AWS DevOps Upskilling Program - Guided Lab Assignments

Lab 1: Lab Guide

AWS & DevOps Foundations

Lab 1.1: AWS CLI Setup & Multi-Region Resource Deployment

Duration: 2-3 hours
Objective: Master AWS CLI basics and understand global infrastructure

Prerequisites

- AWS Account with appropriate permissions
- GitHub account
- Local terminal/command line access

Learning Outcomes

By completing this lab, you will:

- Configure AWS CLI with multiple regional profiles
- Deploy resources across different AWS regions using automation
- Understand region-specific configuration requirements
- Implement basic cost tracking with resource tagging

Part 1: Environment Setup

Task 1.1: Install and verify AWS CLI v2

- Install AWS CLI appropriate for your operating system
- Verify the installation is successful
- **Hint:** Use the `--version` flag to check

Task 1.2: Configure your primary AWS CLI profile

- Set up AWS CLI with your credentials
- Choose `us-east-1` as your default region
- Use JSON as your output format
- **Hint:** The `aws configure` command is your starting point

Task 1.3: Create named profiles for additional regions

- Create a profile named `eu-region` pointing to `eu-west-1`
- Create a profile named `asia-region` pointing to `ap-southeast-1`
- **Hint:** Use the `--profile` flag with the configure command

Part 2: Deploy Resources Across Regions

Task 2.1: Initialize your project repository

- Create a new directory called `week1-aws-cli-deployment`
- Initialize it as a Git repository
- Create a README.md file describing the project
- Make your initial commit

Task 2.2: Create an S3 bucket deployment script Create a bash script named `deploy-s3-buckets.sh` that:

- Deploys S3 buckets to three regions: us-east-1, eu-west-1, and ap-southeast-1
- Uses a timestamp in the bucket name to ensure uniqueness
- Handles the special case for us-east-1 (doesn't require LocationConstraint)
- Enables versioning on all created buckets

Hints:

- Use an array to store your region list
- The `date` command can generate timestamps
- S3 bucket names must be globally unique
- Research: `aws s3api create-bucket` vs `aws s3 mb`
- Check AWS documentation for the LocationConstraint parameter

Expected output format:

```
Creating bucket in us-east-1...
Bucket amalitech-devops-us-east-1-20241001-143022 created successfully
Creating bucket in eu-west-1...
…
```

Task 2.3: Execute and verify your deployment

- Make your script executable
- Run the deployment script
- Verify all buckets were created successfully
- Check the location of each bucket

Hints:

- Use `chmod` to make scripts executable
- `aws s3 ls` lists all your buckets
- `aws s3api get-bucket-location` shows where a bucket is located

Task 2.4: Create a resource inventory script Create a script named `list-resources.sh` that:

- Lists all S3 buckets in your account
- Displays each bucket name alongside its region
- Handles the special case where us-east-1 returns "None"

Expected output format:

```
=== S3 Buckets Inventory ===
Bucket: my-bucket-name | Region: us-east-1
Bucket: another-bucket | Region: eu-west-1
...
```

Task 2.5: Push your work to GitHub

- Add all your scripts to Git
- Commit with a descriptive message
- Create a remote repository and push your code

Part 3: Cost Analysis

Task 3.1: Research and document S3 pricing Create a document named `cost-analysis.md` that includes:

- Standard S3 storage pricing for each of the three regions
- Estimated monthly cost for storing 100GB in each region
- Data transfer costs between regions
- Any pricing differences you observe

Hints:

- Visit the AWS S3 pricing page
- Consider storage class (use Standard for this calculation)
- Don't forget to account for requests/retrievals

Task 3.2: Implement resource tagging

- Tag all your S3 buckets with appropriate metadata
- Include at minimum: Project and Environment tags
- **Hint:** Use `aws s3api put-bucket-tagging` with a TagSet

Deliverables Checklist

Submit the following:

- GitHub repository URL containing:
  - `deploy-s3-buckets.sh` script
  - `list-resources.sh` script
  - `cost-analysis.md` document
  - README.md with project description
- Screenshot showing successful bucket creation in 3 regions
- Screenshot of resource inventory output
- Screenshot of AWS Console showing tagged buckets

Evaluation Criteria

- **Functionality (40%):** Scripts work correctly and deploy resources as specified
- **Code Quality (20%):** Scripts are well-commented and follow best practices
- **Documentation (20%):** Clear README and accurate cost analysis
- **Completeness (20%):** All deliverables submitted with proper evidence

