notebook instance** and wait for it to start.
7. Once the status is "InService," open **Jupyter Notebook**.

---

2. Prepare the Dataset

1. **Upload the dataset** to an Amazon S3 bucket.

Open a new Jupyter Notebook and **load the dataset**:
python

```
import pandas as pd


s3_bucket = "your-bucket-name"

dataset_path = f"s3://{s3_bucket}/your-dataset.csv"


df = pd.read_csv(dataset_path)

print(df.head())  # Display the first few rows
```

Preprocess the data (handle missing values, encode categorical variables, normalize numerical values):
python

```
df = df.dropna()  # Remove missing values

df.to_csv("processed_data.csv", index=False)
```

Upload the processed data back to S3:
python

```
import sagemaker

from sagemaker.s3 import S3Uploader


sagemaker_session = sagemaker.Session()

s3_data_path = S3Uploader.upload("processed_data.csv",
f"s3://{s3_bucket}/processed/")

print(f"Data uploaded to {s3_data_path}")
```

---

3. Train an ML Model Using SageMaker

SageMaker provides built-in ML algorithms. We will use **Linear Learner** for binary classification.

Set up SageMaker training session:
python

```
from sagemaker import get_execution_role

from sagemaker.amazon.linear_learner import LinearLearner

role = get_execution_role()

train_data = s3_data_path  # Use processed dataset
```

Create a training job:
python

```
linear = LinearLearner(role=role,
                       instance_count=1,
                       instance_type="ml.m4.xlarge",
                       predictor_type="binary_classifier",
                       output_path=f"s3://{s3_bucket}/output/")
```

Train the model:
python

```
linear.fit({"train": train_data})
```

---

4. Deploy the Model as an Endpoint

Once the model is trained, deploy it as a real-time API.

Deploy the trained model:
python

```
predictor = linear.deploy(instance_type="ml.m4.xlarge", initial_instance_count=1)
```

Test the model with sample data:
python

```
import numpy as np


test_sample = np.array([[0.5, 0.2, 0.8]])  # Example feature vector

prediction = predictor.predict(test_sample)

print("Prediction:", prediction)
```

---

5. Monitor and Optimize

1. **Check logs** in SageMaker to track model performance.
2. Use **AWS CloudWatch** to monitor inference requests.
3. Optimize with **AutoML** for better accuracy.

---

6. Clean Up Resources

Delete the endpoint:
python

```
predictor.delete_endpoint()
```

1. **Delete SageMaker notebook instance** in the AWS console.

Conclusion

This project demonstrated how to use **AWS SageMaker** to train and deploy an ML model for predictive analytics. By leveraging SageMaker's built-in algorithms and managed infrastructure, ML development becomes faster and more scalable.

Project 2: Fraud Detection System – Build an ML-based fraud detection model on AWS.

Introduction

Fraud detection is essential for preventing fraudulent transactions in financial, banking, and e-commerce sectors. Machine learning (ML) models can analyze transaction data and detect anomalies to identify potential fraud. AWS provides scalable tools like **S3, SageMaker, Lambda, API Gateway, and CloudWatch** to build an end-to-end fraud detection system.

Complete Steps

1. Set Up AWS Environment

- Create an **AWS Account** if not already registered.
- Configure an **IAM Role** with necessary permissions for SageMaker, S3, Lambda, and API Gateway.
- Install and configure the **AWS CLI** to interact with AWS services from the command line.

---

2. Prepare the Dataset

- Use publicly available datasets like **Kaggle's Credit Card Fraud Dataset** or your organization's dataset.
- Preprocess the dataset:

    - Handle missing values.
    - Normalize numerical features.
    - Encode categorical data if needed.
    - Split the dataset into **training (80%) and testing (20%)**.
- Upload the dataset to an **S3 bucket** for storage.

---

3. Train the ML Model Using Amazon SageMaker

3.1 Launch SageMaker Notebook Instance

- Go to AWS SageMaker → Create a new Jupyter Notebook instance.
- Attach an IAM role that allows access to S3.

