5 AWS Projects That Can Launch Your Cloud Career (2025 Edition)

by @abhishek_ctrl_alt_delete

Introduction

Welcome to your practical roadmap for building a strong AWS portfolio and landing your first cloud role.

If you're serious about starting a cloud or DevOps career, this guide will walk you through five hands-on AWS projects that demonstrate real-world architectures, essential services, and interview-ready skills.

These projects are carefully designed to cover almost everything hiring managers expect — from hosting a secure website to building scalable, event-driven, and containerized systems.

Each project builds upon the previous one, taking you step by step from beginner-friendly setups to professional-grade architectures used in real production environments.

By the end, you'll have a portfolio that proves your skills and stories to confidently share in interviews.

How to Use This Guide

Before you begin, create an AWS Free Tier account. You'll get free credits that are more than enough to complete these projects.

Spend a little time understanding two AWS fundamentals first:

- AWS Global Infrastructure – how Regions and Availability Zones work
- AWS IAM (Identity & Access Management) – how permissions and access control work

Start with the AWS Management Console to get a visual understanding of how services connect.

🧰 Project 1 — Host a Static Website on AWS (Beginner)

Goal: Host a portfolio/static site on S3 + CloudFront + Route 53 + ACM + WAF

Prereqs: AWS account, basic HTML/CSS (even a one-page site), AWS Console, optional AWS CLI.

Estimated time: 1–2 hours

Steps

1. Create website files
    - Make index.html and optional 404.html, CSS and assets in a folder site/.
2. Create S3 bucket
    - In AWS Console → S3 → Create bucket.
    - Bucket name = your domain or yourname-portfolio. Region: any.
    - Uncheck “Block all public access” only temporarily (we will secure via CloudFront/OAC). Click Create.
3. Enable static website hosting (optional)
    - In bucket Properties → Static website hosting → Enable → set Index document index.html.
    - (Note: For production use with CloudFront, you typically keep bucket private and use Origin Access Control.)
4. Upload website files
    - Upload site/* to the bucket (drag & drop in console) or use aws s3 sync site/ s3://your-bucket-name
5. Create CloudFront distribution

- Console → CloudFront → Create distribution
- Origin = your S3 bucket (choose the bucket endpoint, prefer the S3 REST endpoint)
- Set Origin Access Control (OAC) so CloudFront can read S3 while blocking public access.
- Default Root Object = index.html
- Cache & behavior: use defaults initially.

6. Make S3 private + configure OAC
    - Block public access on bucket.
    - Add a bucket policy (or use CloudFront OAC) to allow CloudFront to get objects.
7. Get SSL certificate
    - Console → AWS Certificate Manager → Request public certificate
    - Choose your domain name (e.g., www.yourdomain.com) and validation via DNS (recommended).
8. Register domain / configure DNS
    - If you have a domain, use Route 53 or add CNAME/Alias in your DNS provider to point to CloudFront distribution.
    - In Route 53 create an Alias record pointing domain to CloudFront domain.
9. Configure WAF (optional but recommended)
    - Console → WAF → Web ACL → Create -> attach to CloudFront and add basic managed rules (SQLi, XSS).
10. Test
    - Browse your domain. Validate index.html loads, HTTPS works, and resources load via CloudFront (check response headers).
11. Cost & cleanup tip
    - Use AWS Free Tier. Delete resources if practicing repeatedly (S3, CloudFront distributions, Route53 hosted zones are billed).

Extensions / Interview talking points

- Show you used OAC to prevent public bucket access.
- Explain CloudFront caching & TTL decisions.
- Add automatic invalidation via CLI after content updates: aws cloudfront create-invalidation --distribution-id XYZ --paths "/*"

🏗️ Project 2 — Three-Tier Web App (Intermediate)

Goal: Deploy a web UI → app servers → RDS database inside VPC with ALB + Auto Scaling

Prereqs: AWS account, basic web app (Node/Flask/Java app), basic Linux commands, Git.

Estimated time: 3–6 hours (longer if building app from scratch)

Steps

1. Prepare a simple web app
    - Example: small Node Express or Python Flask app that reads/writes to a database. Use sample repo or scaffold quickly.
2. Create VPC with subnets
    - Console → VPC → Create VPC or use default VPC.
    - Create at least two public subnets (for ALB) and two private subnets (for EC2 app and RDS) across different AZs.
3. Create Security Groups
    - ALB SG: allow HTTP/HTTPS from 0.0.0.0/0.
    - EC2 app SG: allow traffic from ALB SG on app port (e.g., 3000).
    - RDS SG: allow traffic from EC2 app SG on DB port (3306/Postgres).