Troubleshooting Tips

- If bucket creation fails, check that your bucket name is globally unique
- If you get permission errors, verify your IAM user has S3 full access
- For "None" region returns, this is normal for us-east-1 (handle it in your code)
- If scripts won't execute, check file permissions with `ls -l`

Lab 1.2: Well-Architected Framework Assessment

Duration: 2 hours
Difficulty: ⭐⭐⭐☆☆
Objective: Apply Well-Architected Framework principles to analyze and improve a sample architecture

Learning Outcomes

By completing this lab, you will:

- Understand the six pillars of the AWS Well-Architected Framework
- Assess an architecture against best practices
- Identify risks and propose improvements
- Prioritize improvements based on impact and effort

Part 1: Architecture Design

Task 1.1: Set up your project structure

- Create a new directory called `well-architected-assessment` in your repository
- Initialize the necessary files for documentation

Task 1.2: Design a 3-tier web application architecture Create an architecture diagram that includes:

- **Web Tier:** EC2 instances behind an Application Load Balancer (ALB)
- **Application Tier:** EC2 instances in a private subnet
- **Database Tier:** RDS database instance

Your diagram should show:

- A VPC spanning 2 Availability Zones
- Public and private subnets in each AZ
- Internet Gateway for public internet access
- NAT Gateway for private subnet internet access
- All necessary connections and traffic flow

Hints:

- Use free tools like draw.io (diagrams.net) or Lucidchart
- Consider how traffic flows from the internet to the database
- Think about which resources need public IPs
- Reference AWS architecture diagrams for inspiration

Part 2: Well-Architected Assessment

Task 2.1: Create your assessment document Create a file named `well-architected-review.md` that evaluates your architecture against all six pillars of the Well-Architected Framework.

For EACH pillar, you must include:

1. **Current State:** Describe what's currently implemented
2. **Risks:** Identify at least 2-3 potential risks or gaps
3. **Improvements:** Suggest at least 3 specific improvements

The Six Pillars:

1. **Operational Excellence** - How well can you run and monitor the system?
2. **Security** - How is data and infrastructure protected?
3. **Reliability** - Can the system recover from failures?
4. **Performance Efficiency** - How efficiently does it use resources?
5. **Cost Optimization** - Are you getting the best value?
6. **Sustainability** - What's the environmental impact?

Hints:

- Research each pillar on the AWS Well-Architected Framework documentation
- Think about: What happens if an AZ fails? How do you handle traffic spikes?
- Consider: Are there single points of failure? How are databases backed up?
- Ask yourself: Can unauthorized users access data? How are costs monitored?

Task 2.2: Check service quotas

- Use AWS CLI to check your current EC2 service quotas
- Document any quotas that might limit your architecture's scalability
- **Hint:** The `aws service-quotas` command can list quotas for different services

Part 3: Prioritization Matrix

Task 3.1: Create an improvements priority document Create a file named `improvements-priority.md` that ranks ALL your suggested improvements.

For each improvement, rate it on:

- **Impact:** High / Medium / Low (effect on business or users)
- **Effort:** High / Medium / Low (time and resources needed)
- **Priority:** P0 (Critical) / P1 (High) / P2 (Nice to have)

Format your prioritization as a table:

```
| Improvement | Pillar | Impact | Effort | Priority | Justification |
|-------------|--------|--------|--------|----------|---------------|
| Example     | Security | High | Low | P0 | [Why this matters] |
```

Hints:

- P0 items are typically High Impact + Low Effort OR Critical security issues
- P1 items are High Impact + High Effort OR Medium Impact + Low Effort
- Consider quick wins (High Impact + Low Effort) first
- Security issues should generally be higher priority

Deliverables Checklist

Submit the following:

- Architecture diagram (PNG, JPG, or PDF format)
- `well-architected-review.md` with complete assessment of all 6 pillars
- `improvements-priority.md` with prioritized recommendations
- Documentation of service quotas check
- All files committed to your GitHub repository

Evaluation Criteria

- **Diagram Quality (20%):** Clear, complete architecture with proper AWS components
- **Assessment Depth (40%):** Thorough analysis of all 6 pillars with specific, actionable recommendations
- **Prioritization (20%):** Logical ranking of improvements with clear justification
- **Technical Accuracy (20%):** Recommendations demonstrate understanding of AWS best practices

Challenge Questions

1. How would you modify this architecture to support multiple regions?
2. What AWS services could you use to automate the deployment of this architecture?
3. How would you implement disaster recovery for this system?