3.2 Load Dataset and Train the Model

- Load the dataset from S3.
- Choose an appropriate ML algorithm:
    - **XGBoost** (gradient boosting for tabular data).
    - **Random Forest** (good for fraud pattern recognition).
    - **Neural Networks** (for deep learning-based fraud detection).
- Train the model on SageMaker's managed training service.

3.3 Evaluate the Model

- Use metrics like **Precision, Recall, F1-score, and ROC-AUC** to assess model performance.
- Fine-tune hyperparameters if needed.

---

4. Deploy the ML Model as an API

4.1 Create a SageMaker Endpoint

- Deploy the trained model to a **SageMaker Endpoint** for real-time inference.

4.2 Expose the Model via API Gateway

- Create an **AWS Lambda function** to invoke the SageMaker endpoint.
- Use **API Gateway** to expose the fraud detection API to external applications.

---

5. Automate Fraud Detection

5.1 Real-Time Fraud Detection with AWS Lambda

- Configure an **AWS Lambda function** that automatically triggers when a new transaction is processed.
- The function will send transaction data to the SageMaker endpoint and receive a fraud probability score.

5.2 Store and Monitor Transactions

- Store transaction logs in **Amazon DynamoDB** or **Amazon RDS**.
- Use **Amazon CloudWatch** to monitor API calls and Lambda function performance.
- Set up **SNS alerts** for suspicious transactions.

---

6. Visualizing Fraud Analytics with Amazon QuickSight

- Connect Amazon QuickSight to your transaction database.
- Create dashboards to visualize fraud trends and suspicious activities.

Conclusion

This AWS-based fraud detection system provides an automated solution for identifying fraudulent transactions in real-time. Using SageMaker for ML model training, Lambda for automation, and API Gateway for integration, this architecture ensures scalability and efficiency.

Project 3: AI-Powered Image Recognition – Use AWS Rekognition to analyze images.

AWS Rekognition is a cloud-based AI service that enables image and video analysis using deep learning models. It can identify objects, people, text, scenes, and activities in images, making it useful for applications like facial recognition, content moderation, and automated tagging.

Steps to Implement AWS Rekognition for Image Analysis

1. Set Up AWS Environment

- Sign in to the AWS Management Console
- Navigate to **IAM** and create a user with **AmazonRekognitionFullAccess**

Configure the AWS CLI on your local machine
aws configure

---

2. Upload Images to S3 (Optional)

- Create an **S3 bucket** to store images

Upload an image to the bucket
aws s3 cp image.jpg s3://your-bucket-name/

---

3. Use AWS Rekognition for Image Analysis

Detect Objects and Scenes

aws rekognition detect-labels \

--image "S3Object={Bucket=your-bucket-name,Name=image.jpg}" \

--region your-region

Detect Faces in an Image

aws rekognition detect-faces \

--image "S3Object={Bucket=your-bucket-name,Name=image.jpg}" \

--region your-region

Detect Text in an Image

aws rekognition detect-text \

--image "S3Object={Bucket=your-bucket-name,Name=image.jpg}" \

--region your-region

---

4. Implement with Python (Boto3 SDK)

Install the AWS SDK for Python:

pip install boto3

Example Python script for object detection:

python

import boto3

```
rekognition = boto3.client("rekognition")

response = rekognition.detect_labels(
    Image={"S3Object": {"Bucket": "your-bucket-name", "Name": "image.jpg"}}
)

for label in response["Labels"]:
    print(f"{label['Name']} - {label['Confidence']:.2f}%")
```

---

5. Deploy as a Serverless API (Optional)

- Use **AWS Lambda** with an **API Gateway** to process images dynamically
- Trigger **AWS Rekognition** when a new image is uploaded to S3

Conclusion

AWS Rekognition simplifies AI-based image analysis without requiring complex ML models. It can be integrated into applications for automated tagging, facial recognition, content moderation, and OCR.