4. Deploy RDS
    - Console → RDS → Create database (MySQL/Postgres/MariaDB).
    - Choose Multi-AZ for high availability (optional), place in private subnets, enable automated backups.
    - Note DB username/password (store securely).
5. Create an AMI or use user-data for EC2
    - Prepare an EC2 launch configuration/User Data script to install app runtime (Node/Python), clone your repo, npm start or run gunicorn.
6. Create EC2 Auto Scaling Group
    - Launch Template/Configuration → specify AMI, instance type, user-data script, SG (app SG).
    - Auto Scaling Group across private subnets; set min=1, desired=2, max=4 (sample).
7. Create Application Load Balancer (ALB)
    - Console → EC2 → Load Balancers → Create ALB in public subnets.
    - Target group → register public/private IPs of EC2 Auto Scaling Group (use health checks).
8. Configure session / sticky behavior (optional)
    - For stateful apps, consider sticky sessions or move session store to Redis.
9. Configure DNS
    - Route53 → create record pointing domain to ALB.
10. Testing
    - Access ALB DNS name / domain. Test creating and retrieving data to/from RDS.
11. Monitoring
    - CloudWatch metrics for CPU, ALB request count, RDS storage.
12. Cost & security tips

- Use t3.micro for dev (Free Tier may apply), turn off resources after demo.
- Use IAM roles for EC2 to pull secrets from AWS Secrets Manager instead of hardcoding credentials.

Extensions / Interview talking points

- Explain VPC subnet design and security groups as micro firewalls.
- Describe Auto Scaling policy (CPU threshold).
- Show DB backup/restore and multi-AZ failover demo.

⚡ Project 3 — Serverless REST API (Intermediate)

Goal: Build a REST API using API Gateway, Lambda, DynamoDB, and Cognito auth

Prereqs: Node or Python knowledge to write Lambda functions, AWS Console, basic JSON.

Estimated time: 2–4 hours

Steps

1. Design the API
    - Decide endpoints: e.g., GET /items, POST /items, GET /items/{id}.
2. Create DynamoDB table
    - Console → DynamoDB → Create table (Partition key id), start with On-Demand capacity (cheaper for small usage).
3. Write Lambda functions
    - Create functions for each endpoint in Node/Python.
    - Use AWS SDK to interact with DynamoDB.

    - Example POST /items Lambda: parse body, generate UUID, put item in table.

4. Create API Gateway
    - Console → API Gateway → Create HTTP API or REST API.
    - Create routes and integrate each route with the corresponding Lambda function.
5. Set up Cognito for auth (optional but recommended)
    - Console → Amazon Cognito → Create User Pool.
    - Enable app client and create test users.
    - Attach Cognito Authorizer to API Gateway routes (so only authenticated requests can call POST).
6. Add CORS
    - If you have a front-end or Postman, ensure CORS is enabled.
7. Test with Postman / curl
    - curl to GET/POST and verify Lambda logs in CloudWatch show execution.
8. Add CloudWatch logs & alarms
    - Ensure Lambda log groups exist, create basic alarm for error count.
9. Secure APIs with WAF (optional)
10. Cost & dev tips
    - Use small memory for Lambdas for cost; use On-Demand DynamoDB for learning.

Extensions / Interview talking points

- Discuss cold starts & memory tuning.
- Explain API Gateway stages / deployment lifecycle.
- Show adding versioning and aliasing to Lambda for blue/green releases.

🔔 Project 4 — Event-Driven Application (Advanced)

Goal: Build async pipeline using SNS → SQS → Lambda with DLQ and retries

Prereqs: Node/Python for Lambda, understanding of Pub/Sub concepts.