Resources

- AWS Well-Architected Framework whitepaper
- AWS Architecture Center
- AWS Well-Architected Tool (in AWS Console)

Lab 1.3: Shared Responsibility Model & Security Audit

Duration: 1-2 hours
Objective: Understand AWS Shared Responsibility Model and perform a basic security audit

Learning Outcomes

By completing this lab, you will:

- Understand the division of security responsibilities between AWS and customers
- Create a comprehensive responsibility matrix for common AWS services
- Implement an automated security audit script
- Identify and document security gaps with remediation plans

Part 1: Shared Responsibility Matrix

Task 1.1: Research and document the Shared Responsibility Model Create a file named `shared-responsibility-matrix.md` with a detailed table.

Your table must include at least **10 AWS services** with:

- Service name
- What AWS is responsible for
- What the customer (you) is responsible for

Services to include (minimum):

- EC2
- RDS
- S3
- Lambda
- ECS/Fargate
- VPC
- IAM
- CloudFront
- DynamoDB
- EBS

Format example:

markdown

```
| Service | AWS Responsibility | Customer Responsibility |
|---------|---------------------------|-------------------------|
```

| EC2 | [What AWS manages] | [What you manage] |

Hints:

- Research the AWS Shared Responsibility Model documentation
- For EC2: Who patches the OS? Who manages the hypervisor?
- For S3: Who ensures durability? Who sets bucket permissions?
- Think about: Infrastructure, platform, software, data

Part 2: Automated Security Audit

Task 2.1: Create a security audit script Create a bash script named `security-audit.sh` that checks for common security issues:

Your script should audit at least these areas:

1. **S3 Bucket Security**
   - Check for buckets without public access blocks
   - Identify buckets without encryption
2. **IAM Security**
   - Check if root account MFA is enabled
   - Verify password policy is configured
   - Generate a credential report
3. **Additional checks** (implement at least 2 more):
   - VPC flow logs enabled?
   - CloudTrail enabled?
   - Unused IAM access keys?
   - Security groups with overly permissive rules?

Output requirements:

- Clear pass/fail indicators (✓ for pass, ⚠ for warnings)
- Organized by security category
- Summary of total issues found

Hints:

- Use `aws iam get-account-summary` for account-level info
- `aws s3api` commands can check bucket configurations
- `aws iam generate-credential-report` creates user access reports
- Test each check individually before combining
- Handle errors gracefully (use `2>&1` and check exit codes)

Task 2.2: Run the audit and document results

- Execute your security audit script
- Save the output to a file named `audit-results.txt`
- Take a screenshot of the audit running

Part 3: Remediation Planning

Task 3.1: Create a remediation plan Based on your audit results, create a file named `remediation-plan.md` that:

1. **Lists all identified issues** organized by severity:
    - Critical (fix immediately)
    - High (fix within 1 week)
    - Medium (fix within 2 weeks)
    - Low (fix within 1 month)
2. **For each issue, provide:**
    - Clear description of the problem
    - Security impact explanation
    - Step-by-step remediation instructions
    - Estimated time to fix
    - Assigned owner (can be "DevOps Team" for this lab)

Format template:

```
markdown
## Critical Issues


### Issue 1: [Title]
- **Description:** What's wrong?
- **Impact:** What could happen?
- **Remediation Steps:**
  1. Step 1
  2. Step 2
  3. Step 3
- **Time Estimate:** X hours
- **Owner:** [Team/Person]
- **Deadline:** [Date]
```

Task 3.2: Implement at least one fix

- Choose one issue from your audit
- Implement the fix using AWS CLI or Console
- Document the commands used
- Re-run the relevant audit check to verify the fix

Deliverables Checklist

Submit the following:

- `shared-responsibility-matrix.md` with at least 10 services documented
- `security-audit.sh` script with multiple security checks
- `audit-results.txt` showing script output
- Screenshot of audit script execution
- `remediation-plan.md` with organized, prioritized issues
- Documentation of at least one implemented fix
- All files committed to GitHub repository

Evaluation Criteria

- **Matrix Completeness (25%):** Accurate documentation of 10+ services with clear responsibility divisions
- **Audit Script (35%):** Working script that checks multiple security aspects with clear output
- **Remediation Plan (25%):** Well-organized plan with actionable steps and proper prioritization
- **Implementation (15%):** Evidence of understanding through at least one applied fix

Security Best Practices to Consider

- Never commit AWS credentials to Git
- Use IAM roles instead of access keys where possible
- Enable MFA on all accounts
- Follow principle of least privilege
- Encrypt data at rest and in transit
- Enable logging and monitoring on all resources

Resources

- AWS Shared Responsibility Model documentation
- AWS Security Best Practices whitepaper
- AWS Well-Architected Security Pillar
- AWS CLI documentation for IAM and S3