I have AWS SDK professional training for 20 years laterals

Since your audience has **20+ years of experience**, the training must be **architecture-focused, enterprise-level, and practical**, not beginner-level SDK syntax. Senior professionals care about **design patterns, scalability, automation, performance, and real-world enterprise integration**.

Here is a **complete AWS SDK Professional Training Plan (Senior / 20+ Years Experience)**:

AWS SDK Professional Training Plan (For Senior Professionals)

Training Objective

Enable experienced professionals to:

- Architect enterprise-grade cloud applications using AWS SDK
- Automate infrastructure and workflows programmatically
- Build scalable, secure, production-ready systems
- Integrate AWS services into enterprise applications

Module 1: AWS SDK Architecture Deep Dive

Topics

- AWS SDK internal architecture
- SDK vs CLI vs Console vs IaC
- When to use SDK vs Terraform vs CloudFormation
- SDK lifecycle and request flow

Architecture Flow

java Copy code

```
Application
   ↓
AWS SDK
   ↓
AWS API Gateway
   ↓
AWS Service (S3, EC2, DynamoDB)
```

Demo (Python boto3)

python Copy code

```python
import boto3

s3 = boto3.client('s3')
response = s3.list_buckets()

for bucket in response['Buckets']:
    print(bucket['Name'])
```

Enterprise Use Case

ERP system listing storage buckets dynamically.

Module 2: Enterprise Authentication and Security

Topics

- IAM Roles vs IAM Users
- AssumeRole pattern
- Temporary credentials (STS)
- Cross-account access

Demo: Assume Role

python Copy code

```python
import boto3
```

```
sts = boto3.client('sts')

response = sts.assume_role(
    RoleArn="arn:aws:iam::123456789012:role/AdminRole",
    RoleSessionName="Session1"
)

print(response['Credentials'])
```

Enterprise Use Case

Finance system accessing secure AWS account.

Module 3: SDK with Compute Services

Topics

- Launch EC2 instances via SDK
- Auto Scaling automation
- Lambda invocation via SDK

Demo: Launch EC2

python Copy code

```
ec2 = boto3.resource('ec2')

instance = ec2.create_instances(
    ImageId='ami-12345',
    MinCount=1,
    MaxCount=1,
    InstanceType='t2.micro'
)

print(instance[0].id)
```

Enterprise Use Case

Auto-provision servers during peak load.

Module 4: SDK with Storage Services

Topics

- Upload/download files programmatically
- Backup automation
- Multi-region replication

Demo: Upload file

python Copy code

```python
s3 = boto3.client('s3')

s3.upload_file(
    'report.pdf',
    'company-bucket',
    'reports/report.pdf'
)
```

Enterprise Use Case

Automated backup system.

Module 5: SDK with Database Services

Topics

- DynamoDB CRUD via SDK
- RDS management
- Query optimization

Demo: Insert into DynamoDB

python Copy code

```python
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Employees')

table.put_item(
    Item={
```

```
        'ID': '101',
        'Name': 'Robert',
        'Department': 'Finance'
    }
)
```

Module 6: Event-Driven Enterprise Automation

Topics

- SDK with Lambda
- SDK with EventBridge
- Automation workflows

Architecture

mathematica Copy code

```
Application
   ↓
AWS SDK
   ↓
Lambda
   ↓
DynamoDB / S3 / SNS
```

Module 7: Enterprise Integration Patterns

Topics

- SDK in Microservices
- SDK in enterprise backend systems
- SDK with message queues (SQS)

Demo: Send message to queue

python Copy code

```
sqs = boto3.client('sqs')

sqs.send_message(
    QueueUrl='queue-url',
    MessageBody='Order created'
)
```

Module 8: Error Handling and Resilience

Topics

- Retry strategies
- Exponential backoff
- Fault tolerance

Demo

python Copy code

```python
import botocore

try:
    s3.list_buckets()
except botocore.exceptions.ClientError as e:
    print("Error:", e)
```

Module 9: Performance Optimization

Topics

- Connection pooling
- Parallel processing
- Async SDK usage

Module 10: Multi-Account Enterprise Architecture

Topics

- Cross-account automation
- Enterprise governance
- Role chaining

Module 11: SDK in DevOps and Automation

Topics

- CI/CD integration
- Automated deployments
- Infrastructure provisioning

Module 12: Real-World Enterprise Case Study

Case Study: Automated Enterprise Backup System

Flow:

```mathematica
Application
  ↓
SDK
  ↓
Lambda
  ↓
S3 Backup
  ↓
Notification via SNS
```

Advanced Enterprise Demo (End-to-End)

Automate:

- Launch EC2

- Upload logs to S3
- Store metadata in DynamoDB

Enterprise-Level Capstone Project

Project: Enterprise Infrastructure Automation System

Features:

- Auto-create EC2 servers
- Store metadata in DynamoDB
- Upload logs to S3
- Send notifications via SNS

Recommended SDK Languages for Enterprise

Priority order:

1. Python (boto3)
2. Java SDK
3. Node.js SDK
4. .NET SDK

Training Duration Plan (Recommended)

Module	Duration
Architecture	2 hours
Security	2 hours
Compute	3 hours
Storage	2 hours
Database	2 hours

Module	Duration
Event-Driven	2 hours
Integration	2 hours
Resilience	1.5 hours
Performance	1.5 hours
DevOps	2 hours
Capstone	4 hours

Total: 24 Hours

What Senior Professionals Expect (Important)

Focus on:

- Architecture diagrams
- Real enterprise use cases
- Automation patterns
- Performance and scaling
- Security best practices
- Failure scenarios

NOT basic syntax.