Project 4: IoT Data Processing with AWS IoT Core – Store and analyze IoT sensor data in AWS.

AWS IoT Core is a managed cloud service that enables connected IoT devices to communicate securely with AWS services. This project focuses on collecting, storing, and analyzing IoT sensor data using AWS IoT Core, Amazon Kinesis, and Amazon S3.

Project Steps

Step 1: Set Up AWS IoT Core

- Go to the **AWS IoT Core** service in the AWS Console.
- Create a new **Thing** (IoT device) and download the security certificates.
- Attach an IoT policy to grant the necessary permissions.

Step 2: Configure an IoT Rule to Process Data

- Create an **IoT rule** to process incoming sensor data.
- Define an **SQL statement** to filter messages from an IoT topic.
- Set the **destination** for the data (e.g., Amazon Kinesis, DynamoDB, or S3).

Step 3: Simulate an IoT Device (Using MQTT)

- Install the **AWS IoT Device SDK** (Python or Node.js).
- Use an MQTT client to publish sensor data to the AWS IoT Core topic.

Step 4: Store Data in Amazon S3 or DynamoDB

- Configure AWS IoT rules to send data to **Amazon S3** for storage.
- Optionally, store real-time sensor readings in **DynamoDB** for quick access.

Step 5: Analyze IoT Data Using AWS Kinesis and QuickSight

- Stream data to **Amazon Kinesis** for real-time processing.
- Use **AWS Lambda** to process and transform IoT data.
- Visualize insights using **Amazon QuickSight**.

Step 6: Monitor IoT Data with AWS CloudWatch

- Enable **CloudWatch metrics** for IoT Core to track device connectivity and message rates.
- Set up **CloudWatch alarms** for abnormal activity detection.

Step 7: Secure the IoT Communication

- Use **AWS IoT Policies** to control device access.
- Enable **TLS encryption** for secure MQTT communication.
- Configure **AWS IAM roles** for authentication and authorization.

Conclusion

This project demonstrates how to securely process IoT sensor data using AWS IoT Core, store it in Amazon S3 or DynamoDB, analyze it with AWS Kinesis, and visualize it using QuickSight. By integrating CloudWatch monitoring, you can ensure real-time tracking and security for your IoT ecosystem.

---

12. Disaster Recovery & Backup

Project 1: Multi-Region Disaster Recovery Plan – Set up cross-region replication for S3 & RDS.

Introduction

A multi-region disaster recovery (DR) plan ensures business continuity by replicating critical data and applications across AWS regions. In this project, we will set up **Cross-Region Replication (CRR) for S3** and **Read Replica for RDS** to maintain availability in case of regional failures.

Steps to Implement the Multi-Region Disaster Recovery Plan

1. Set Up S3 Cross-Region Replication (CRR)

Step 1: Create Two S3 Buckets

- Navigate to **AWS S3 Console** → Create two buckets in different AWS regions (e.g., primary-bucket-us-east-1 and backup-bucket-us-west-2).
- Enable **Versioning** on both buckets.

Step 2: Configure IAM Role & Permissions

- Create an **IAM Role** with s3:ReplicateObject permission.
- Attach this role to your primary S3 bucket.

Step 3: Enable Cross-Region Replication

- Go to **primary S3 bucket** → Select **Replication Rules** → Create a new rule.
- Set **Destination Bucket** as the backup bucket (backup-bucket-us-west-2).
- Choose the **IAM Role** created earlier.
- Enable **Replica Modification Sync** (optional) to replicate ACLs, metadata, and encryption settings.

Step 4: Validate Replication

- Upload files to the primary bucket and check if they are replicated to the backup bucket.

---

2. Set Up RDS Cross-Region Read Replica

Step 1: Launch an RDS Instance

- Create an RDS instance in **Region A** (e.g., us-east-1).
- Choose **Multi-AZ Deployment** for high availability.