Estimated time: 2–4 hours

Steps

1. Define the use case
    - Example: Order placed → publish order event → multiple consumers (inventory, notifications) process asynchronously.
2. Create SNS topic
    - Console → SNS → Create Topic orders-topic.
3. Create SQS queues
    - Create inventory-queue, email-queue, and Dead-Letter Queue dlq.
    - Set redrive policy so failed messages go to DLQ after N attempts.
4. Subscribe SQS queues to SNS
    - For each SQS queue, subscribe it to the SNS topic (SNS → Subscriptions → Create subscription).
    - Optionally add filter policies to SNS so only some messages go to certain queues.
5. Create Lambda processors
    - Create Lambdas that are triggered by SQS (inventory-processor, email-processor).
    - Implement idempotency in Lambdas (check if order already processed using DynamoDB or item status).

6. Publish an event
    - Simulate order creation (via API Gateway or console script) that publishes to SNS.
7. Handle failures
    - Make Lambda throw an error for one test message and verify it moves to DLQ after retries.
8. Monitoring & visibility
    - CloudWatch metrics for SNS Published/Failed, SQS ApproximateNumberOfMessages, Lambda errors.
9. Scalability & cost tips
    - SQS scales automatically; use batching in Lambda to process multiple messages per invocation for efficiency.

Extensions / Interview talking points

- Explain asynchronous decoupling benefits and eventual consistency.
- Talk about idempotency design patterns and reasons for DLQs.
- Show message filtering to reduce unnecessary processing.

🐳 Project 5 — Containerized Microservices (Expert)

Goal: Containerize app, push images to ECR, deploy with ECS (Fargate) or EKS (Kubernetes)

Prereqs: Docker basics, simple multi-service app (e.g., front-end + API + DB), AWS Console.

Estimated time: 4–8 hours (EKS longer)

Steps (ECS Fargate — simpler for beginners)

1. Containerize services
    - Add Dockerfile for each service. Example: docker build -t myapp-api:latest .
2. Create ECR repositories
    - Console → ECR → Create repo myapp-api.
3. Authenticate & push images
    - aws ecr get-login-password | docker login --username AWS --password-stdin <account>.dkr.ecr.region.amazonaws.com
    - docker tag myapp-api:latest <repo-uri>:v1
    - docker push <repo-uri>:v1
4. Create Task Definitions (ECS)
    - Console → ECS → Task Definitions → Create task with container image, CPU, memory, port mapping.
5. Create ECS Cluster (Fargate)
    - Create cluster (Fargate), then create Service using task definition.
6. Setup Load Balancer
    - Create Application Load Balancer, target group pointing to ECS service.
7. Service discovery & networking
    - Use VPC networking, configure security groups for inter-service comms.
8. Auto-scaling
    - Configure service auto-scaling based on CPU or request count.
9. CI/CD (optional but recommended)
    - Connect GitHub Actions or CodePipeline to build Docker image and push to ECR, then update ECS service.
10. Testing
    - Access service via ALB, check logs in CloudWatch container logs.

11. Cost & cleanup
    - Use AWS Fargate with minimal CPU/memory for demos; remember to remove services to avoid charges.

Steps (EKS — if you want Kubernetes)

- Provision EKS cluster (EKS console or eksctl).
- Configure kubectl, create deployments/services, use Ingress with ALB Ingress Controller.
- Use Helm charts for easier deployments.

Extensions / Interview talking points

- Discuss container immutability, differences between ECS and EKS.
- Explain sidecar patterns (logging, monitoring), service mesh (linkerd/istio) basics.
- Show rolling updates & blue/green deployments.

✅ Common tips across all projects (beginners)

- Use Free Tier: Try to use free-tier resources and delete after use.
- Use IaC later: After console success, rewrite with Terraform / CloudFormation / CDK — that's gold for interviews.
- Logging & Observability: Add CloudWatch logs & simple dashboards — recruiters love it.
- Security basics: Use IAM roles (never hardcode keys), secure DB credentials in Secrets Manager.
- Document your work: Add README for each project with architecture diagram, steps, challenges faced, and learning — put on GitHub.

- Show cost-awareness: Explain how you optimized costs or used on-demand/smaller sizes for dev.

✅

Summary

By completing these five projects, you'll cover 90% of what's asked in AWS interviews — while building a real, deployable portfolio.

Each project isn't just a demo — it's a conversation starter that shows you understand real-world cloud challenges like scalability, reliability, and cost-efficiency.

Your next step:

👉 Rebuild each project using Infrastructure as Code

👉 Add monitoring, alerts, and dashboards

👉 Share your portfolio link on LinkedIn

Your AWS journey starts now.

Keep learning. Keep building. Keep growing.

— @abhishek_ctrl_alt_delete