If you want, I can also create a **5-day AWS SDK enterprise training plan with demos, architecture diagrams, and hands-on labs specifically tailored for 20-year experienced professionals.**

I have DAY 1 -> TOC below -> I wnated professional Demos and Lab steps -> also examples, scenarios: Day 1 – Introduction to AWS Cloud & Developer Role

- AWS Overview & Free Tier Setup
- IAM Basics (Users, Groups, Policies)
- Developer Tools: AWS Console, CLI, SDKs

Hands-on: Free Tier setup + IAM user creation + AWS CLI configuration

Below is a **Day 1 Professional Training Plan with Enterprise-Level Demos, Labs, Scenarios, and Architecture Thinking**, specifically designed for **20+ years experienced professionals**.

Day 1 – Introduction to AWS Cloud & Developer Role (Professional Level)

Learning Outcome (Senior-Level)

By end of Day 1, participants will:

- Understand AWS architecture from an enterprise developer perspective
- Configure secure developer access using IAM best practices
- Use AWS CLI and SDK professionally (not console-only usage)
- Implement enterprise-grade authentication model

Part 1: AWS Overview & Free Tier Setup (Professional Perspective)

Concept Focus (Not basic)

Explain AWS in terms of enterprise architecture layers:

java Copy code

```
Enterprise Application
      ↓
AWS SDK / CLI
      ↓
AWS Control Plane (IAM, EC2, S3, Lambda)
      ↓
AWS Infrastructure Layer (Regions, AZs, Edge)
```

Professional Demo 1: Enterprise Application Hosting Architecture

Scenario

Your company is migrating an on-prem application to AWS.

Architecture:

pgsql Copy code

```
Developer Laptop
    ↓
AWS CLI / SDK
    ↓
EC2 (Application Server)
    ↓
S3 (Storage)
    ↓
CloudWatch (Monitoring)
```

Explain:

Developers interact using:

- Console → Manual
- CLI → Automation
- SDK → Application integration

Enterprise uses mostly CLI and SDK.

Lab 1: AWS Free Tier Setup (Professional Way)

Step 1: Create AWS Account

Go to:

https://aws.amazon.com/free ↗

Configure:

- Root user email
- Enable MFA (mandatory enterprise practice)

Step 2: Setup Billing Alert (Critical Enterprise Step)

Go to:

Billing → Budgets → Create Budget

Set:

makefile Copy code

```
Budget: $5
Alert: Email notification
```

Enterprise reason:
Prevent unexpected cloud costs.

Part 2: IAM Basics (Enterprise Security Model)

Explain IAM from enterprise architecture perspective:

Never use Root user.

Use model:

pgsql Copy code

```
Root User (Locked)
   ↓
Admin User
   ↓
Developer Users
   ↓
Roles (Recommended)
```

Professional Demo 2: Enterprise IAM Model

Scenario

Company has:

- 100 developers
- Each needs controlled access

Use IAM Users + Groups.

Architecture:

pgsql Copy code

```
IAM Group: Developers
   ↓
IAM Policy: EC2 + S3 Access
   ↓
IAM Users assigned to group
```

Lab 2: Create Enterprise IAM Structure

Step 1: Create IAM Group

Go to:

IAM → User Groups → Create group

Name:

nginx Copy code

```
Developers
```

Attach Policy:

nginx Copy code

```
AmazonEC2FullAccess
AmazonS3FullAccess
```

Step 2: Create IAM User

IAM → Users → Create user

Name:

Copy code

```
dev-user1
```

Select:

pgsql Copy code

```
Programmatic Access
AWS Console Access
```

Add to group:

nginx Copy code

```
Developers
```

Download credentials.

Enterprise Note:
Never share credentials via email. Use secure vault.

Professional Demo 3: Access Validation Test

Login using IAM user.

Verify:

Can access EC2 and S3

Cannot access Billing (important security validation)

Part 3: Developer Tools: Console vs CLI vs SDK

Explain enterprise usage comparison:

Tool	Enterprise Usage
Console	Manual tasks, debugging
CLI	Automation, DevOps
SDK	Application integration

Professional Demo 4: AWS CLI Installation and Setup

Step 1: Install AWS CLI

Windows:

Download:

https://aws.amazon.com/cli/ ↗

Verify:

```css
aws --version
```

Copy code

Expected:

Copy code

```
aws-cli/2.x.x
```

Step 2: Configure CLI

Run:

nginx Copy code

```
aws configure
```

Enter:

vbnet Copy code

```
Access Key
Secret Key
Region: ap-south-1
Output: json
```

Enterprise Explanation:

CLI stores credentials securely locally.

Lab 3: Professional CLI Hands-on

Test 1: List S3 Buckets

bash Copy code

```
aws s3 ls
```

Expected output:

php Copy code

```
(empty or bucket list)
```

Test 2: Create Bucket

pgsql Copy code

```
aws s3 mb s3://company-dev-bucket-unique-name
```

Verify:

bash Copy code

```
aws s3 ls
```

Test 3: Upload File

Create file:

Copy code

```
test.txt
```

Upload:

bash Copy code

```
aws s3 cp test.txt s3://company-dev-bucket-unique-name
```

Enterprise Use Case:

Application logs uploaded automatically.

Professional Demo 5: EC2 Creation via CLI

Run:

sql Copy code

```
aws ec2 describe-instances
```

Explain:

Developers automate infrastructure.

Part 4: SDK Introduction (Professional Developer Perspective)

Explain:

SDK used inside application code.

