Ghulam Ishaq Khan Institute (GIKI)

DevOps Assignment # 3

Subject: DevOps	**Course Code:** CS - 328 – Spring - 2025
Class: BS CY **Batch:** Fall – 2022	**Submission Deadline:** 17 March 2025 Monday (11:00 AM)
Course Instructor: Muhammad Ahmad Nawaz	**Total Marks:** 15
Course TA: Muhammad Hamza	

Note (Read notes & instructions first)

- First, read the instructions and statements of each exercise/question carefully then write the solution.
- *For Assignment:*
  - The name of your zip file should contain your assignment number and your roll number as shown in following example, For Example if your roll number is 2022532 and you have done assignment number 1 then the name of file should be as ---> 2022532_1.zip
    - monitor_and_push.sh
    - config.cfg
    - README.md
    - Any additional files used for testing.
  - Then upload that pdf file at Microsoft teams. Remember the sequence of pages should be right.

CHEATING/COPY CASE or LATE SUBMISSION will be graded as STRAIGHT ZERO MARKS.

Problem Statement

An e-commerce platform requires a payment processing backend to handle high transaction volumes (e.g., 10,000 requests/minute during peak sales events like Black Friday) while adhering to PCI DSS security standards. The system must be deployed on AWS with a custom VPC for network isolation, a tailored AMI for consistent EC2 deployments, and robust traffic control via NACLs and Security Groups. An Application Load Balancer (ALB) must ensure scalability and high availability across multiple Availability Zones (AZs). To minimize operational risks and enable rapid iteration, the deployment process must be fully automated using GitHub Actions and Bash scripting, leveraging AWS CLI for infrastructure provisioning. The solution should output a functional API endpoint returning payment confirmation responses, testable via the ALB DNS.

Task: Design, Implement, and Automate the Payment Processing Backend

Requirements

Design and deploy a payment processing backend on AWS with the following specifications:

1. **VPC Configuration:**
   - CIDR: 10.0.0.0/16.
   - Public Subnets: 10.0.1.0/24 (us-east-1a), 10.0.3.0/24 (us-east-1b) for ALB and NAT Gateways, routed to an Internet Gateway (0.0.0.0/0).
   - Private Subnets: 10.0.2.0/24 (us-east-1a), 10.0.4.0/24 (us-east-1b) for EC2 instances, routed to NAT Gateways.
   - Tag all resources with Application: Payment-Processing.
2. **Custom AMI Creation:**
   - Base AMI: Amazon Linux 2 (latest).
   - Builder Instance: t3.micro in a public subnet.
   - Bash Script (build-ami.sh):
     - Install Nginx (reverse proxy) and Node.js (v18).
     - Configure Nginx to proxy TCP 80 to Node.js on TCP 3000.

        - Deploy a Node.js API at /opt/payment-api/server.js returning { "status": "Payment Processed", "timestamp": "<ISO-date>" }.
        - Persist API with a systemd service.
        - Create AMI (payment-api-ami) using aws ec2 create-image --no-reboot.
3. **EC2 Instance Deployment:**
    - AMI: payment-api-ami.
    - Instance Type: t3.small.
    - Deploy one instance per private subnet (2 total).
    - Tag instances with Role: Payment-Server.
4. **NACL Configuration:**
    - Public Subnets:
        - Inbound: TCP 80 from 0.0.0.0/0, TCP 1024-65535 from 0.0.0.0/0.
        - Outbound: All traffic to 0.0.0.0/0.
    - Private Subnets:
        - Inbound: TCP 80 from 10.0.1.0/24 and 10.0.3.0/24.
        - Outbound: TCP 443 to 0.0.0.0/0, TCP 1024-65535 to 0.0.0.0/0.
        - Deny all other traffic.
5. **Security Groups:**
    - EC2 Security Group (sg-payment-api):
        - Inbound: TCP 80 from ALB Security Group.
        - Outbound: All traffic to 0.0.0.0/0.
    - ALB Security Group (sg-alb):
        - Inbound: TCP 80 from 0.0.0.0/0.
        - Outbound: TCP 80 to sg-payment-api.
6. **Application Load Balancer:**
    - Deploy ALB in public subnets (10.0.1.0/24, 10.0.3.0/24).
    - Target Group: HTTP, TCP 80, health check on / (200 OK, 5s interval, 2 healthy threshold), register both EC2 instances.
    - Listener: TCP 80 → Target Group.
7. **GitHub and GitHub Actions:**
    - Repository Structure:
        - scripts/build-ami.sh: AMI creation script.
        - scripts/deploy.sh: Infrastructure provisioning script.
        - .github/workflows/deploy.yml: CI/CD pipeline.
    - Workflow:
        - Trigger: Push to main.
        - Runner: ubuntu-latest.
        - Steps: Configure AWS credentials (via GitHub Secrets: AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY), run build-ami.sh, run deploy.sh.
8. **Bash Scripting:**
    - build-ami.sh:
        - Install dependencies, configure Nginx/Node.js, ensure idempotency (e.g., || true).
        - Output AMI ID to ami-id.txt.
    - deploy.sh:
        - Chain AWS CLI commands with error handling (e.g., set -e, || exit 1).
        - Capture resource IDs dynamically (e.g., vpc_id=$(aws ec2 create-vpc ...)).

Actions

- Use AWS CLI to provision all infrastructure components (VPC, subnets, NACLs, Security Groups, EC2, ALB).
- Develop and test Bash scripts (build-ami.sh, deploy.sh) for automation.
- Commit scripts and workflow to a GitHub repository and execute the pipeline.
- Verify deployment by curling the ALB DNS to confirm the API response.

Deliverables

1. **GitHub Repository:**
    - scripts/build-ami.sh: AMI builder script with full configuration.
    - scripts/deploy.sh: Complete AWS CLI-based deployment script.
    - .github/workflows/deploy.yml: GitHub Actions workflow file.
2. **Documentation:**
    - README.md including:
        - Full AWS CLI commands used in deploy.sh.

        - Instructions to set up GitHub Secrets for AWS credentials.
        - Validation steps (e.g., curl <ALB-DNS>).
3. **Evidence:**
    - Logs or screenshots demonstrating:
        - Successful AMI creation (AMI ID output).
        - EC2 instances responding with { "status": "Payment Processed", "timestamp": "<ISO-date>" }.
        - ALB DNS returning the API response.

Evaluation Criteria

- **Security:** NACLs and Security Groups enforce PCI DSS-compliant isolation.
- **Scalability:** Multi-AZ ALB and EC2 deployment handle high transaction loads.
- **Automation:** GitHub Actions pipeline deploys the system end-to-end without manual steps.
- **Robustness:** Bash scripts include error handling, idempotency, and dynamic resource referencing.
- **Completeness:** All components (VPC, AMI, EC2, NACLs, Security Groups, ALB) are fully implemented and documented.