Day 2 — Networking, Database & Serverless Practical Labs (CloudOps Style)

AWS CLI (Linux/macOS syntax) | Full step-by-step labs with verification & cleanup

Lab 1: Custom VPC with Public & Private Subnets (NAT Gateway)

Objective: Create a VPC with 2 public and 2 private subnets, configure routing, NAT gateway, and verify internet access from private subnet via NAT.
Architecture (textual):

```
Internet -> IGW -> Public Subnet (Bastion / ALB)
Private Subnet -> NAT Gateway (in Public Subnet) -> Internet via IGW
```

Console Steps:
1. VPC → Create VPC (CIDR 10.0.0.0/16).
2. Create 2 public subnets and 2 private subnets across AZs.
3. Create and attach Internet Gateway (IGW).
4. Create NAT Gateway in a public subnet and allocate an Elastic IP.
5. Update route tables: public RT -> IGW; private RT -> NAT Gateway.

CLI Steps:

```
# create VPC
aws ec2 create-vpc --cidr-block 10.0.0.0/16

# (example) describe vpcs and subnets
aws ec2 describe-vpcs
aws ec2 describe-subnets --filters "Name=vpc-id,Values=<vpc-id>"

# create nat gateway (requires allocate-address then create-nat-gateway)
aws ec2 allocate-address --domain vpc
aws ec2 create-nat-gateway --subnet-id subnet-xxxxxxxx --allocation-id eipalloc-xxxxxxxx
```

Verification:
- Launch EC2 in private subnet and confirm it can reach internet (e.g., `curl https://ifconfig.co`).
- Confirm route tables and NAT gateway metrics in console.

Cleanup:

```
# delete resources via console or CLI: delete nat gateway, release EIP, delete subnets, delete v
# Example: release address
aws ec2 release-address --allocation-id eipalloc-xxxxxxxx
```

Notes: NAT gateways incur hourly and data processing charges; consider NAT instances for cost control in labs.

Lab 2: Application Load Balancer (ALB) with EC2 Auto Scaling

Objective: Deploy an ALB fronting an Auto Scaling Group of EC2 instances running a simple web app; configure health checks and scale policy.

Architecture (textual):

```
Internet -> ALB -> Auto Scaling Group (EC2 in private subnets) -> EBS volumes
```

Console Steps:

1. Create a Launch Template/Configuration with user data that installs a web server.
2. Create Auto Scaling Group (ASG) across 2 AZs, desired capacity 2.
3. Create ALB and target group, register ASG instances.
4. Configure health checks (HTTP:80/).

CLI Steps (simplified):

```
# create target group
aws elbv2 create-target-group --name my-tg --protocol HTTP --port 80 --vpc-id <vpc-id>

# create load balancer
aws elbv2 create-load-balancer --name my-alb --subnets subnet-aaa subnet-bbb

# register targets (after instances exist)
aws elbv2 register-targets --target-group-arn <tg-arn> --targets Id=i-0123456789abcdef0
```

Verification:

- Access ALB DNS and ensure requests are balanced across instances.
- Check ASG Activity History and CloudWatch metrics for scaling events.

Cleanup:

```
# delete ASG, delete load balancer, delete target groups, terminate instances
aws autoscaling delete-auto-scaling-group --auto-scaling-group-name my-asg --force-delete
aws elbv2 delete-load-balancer --load-balancer-arn <alb-arn>
```

Notes: Use lifecycle hooks and health checks to ensure graceful termination and avoid request drops.

Lab 3: RDS MySQL Multi-AZ Deployment and Connectivity

Objective: Deploy RDS MySQL with Multi-AZ for high availability and connect from EC2 instance.
Architecture (textual):

```
EC2 (app) -> Security Group -> RDS MySQL (Multi-AZ) -> EBS storage
```

Console Steps:
1. RDS → Create database → MySQL → Free Tier (if available) → Multi-AZ enabled.
2. Select VPC and private subnets; ensure security group allows app EC2 to connect on port 3306.
3. Set backup retention and maintenance window as required.

CLI Steps (example):

```
# create a DB instance (example)
aws rds create-db-instance --db-instance-identifier mydb --db-instance-class db.t3.micro --engin
```

Verification:
- From EC2, install mysql client and connect: `mysql -h -u admin -p`
- Check RDS console for Multi-AZ status and recent failover events.

Cleanup:

```
# delete the DB instance (optionally skip final snapshot in labs)
aws rds delete-db-instance --db-instance-identifier mydb --skip-final-snapshot --delete-automate
```

Notes: RDS is managed but can be costly; use short-lived instances for labs or snapshot+restore strategy.

Lab 4: Lambda Function with S3 Trigger

Objective: Create a Lambda function that triggers on S3 object creation and logs the event.
Architecture (textual):

```
S3 Bucket (uploads) -> Event Notification -> Lambda (process) -> CloudWatch Logs
```

Console Steps:
1. Create S3 bucket and configure Event Notifications on ObjectCreated to invoke Lambda.
2. Create Lambda function (Python) with basic handler to log event details.
3. Grant Lambda necessary IAM permissions (S3 read) via role.

CLI Steps:

```
# create function (zip file with handler)
aws lambda create-function --function-name s3EventHandler --runtime python3.9 --role arn:aws:iam

# add permission for s3 to invoke lambda (if needed)
aws lambda add-permission --function-name s3EventHandler --statement-id s3invoke --action "lambd
```

Verification:
- Upload a file to S3 and check CloudWatch Logs for the Lambda execution and event payload.

Cleanup:

```
# delete lambda and remove s3 notification
aws lambda delete-function --function-name s3EventHandler
# remove event notification in console or using s3api
```

Notes: Use dead-letter queues for failed Lambda executions and monitor retries.

Lab 5: API Gateway REST API integrated with Lambda

Objective: Build a simple REST endpoint with API Gateway that invokes a Lambda function and returns JSON.

Architecture (textual):

```
Client -> API Gateway -> Lambda -> (optional) DynamoDB / RDS
```

Console Steps:

1. Create Lambda (simple JSON response).
2. API Gateway → Create REST API → New Resource `/hello` → Create Method GET → Integration: Lambda Function.
3. Deploy API to a stage (e.g., prod).

CLI Steps (example):

```
# create rest api (simplified)
aws apigateway create-rest-api --name 'MyAPI'

# create resource, method and integrate with lambda via CLI is verbose; prefer console for labs
```

Verification:

- Call the endpoint: `curl https://.execute-api..amazonaws.com/prod/hello` and expect JSON response.

Cleanup:

```
# delete rest api
aws apigateway delete-rest-api --rest-api-id <api-id>
# delete lambda if not needed
```

Notes: For production, consider usage plans, API keys, WAF integration, and throttling settings.