Architecture:

markdown Copy code

```
Java/Python Application
     ↓
AWS SDK
     ↓
AWS Services
```

Professional Demo 6: SDK Example (Python)

Install:

nginx Copy code

```
pip install boto3
```

Code:

python Copy code

```
import boto3

s3 = boto3.client('s3')

response = s3.list_buckets()
```

```
for bucket in response['Buckets']:
    print(bucket['Name'])
```

Enterprise Scenario:

Application dynamically reads storage.

Professional Scenario 1: Enterprise Developer Workflow

Developer builds application that:

- Uploads reports to S3
- Launches EC2 automatically
- Stores data in database

Workflow:

mathematica Copy code

```
Application
   ↓
AWS SDK
   ↓
AWS Infrastructure
```

Professional Scenario 2: Secure Developer Access Model

Enterprise Security Architecture:

pgsql Copy code

```
Developer
   ↓
IAM User
   ↓
IAM Role
```

```
    ↓
AWS Resources
```

Not direct root access.

Lab 4: Real Enterprise Lab (End-to-End)

Objective:

Simulate enterprise developer setup.

Step 1

Create IAM User

Step 2

Configure CLI

Step 3

Create S3 bucket via CLI

Step 4

Upload file via CLI

Step 5

Access via SDK code

Expected Outcome:

Fully functional enterprise developer environment.

Professional Troubleshooting Demo

Common errors and fixes:

Error:

```nginx
AccessDenied
```

Cause:

IAM policy missing.

Fix:

Attach correct policy.

Error:

```nginx
InvalidAccessKeyId
```

Fix:

Reconfigure CLI.

Enterprise Best Practices (Important for Senior Professionals)

Always use:

- IAM Users or Roles (Never root)
- MFA enabled

- Least privilege access
- CLI or SDK for automation
- Billing alerts enabled

End of Day 1 Architecture Summary

java Copy code

```
Developer Laptop
      ↓
IAM User
      ↓
AWS CLI / SDK
      ↓
AWS Services (S3, EC2)
      ↓
Enterprise Applications
```

Day 1 Professional Capstone Lab

Participants will:

- Create IAM User
- Configure CLI
- Create S3 bucket
- Upload file
- Access using SDK

If you want, I can also prepare **Day 1 PowerPoint slides, lab guide PDF, and trainer demo script specifically tailored for senior professionals.**

I have DAY 2 -> TOC below -> I wanated professional Demos and Lab steps -> also examples, scenarios: Day 2 – Terraform Infrastructure Automation and ECS Setup ()

- Infrastructure as Code Concepts
- Manual console setup vs declarative provisioning.
- Version-controlled, repeatable deployments.
- Terraform Workflow Overview
- init → plan → apply → destroy cycle.
- What Terraform means by “desired state.”
- Terraform Files & Structure
- Terraform Variables (Theory)
- Purpose of variables (reusability & customization).
- Types: string, number, bool, map, list.
- Variable precedence (defaults > tfvars > CLI > env).
- Create the AWS Resource using KMS using Terraform Code

Below is a **Day 2 – Professional-Level Training Plan** designed for senior/lateral professionals with strong architecture thinking.

This is not basic Terraform syntax — this is **enterprise IaC mindset + production scenarios + hands-on labs**.

Day 2 – Terraform Infrastructure Automation & ECS Setup

Learning Outcome (Senior-Level)

By end of Day 2, participants will:

- Understand Infrastructure as Code from enterprise architecture perspective
- Compare manual vs declarative provisioning (governance impact)
- Master Terraform workflow (init → plan → apply → destroy)
- Understand desired state model deeply
- Implement reusable Terraform structure
- Use variables professionally

- Create AWS KMS key via Terraform
- Provision ECS cluster using Terraform

PART 1 – Infrastructure as Code (Enterprise Thinking)

Professional Explanation

Infrastructure as Code (IaC) means:

> Infrastructure defined as version-controlled code.

Traditional model:

```arduino
Engineer → Console → Click → Resource created
```

Enterprise Problem:

- No tracking
- No reproducibility
- Human error
- No audit

Terraform model:

```css
Git Repo → Terraform Code → Plan → Apply → Infrastructure
```

Enterprise Advantage:

- Version control
- Peer review
- Repeatable environments
- Disaster recovery ready

Professional Demo 1: Manual Console vs Terraform

Scenario

Company needs 3 environments:

- Dev
- QA
- Prod

Manual Console Issues:

- Different configurations
- Missed tags
- Wrong security group

Terraform Solution:

Single codebase with variable-based environments.

PART 2 – Terraform Workflow Deep Dive

Terraform Lifecycle

csharp Copy code

```
terraform init
terraform plan
terraform apply
terraform destroy
```

Professional Demo 2 – Basic Terraform Workflow

Step 1: Install Terraform

Verify:

nginx Copy code

```
terraform -version
```

Step 2: Create Project Structure

```
aws-terraform-project/
 ├── main.tf
 ├── variables.tf
 ├── terraform.tfvars
 ├── outputs.tf
```

Lab 1 – First Terraform Resource (S3 Example)

main.tf

```hcl
provider "aws" {
  region = "ap-south-1"
}

resource "aws_s3_bucket" "company_bucket" {
  bucket = "company-dev-bucket-unique-12345"

  tags = {
    Environment = "Dev"
    Owner       = "PlatformTeam"
  }
}
```

Step 3: Run Workflow

Initialize

```
terraform init
```

See Execution Plan

```nginx
terraform plan
```

Explain:

Terraform compares:

- Current state
- Desired state (code)

Shows delta.

Apply

```nginx
terraform apply
```

Confirm: yes

Destroy

```nginx
terraform destroy
```

Enterprise Use Case:
Environment teardown after testing.

PART 3 – Desired State Model (Deep Understanding)

Terraform does NOT say:

"Create bucket"

It says:

"This is what infrastructure should look like."