Step 2: Enable Automated Backups

- Go to **Modify RDS Instance** → Enable **Automated Backups**.

Step 3: Create a Cross-Region Read Replica

- Open **AWS RDS Console** → Select your primary RDS instance.
- Click on **Actions** → **Create Read Replica**.
- Select a different AWS region (e.g., us-west-2).
- Choose the instance class and storage settings.
- Enable **Replication Monitoring**.

Step 4: Configure Failover Mechanism

- In case of failure in **Region A**, manually promote the read replica to become the primary database.
- Update application connection settings to point to the new primary instance.

---

3. Set Up Route 53 for Failover Routing

- Create a **Route 53 DNS Record** with **Failover Routing Policy**.
- Define a **primary health check** for the main S3 bucket and RDS instance.
- Set up a **secondary health check** for the backup region.
- Route traffic to the backup region when the primary fails.

Testing & Validation

1. **Test S3 Replication:** Upload a file to the primary bucket and verify replication.
2. **Test RDS Read Replica:** Run queries on the read replica to check for data consistency.
3. **Simulate Failover:** Stop the primary instance and validate automatic traffic redirection to the backup region.

Conclusion

By implementing **S3 Cross-Region Replication (CRR)** and **RDS Read Replica**, we achieve high availability and disaster recovery for critical AWS resources. This setup minimizes downtime and ensures business continuity in case of regional failures.

Project 2: AWS Backup for EBS Snapshots & Recovery – Automate backups and restore EC2 instances.

Introduction

AWS Backup is a fully managed service that automates the backup and recovery of AWS resources, including EBS volumes. In this project, we will configure AWS Backup to create automated snapshots of Amazon EBS volumes and restore EC2 instances from these snapshots in case of data loss or failures.

Steps to Automate EBS Backups & Restore EC2 Instances

Step 1: Create a Backup Vault

1. Go to the **AWS Backup** service in the AWS Management Console.
2. Click on **Backup vaults** → **Create backup vault**.
3. Provide a **name** and select an **encryption key** (default or custom).
4. Click **Create backup vault**.

Step 2: Create a Backup Plan

1. Navigate to **AWS Backup** → **Backup plans**.
2. Click **Create a backup plan** → Choose **Build a new plan**.
3. Provide a **name** for the backup plan.
4. Select **Backup rule** and configure:
    - **Frequency**: Daily, weekly, or custom
    - **Backup vault**: Select the one created earlier
    - **Lifecycle**: Define retention period for backups

5. Click **Create plan**.

Step 3: Assign Resources to the Backup Plan

1. Under the created backup plan, go to **Assign resources**.
2. Provide a **resource assignment name**.
3. Select **IAM Role** (AWS Backup Default Role or create a custom role).
4. Choose **Resource type**: **EBS volumes**.
5. Select the **EBS volumes** attached to EC2 instances.
6. Click **Assign resources**.

Step 4: Test Backup Execution

1. Wait for the scheduled backup to trigger or manually start it.
2. Go to **AWS Backup → Backup jobs** to monitor progress.
3. Verify that EBS snapshots are created in **Amazon EC2 → Snapshots**.

Step 5: Restore EC2 Instance from Backup

1. Navigate to **AWS Backup → Backup vaults →** Select the vault.
2. Find the backup and click **Restore**.
3. Choose **EBS volume** as the restore target.
4. Specify the **availability zone** (same as the original EC2 instance).
5. Click **Restore backup** and wait for the process to complete.
6. Attach the restored EBS volume to a new or existing EC2 instance.

Conclusion

By automating EBS backups with AWS Backup, you ensure business continuity by regularly creating snapshots and quickly restoring EC2 instances in case of failures. This project helps in disaster recovery planning and compliance management.

Project 3: Automated AMI Creation & Cleanup – Schedule AMI snapshots and delete old backups.