Terraform then:

- Compares real world
- Makes changes to match desired state

Enterprise Example:

If bucket deleted manually,
Next `terraform apply` → Recreates bucket.

This ensures compliance.

PART 4 – Terraform File Structure (Professional Standard)

Enterprise Structure:

```
css
```

Copy code

```
project/
 ├── main.tf
 ├── variables.tf
 ├── outputs.tf
 ├── terraform.tfvars
 ├── backend.tf
 ├── modules/
       ├── ecs/
       ├── kms/
       ├── vpc/
```

Why modular?

- Reusability
- Team collaboration
- Governance

PART 5 – Terraform Variables (Professional Usage)

Why Variables?

Avoid hardcoding:

Bad:

```ini
region = "ap-south-1"
```

Good:

```ini
region = var.aws_region
```

---

variables.tf

```hcl
variable "aws_region" {
  type        = string
  description = "AWS Region"
}

variable "environment" {
  type    = string
  default = "dev"
}

variable "enable_deletion" {
  type    = bool
  default = false
}
```

---

terraform.tfvars

hcl Copy code

```
aws_region = "ap-south-1"
environment = "dev"
enable_deletion = true
```

Variable Precedence (Enterprise Exam Concept)

Lowest → Highest:

1. Default in variable block
2. terraform.tfvars
3. CLI `-var`
4. Environment variables

Enterprise Scenario:
CI/CD pipeline overrides region via CLI.

PART 6 – Create AWS KMS Key Using Terraform

Enterprise Scenario

Finance application needs encryption.

Requirement:

- Centralized KMS key
- Rotation enabled
- Tagged properly

Lab 2 – Create KMS Key

main.tf

hcl Copy code

```
provider "aws" {
  region = var.aws_region
}

resource "aws_kms_key" "company_kms" {
  description             = "Enterprise KMS Key"
  deletion_window_in_days = 7
  enable_key_rotation     = true

  tags = {
    Environment = var.environment
    Owner       = "SecurityTeam"
  }
}
```

---

outputs.tf

hcl Copy code

```
output "kms_key_id" {
  value = aws_kms_key.company_kms.id
}
```

Run:

csharp Copy code

```
terraform init
terraform plan
terraform apply
```

Verify in AWS Console → KMS → Keys

Enterprise Discussion:

Why enable rotation?

Compliance + audit requirements.

PART 7 – ECS Setup Using Terraform

Enterprise Scenario

Company wants containerized microservice platform.

Architecture:

pgsql Copy code

```
ECS Cluster
   ↓
Task Definition
   ↓
Service
   ↓
Load Balancer
```

Lab 3 – Create ECS Cluster

main.tf

hcl Copy code

```
resource "aws_ecs_cluster" "company_cluster" {
  name = "company-ecs-cluster"
}
```

Apply:

nginx Copy code

```
terraform apply
```

Verify in ECS Console.

Professional Scenario – Real Enterprise Deployment

Company runs:

- 10 microservices
- 3 environments
- Central encryption via KMS
- ECS-based architecture

Terraform Flow:

mathematica Copy code

```
GitHub
   ↓
Pull Request
   ↓
Terraform Plan
   ↓
Approval
   ↓
Terraform Apply
   ↓
Production Infrastructure
```

PART 8 – End-to-End Professional Lab

Objective:

Provision:

- KMS key
- ECS cluster
- S3 bucket (logs)

Lab Steps Summary

1. Create folder structure
2. Create variables.tf
3. Create main.tf

4. Add KMS resource
5. Add ECS cluster resource
6. Run:

```csharp
terraform init
terraform plan
terraform apply
```

Troubleshooting Demo (Important for Seniors)

Error:

```pgsql
Error: No valid credential sources found
```

Fix:

Configure AWS CLI first.

Error:

```nginx
BucketAlreadyExists
```

Fix:

Use unique name.

Enterprise Best Practices (Very Important)

- Use remote backend (S3 + DynamoDB)
- Use state locking
- Never commit secrets
- Use modules

- Tag everything
- Use separate workspaces per environment

Architecture Summary – Day 2

```
Terraform Code
     ↓
Terraform State
     ↓
AWS API
     ↓
Infrastructure (KMS, ECS, S3)
```

Day 2 Professional Capstone

Build:

- Secure container platform
- Encrypted infrastructure
- Version-controlled IaC

If you want, I can prepare **Day 3 – ECS Full Deployment with Task Definition, Service, ALB, IAM Roles, and CI/CD Integration** at enterprise level.

I have DAY 3 -> TOC below -> I wnated professional Demos and Lab steps -> also examples, scenarios: Day 3 – AWS CLI & SDK (Node.js or Python)

- CLI Scripting for EC2, RDS, IAM
- SDK Authentication and API Calls

Hands-on: Script EC2/RDS creation and test SDK query

- Why use the AWS CLI?

– Repeatable automation, faster testing, integration with shell scripts.

- CLI Configuration Concepts

– Access Key ID, Secret Key, Region, Output Format.

– Profiles (default vs custom).

- Command Structure Pattern– How to find help and documentation.
- CLI vs Console vs SDK

– When to choose each for automation.

- Security Considerations

– IAM least-privilege roles, avoiding hard-coded keys, using SSO and environment variables.

Below is a **Day 3 – Professional-Level Training Plan (AWS CLI & SDK)** designed specifically for **20+ years experienced professionals**, with **enterprise demos, automation labs, real-world scenarios, and production practices**.

This focuses on **automation, scripting, secure authentication, and programmatic infrastructure control**.