Introduction

Amazon Machine Images (AMIs) are used to create EC2 instances with pre-configured environments. Automating AMI creation ensures that backups of instances are taken regularly, while cleanup of old AMIs helps in cost optimization by removing unnecessary snapshots. This project sets up a scheduled process using AWS Lambda, CloudWatch Events, and IAM policies to automate AMI creation and deletion of outdated backups.

Project Steps

Step 1: Create an IAM Role for Lambda

1. Navigate to **AWS IAM → Roles → Create Role**.
2. Choose **AWS Service** and select **Lambda** as the trusted entity.
3. Attach the following policies:
    - AmazonEC2FullAccess (to create and delete AMIs)
    - AWSLambdaBasicExecutionRole (to allow CloudWatch logging)
4. Name the role (e.g., Lambda-AMI-Backup-Role) and create it.

Step 2: Create a Lambda Function to Automate AMI Creation

1. Go to **AWS Lambda → Create Function →** Select **Author from scratch**.
2. Enter function name: AMI-Creation.
3. Choose **Python 3.x** as the runtime.
4. Assign the previously created IAM role.
5. Under the **Code** section, add the following script:

python

```
import boto3
import datetime

ec2 = boto3.client('ec2')

def lambda_handler(event, context):
    instances = ['i-xxxxxxxxxxxxxxxxx']  # Replace with your instance IDs
    date = datetime.datetime.now().strftime("%Y-%m-%d")

    for instance in instances:
        response = ec2.create_image(
            InstanceId=instance,
            Name=f"AMI-Backup-{instance}-{date}",
            Description="Automated AMI Backup",
            NoReboot=True
        )
        print(f"AMI {response['ImageId']} created for instance {instance}")
    return "AMI Creation Completed"
```

6. Click **Deploy**.

Step 3: Schedule AMI Creation using CloudWatch Events

1. Go to **Amazon EventBridge → Rules → Create rule**.
2. Name the rule (e.g., Schedule-AMI-Creation).
3. Select **Event Source → Schedule**.
4. Use a cron expression for daily execution (e.g., cron(0 2 * * ? *) for 2 AM UTC).
5. Choose **Target** as **Lambda Function**, and select AMI-Creation.
6. Click **Create**.

Step 4: Create a Lambda Function to Delete Old AMIs

1. Go to **AWS Lambda → Create Function → Author from scratch**.
2. Enter function name: AMI-Cleanup.
3. Choose **Python 3.x** as the runtime.
4. Assign the **Lambda-AMI-Backup-Role** created earlier.
5. Under the **Code** section, add this script:

```
python
import boto3
import datetime

ec2 = boto3.client('ec2')

def lambda_handler(event, context):
    retention_days = 7  # Change as needed
    delete_before = datetime.datetime.now() - datetime.timedelta(days=retention_days)
```

```
    images = ec2.describe_images(Owners=['self'])['Images']


    for image in images:
        creation_date = datetime.datetime.strptime(image['CreationDate'], "%Y-%m-%dT%H:%M:%S.%fZ")
        if creation_date < delete_before:
            ec2.deregister_image(ImageId=image['ImageId'])
            print(f"Deleted AMI {image['ImageId']}")


    return "Old AMIs Deleted"
```

6. Click **Deploy**.

Step 5: Schedule AMI Cleanup with CloudWatch Events

1. Go to **Amazon EventBridge → Rules → Create rule**.
2. Name the rule (e.g., Schedule-AMI-Cleanup).
3. Select **Event Source → Schedule**.
4. Use a cron expression (e.g., cron(0 3 * * ? *) for 3 AM UTC).
5. Choose **Target** as **Lambda Function**, and select AMI-Cleanup.
6. Click **Create**.

Outcome

- The first Lambda function automatically creates an AMI for specified EC2 instances daily.
- The second Lambda function removes AMIs older than the retention period (e.g., 7 days).
- CloudWatch schedules ensure automated execution.
- Reduces manual intervention and optimizes storage costs.