Day 3 – AWS CLI & SDK Automation (Enterprise-Level)

Learning Outcome

By end of Day 3, participants will:

- Automate infrastructure using AWS CLI scripts
- Understand CLI command structure deeply
- Use profiles and secure authentication methods
- Automate EC2, RDS, IAM using CLI
- Integrate SDK into enterprise applications
- Implement secure SDK authentication
- Understand CLI vs SDK vs Console usage in production

PART 1 – Why Enterprise Uses AWS CLI

Enterprise Scenario

Developer needs to create:

- 10 EC2 servers
- Multiple environments (Dev, QA, Prod)

Manual Console → Slow and error-prone

CLI → Automated and repeatable

Enterprise Automation Flow:

```
Shell Script
    ↓
AWS CLI
    ↓
AWS API
    ↓
Infrastructure Created
```

Professional Demo 1 – AWS CLI Configuration (Enterprise Way)

Step 1: Configure CLI

Run:

```
aws configure
```

Enter:

```
Access Key ID
Secret Access Key
```

```
Region: ap-south-1
Output: json
```

Step 2: Verify Access

bash Copy code

```
aws sts get-caller-identity
```

Expected Output:

json Copy code

```
{
  "UserId": "...",
  "Account": "...",
  "Arn": "arn:aws:iam::xxxx:user/dev-user"
}
```

Enterprise Use Case:
Verify identity before running automation scripts.

PART 2 – CLI Profiles (Enterprise Multi-Account Setup)

Enterprise Scenario:

Company has:

- Dev Account
- QA Account
- Prod Account

Create profile:

bash Copy code

```
aws configure --profile dev
```

Use profile:

bash Copy code

```
aws s3 ls --profile dev
```

Enterprise Advantage:

Supports multi-account automation.

PART 3 – CLI Command Structure Pattern

Standard Structure:

bash Copy code

```
aws <service> <operation> <parameters>
```

Examples:

List EC2 instances:

bash Copy code

```
aws ec2 describe-instances
```

List IAM users:

bash Copy code

```
aws iam list-users
```

Professional Demo 2 – Finding Help

Command help:

bash Copy code

```
aws ec2 help
```

Specific help:

bash Copy code

```
aws ec2 run-instances help
```

Enterprise Use Case:

Developers explore APIs quickly.

PART 4 – Automate EC2 Creation Using CLI

Enterprise Scenario

Auto-create server for application deployment.

Lab 1 – Create EC2 via CLI

Step 1: Create Security Group

bash Copy code

```
aws ec2 create-security-group \
--group-name dev-sg \
--description "Dev Security Group"
```

Step 2: Launch EC2 Instance

bash Copy code

```
aws ec2 run-instances \
--image-id ami-0f58b397bc5c1f2e8 \
--count 1 \
--instance-type t2.micro
```

Step 3: Verify Instance

bash Copy code

```
aws ec2 describe-instances
```

Enterprise Automation Scenario:

DevOps pipeline auto-creates servers.

PART 5 – Automate IAM Using CLI

Lab 2 – Create IAM User via CLI

bash Copy code

```
aws iam create-user --user-name automation-user
```

Attach policy:

bash Copy code

```
aws iam attach-user-policy \
--user-name automation-user \
--policy-arn arn:aws:iam::aws:policy/AmazonEC2FullAccess
```

Enterprise Scenario:
Automated onboarding system.

PART 6 – Automate RDS Creation Using CLI

Enterprise Scenario

Application requires database automatically during deployment.

Lab 3 – Create RDS Instance

bash

Copy code

```
aws rds create-db-instance \
--db-instance-identifier dev-db \
--db-instance-class db.t3.micro \
--engine mysql \
--master-username admin \
--master-user-password Password123 \
--allocated-storage 20
```

Enterprise Use Case:

Automated database provisioning.

PART 7 – CLI Automation Script (Professional Demo)

Create file:

deploy.sh

bash Copy code

```
#!/bin/bash

echo "Creating EC2 instance..."
aws ec2 run-instances \
--image-id ami-0f58b397bc5c1f2e8 \
--count 1 \
--instance-type t2.micro

echo "Listing instances..."
aws ec2 describe-instances
```

Run:

bash Copy code

```
chmod +x deploy.sh
./deploy.sh
```

Enterprise Use Case:

CI/CD automation.

PART 8 – SDK Authentication and API Calls

SDK is used inside applications.

Enterprise Architecture:

```markdown
Enterprise Application
      ↓
AWS SDK
      ↓
AWS API
```

Professional Demo 3 – SDK Setup (Python)

Install:

```bash
pip install boto3
```

Lab 4 – SDK Query EC2 Instances

Python code:

```python
import boto3

ec2 = boto3.client('ec2')

response = ec2.describe_instances()
```

```
for reservation in response['Reservations']:
    for instance in reservation['Instances']:
        print(instance['InstanceId'])
```

Enterprise Use Case:

Application monitors infrastructure.

Professional Demo 4 – Create EC2 Using SDK

python Copy code

```python
import boto3

ec2 = boto3.resource('ec2')

instance = ec2.create_instances(
    ImageId='ami-0f58b397bc5c1f2e8',
    MinCount=1,
    MaxCount=1,
    InstanceType='t2.micro'
)

print("Instance created:", instance[0].id)
```

PART 9 – CLI vs Console vs SDK (Enterprise Comparison)

Tool	Use Case
Console	Manual troubleshooting
CLI	Automation scripts
SDK	Application integration

Enterprise Flow:

arduino

Copy code

```
Application → SDK → AWS
DevOps → CLI → AWS
Admin → Console → AWS
```

PART 10 – Security Best Practices (Critical Enterprise Topic)

Never hardcode keys like:

python Copy code

```python
aws_access_key_id="ABC"
```

Use:

CLI configured credentials

OR environment variables:

bash Copy code

```bash
export AWS_ACCESS_KEY_ID=xxx
export AWS_SECRET_ACCESS_KEY=xxx
```

OR IAM roles (best practice)

Professional Demo 5 – Using Environment Variables

Linux:

bash Copy code

```bash
export AWS_ACCESS_KEY_ID=your_key
export AWS_SECRET_ACCESS_KEY=your_secret
```

Verify:

bash Copy code

```
aws sts get-caller-identity
```

Enterprise Scenario – Production Application

Application running on EC2 uses IAM role:

pgsql Copy code

```
EC2 Instance
   ↓
IAM Role
   ↓
AWS SDK
   ↓
S3 Access
```

No keys stored.

Secure architecture.

End-to-End Professional Lab

Participants will:

Step 1:
Configure CLI

Step 2:
Create EC2 using CLI

Step 3:
Create IAM user using CLI

Step 4:
Create RDS using CLI

Step 5:

Run SDK script to query EC2

Enterprise Architecture Summary

markdown Copy code

```
Shell Script
    ↓
AWS CLI
    ↓
AWS API
    ↓
Infrastructure

Application
    ↓
AWS SDK
    ↓
AWS API
```

Professional Troubleshooting Demo

Error:

css Copy code

```
Unable to locate credentials
```

Fix:

Run:

bash Copy code

```
aws configure
```

Error:

```nginx
AccessDenied
```

Fix:

Attach proper IAM policy.

Professional Capstone Lab – Enterprise Automation

Create script that:

- Creates EC2
- Creates IAM user
- Queries using SDK

Fully automated deployment.

If you want, I can also create **real enterprise-level CI/CD automation using CLI + SDK + Terraform integrated together**, which is what senior professionals expect in real companies.

I have DAY 3 -> TOC below -> I wnated professional Demos and Lab steps -> also examples, scenarios: Day 4 – Amazon S3 Deep Dive

Introduction to Object Storage

- Difference between block, file, and object storage.
- Why S3 is a backbone for data storage in AWS.

Key S3 Concepts

- Buckets, Objects, and Regions.
- Object structure (key, value, metadata, version ID).

Storage Classes & Use Cases

- Standard, Intelligent-Tiering, Glacier, Deep Archive.

- Real-time example: Storing app logs vs archived backups.

S3 Features Overview

- Versioning, Lifecycle Management, Encryption (SSE-S3, SSE-KMS).
- Access Control: Bucket Policy, ACL, IAM Roles.

Cost Optimization Basics

- Data transfer cost, storage class transitions.

Below is a **Day 4 – Amazon S3 Deep Dive (Professional-Level Training)** with **enterprise demos, architecture scenarios, hands-on labs, automation examples, and real-world best practices**, designed specifically for **senior professionals (20+ years experience)**.

This focuses on **enterprise storage architecture, security, lifecycle automation, and cost optimization**.

Day 4 – Amazon S3 Deep Dive (Enterprise-Level)

Learning Outcomes

By end of Day 4, participants will:

- Understand storage architecture types (block, file, object)
- Design enterprise storage using S3
- Implement versioning, encryption, lifecycle policies
- Secure S3 using IAM roles and bucket policies
- Optimize storage costs using storage classes
- Automate S3 using CLI and SDK

PART 1 – Storage Types Architecture

Enterprise Storage Comparison

Storage Type	Example	AWS Service	Use Case
Block Storage	Hard disk	EBS	Databases

Storage Type	Example	AWS Service	Use Case
File Storage	Network drive	EFS	Shared folders
Object Storage	Cloud storage	S3	Logs, backups, media

Professional Architecture Example

Enterprise Application Storage Model:

mathematica Copy code

```
Application Server (EC2)
   ↓
EBS → Database storage

Application
   ↓
EFS → Shared files

Application
   ↓
S3 → Logs, backups, media files
```

Professional Demo 1 – Why S3 is Backbone

Enterprise systems use S3 for:

- Application logs
- Backups
- Data lakes
- Machine learning datasets
- Static websites

Example Architecture:

mathematica Copy code

```
Web Application
   ↓
Uploads files
   ↓
Amazon S3
   ↓
Analytics / Backup / Archive
```

PART 2 – S3 Key Concepts

Core Components

pgsql Copy code

```
Bucket → Container
Object → File
Key → File path/name
Metadata → File info
Version ID → File version
```

Example Object Structure:

vbnet Copy code

```
Bucket: company-data

Object Key:
logs/2026/app.log

Metadata:
Created date
Owner
Version ID
```

Lab 1 – Create S3 Bucket (Enterprise Standard)

Using CLI

bash Copy code

```
aws s3 mb s3://company-prod-storage-unique123
```

Verify:

bash Copy code

```
aws s3 ls
```

Upload Object

bash Copy code

```
aws s3 cp test.txt s3://company-prod-storage-unique123/logs/test.txt
```

Verify:

bash Copy code

```
aws s3 ls s3://company-prod-storage-unique123/logs/
```

Enterprise Scenario

Application uploads logs automatically.

PART 3 – Storage Classes and Enterprise Use Cases

Storage Class Comparison

Storage Class	Use Case
Standard	Active application data
Intelligent-Tiering	Unknown access patterns

Storage Class	Use Case
Glacier	Backup storage
Deep Archive	Long-term archive

Enterprise Example

Application logs lifecycle:

sql Copy code

```
Day 0-30 → Standard
Day 30-90 → Intelligent-Tiering
Day 90-365 → Glacier
After 365 → Deep Archive
```

Cost reduction up to 90%.

Professional Demo 2 – Upload with Storage Class

bash Copy code

```
aws s3 cp test.txt s3://company-prod-storage-unique123 \
--storage-class STANDARD_IA
```

PART 4 – Versioning (Critical Enterprise Feature)

Why Versioning?

Protects against:

- Accidental deletion
- Overwrites
- Ransomware

Lab 2 – Enable Versioning

bash Copy code

```
aws s3api put-bucket-versioning \
--bucket company-prod-storage-unique123 \
--versioning-configuration Status=Enabled
```

Test Versioning

Upload file:

bash Copy code

```
aws s3 cp test.txt s3://company-prod-storage-unique123
```

Modify file and upload again.

List versions:

bash Copy code

```
aws s3api list-object-versions \
--bucket company-prod-storage-unique123
```

Enterprise Benefit:
Recover any version.

PART 5 – Lifecycle Management (Cost Optimization)

Enterprise Scenario:

Move old files automatically to Glacier.

Lab 3 – Lifecycle Policy

Create file lifecycle.json

json Copy code

```
{
  "Rules": [
    {
      "ID": "ArchiveRule",
      "Prefix": "",
      "Status": "Enabled",
      "Transitions": [
        {
          "Days": 30,
          "StorageClass": "GLACIER"
        }
      ]
    }
  ]
}
```

Apply policy:

bash Copy code

```
aws s3api put-bucket-lifecycle-configuration \
--bucket company-prod-storage-unique123 \
--lifecycle-configuration file://lifecycle.json
```

Enterprise Benefit:

Fully automated cost optimization.

PART 6 – Encryption (Enterprise Security)

Encryption Types:

Encryption	Description
SSE-S3	AWS managed
SSE-KMS	Enterprise-grade

Encryption	Description
Client-side	Application encrypts

Lab 4 – Enable Encryption (KMS)

bash Copy code

```
aws s3api put-bucket-encryption \
--bucket company-prod-storage-unique123 \
--server-side-encryption-configuration '{
"Rules":[{
"ApplyServerSideEncryptionByDefault":{
"SSEAlgorithm":"AES256"
}
}]
}'
```

Enterprise Scenario:

Finance data protection.

PART 7 – Access Control (Enterprise Security Model)

Access Methods:

- IAM Policies
- Bucket Policies
- IAM Roles

Enterprise Model:

pgsql Copy code

```
Application
   ↓
IAM Role
   ↓
S3 Access
```

Professional Demo 3 – Bucket Policy Example

Allow EC2 access:

policy.json

```json
{
 "Version":"2012-10-17",
 "Statement":[
   {
     "Effect":"Allow",
     "Principal":"*",
     "Action":"s3:GetObject",
     "Resource":"arn:aws:s3:::company-prod-storage-unique123/*"
   }
 ]
}
```

Apply:

```bash
aws s3api put-bucket-policy \
--bucket company-prod-storage-unique123 \
--policy file://policy.json
```

PART 8 – SDK Integration with S3

Lab 5 – Upload using Python SDK

```python
import boto3

s3 = boto3.client('s3')
```

```
s3.upload_file(
    'test.txt',
    'company-prod-storage-unique123',
    'logs/test.txt'
)
```

Enterprise Use Case:

Application uploads logs automatically.

PART 9 – Enterprise Architecture Scenario

Production Application Architecture:

pgsql Copy code

```
Application Server (EC2)
   ↓
IAM Role
   ↓
Amazon S3
   ↓
Lifecycle → Glacier
   ↓
Backup Storage
```

PART 10 – Cost Optimization Best Practices

Key Cost Factors:

- Storage class
- Data transfer
- Number of requests

Enterprise Strategy:

Use:

- Intelligent Tiering

- Lifecycle policies
- Compression

Professional Demo 4 – Analyze Storage

bash Copy code

```
aws s3 ls s3://company-prod-storage-unique123 --recursive --human-readable --summ
```

End-to-End Enterprise Lab

Participants will:

Step 1

Create S3 bucket

Step 2

Upload files

Step 3

Enable versioning

Step 4

Enable encryption

Step 5

Apply lifecycle policy

Step 6

Upload via SDK

Professional Troubleshooting

Error:

nginx

Copy code

```
AccessDenied
```

Fix:

Check IAM permissions.

Error:

nginx Copy code

```
BucketAlreadyExists
```

Fix:

Use globally unique name.

Enterprise Best Practices

Always:

- Enable versioning
- Enable encryption
- Use lifecycle policies
- Use IAM roles (not keys)
- Use storage classes wisely

Enterprise Architecture Summary

sql Copy code

```
Application
   ↓
IAM Role
   ↓
S3 Bucket
   ↓
Versioning + Encryption
```

```
    ↓
Lifecycle → Archive
```

Professional Capstone Lab (Enterprise Scenario)

Build enterprise log storage system:

- Secure storage
- Version protection
- Cost optimization
- Automated lifecycle

If you want, I can also prepare **Day 5 – ECS Container Deployment with S3 Integration, IAM Roles, Load Balancer, and CI/CD**, which completes real enterprise architecture.

I have DAY 5-> TOC below -> I wnated professional Demos and Lab steps -> also examples, scenarios: Day 5 – Lambda (Part 1) – Functions & Triggers

Lambda Introduction and Use Cases

- What is serverless computing?
- When to choose Lambda vs ECS/EKS.

Lambda Execution Model

- Function invocation flow, event sources, cold starts.
- Memory allocation and timeout configuration.

IAM Role & Permissions

- Execution role vs invocation permissions.
- How to grant Lambda access to Aurora RDS resources.

Networking Basics for Lambda (VPC Access)

- Public vs private Lambda configuration.
- Subnets, Security Groups, and NAT Gateway impact.

Environment Variables & Secrets

- Managing DB credentials via SSM Parameter Store or

Secrets Manager.
- Encrypting environment variables with KMS.

Below is a **Day 5 – AWS Lambda (Part 1) Professional Training Plan** designed for **senior professionals**, with **enterprise architecture thinking, demos, hands-on labs, production scenarios, security integration, and VPC + RDS connectivity**.

This focuses on **real-world enterprise serverless architecture**, not basic hello-world.

Day 5 – AWS Lambda (Part 1) – Functions & Triggers (Enterprise-Level)

Learning Outcomes

By end of Day 5, participants will:

- Understand serverless architecture deeply
- Know when to use Lambda vs ECS/EKS
- Create Lambda functions with proper IAM roles
- Integrate Lambda with S3, API, and RDS
- Configure Lambda inside VPC securely
- Manage secrets using Parameter Store / Secrets Manager
- Implement enterprise-grade security model

PART 1 – Serverless Computing (Enterprise Perspective)

What is Serverless?

Serverless means:

You manage code, AWS manages infrastructure.

Traditional model:

mathematica Copy code

```
Developer
   ↓
Provision EC2
   ↓
Install software
   ↓
Maintain server
```

Serverless model:

```
Developer
   ↓
Upload code
   ↓
AWS runs automatically
```

Enterprise Architecture Example

Traditional:

```
Application
   ↓
EC2 Server (always running)
```

Serverless:

```
Application Event
   ↓
Lambda Function
   ↓
Runs on demand
```

Enterprise Use Cases

Lambda used for:

- File processing
- API backend
- Event-driven automation
- Log processing
- Database triggers

Lambda vs ECS vs EKS (Enterprise Decision Matrix)

Use Case	Lambda	ECS	EKS
Short tasks	Yes	No	No
Long-running services	No	Yes	Yes
Microservices	Yes	Yes	Yes
Containers	No	Yes	Yes
Kubernetes required	No	No	Yes

Professional Demo 1 – Create Lambda Function

Scenario

Application uploads file → Lambda processes file automatically.

Lab 1 – Create Lambda Function

Step 1: Go to AWS Console → Lambda

Click:

Create Function

Choose:

```csharp
Author from scratch
```

Enter:

```vbnet
Function name: process-file-function
Runtime: Python 3.11
```

Choose:

Create new IAM role

Click Create.

PART 2 – Lambda Execution Model

Invocation Flow Architecture

```sql
Event Source (S3/API/EventBridge)
        ↓
Lambda Service
        ↓
Execution Environment
        ↓
Function Runs
        ↓
Returns response
```

Professional Demo 2 – Lambda Execution Example

Code:

python Copy code

```python
def lambda_handler(event, context):
    print("Event received:", event)

    return {
        'statusCode': 200,
        'body': 'Lambda executed successfully'
    }
```

Click:

Deploy

Click:

Test

Explain Cold Start (Enterprise Concept)

Cold start occurs when:

Lambda starts new execution environment.

Impacts latency slightly.

Enterprise mitigation:

- Provisioned concurrency
- Proper memory configuration

PART 3 – Memory and Timeout Configuration

Enterprise tuning improves performance.

Example settings:

makefile Copy code

```
Memory: 512 MB
Timeout: 30 seconds
```

More memory → More CPU → Faster execution

Lab 2 – Configure Memory and Timeout

Go to:

Configuration → General Configuration

Edit:

Memory: 512 MB
Timeout: 30 sec

Save.

PART 4 – IAM Role & Permissions

Lambda uses execution role to access resources.

Architecture:

pgsql Copy code

```
Lambda Function
   ↓
IAM Execution Role
   ↓
Access S3 / RDS / DynamoDB
```

Professional Demo 3 – Add S3 Access Permission

Go to:

Lambda → Configuration → Permissions

Click:

Execution Role → Add policy

Attach:

```nginx
AmazonS3FullAccess
```

PART 5 – Lambda Access to Aurora RDS (Enterprise Scenario)

Architecture:

```pgsql
Lambda Function
   ↓
IAM Role
   ↓
Aurora RDS
```

Requirements:

- IAM Role permission
- VPC access
- Security group access

PART 6 – Lambda Networking (VPC Access)

Enterprise Database is inside private VPC.

Lambda must connect securely.

Architecture:

vbnet Copy code

```
Lambda Function
   ↓
Private Subnet
   ↓
Security Group
   ↓
Aurora RDS
```

Lab 3 – Configure Lambda VPC Access

Go to:

Lambda → Configuration → VPC

Select:

- Your VPC
- Private subnet
- Security group

Save.

Enterprise Impact:

Lambda can now access database.

PART 7 – Environment Variables & Secrets

Never hardcode credentials.

Bad:

ini Copy code

```
password = "admin123"
```

Enterprise Solution:

Use:

- Secrets Manager
- Parameter Store

Lab 4 – Add Environment Variable

Go to:

Lambda → Configuration → Environment Variables

Add:

ini Copy code

```
DB_HOST = database-url
DB_USER = admin
```

Save.

PART 8 – Using Parameter Store (Enterprise Secure Method)

Store secret:

Go to:

SSM → Parameter Store → Create Parameter

Name:

pgsql Copy code

```
db-password
```

Type:

SecureString

Professional Demo 4 – Access Parameter Store via Lambda

Code:

python Copy code

```python
import boto3

ssm = boto3.client('ssm')

def lambda_handler(event, context):

    response = ssm.get_parameter(
        Name='db-password',
        WithDecryption=True
    )

    password = response['Parameter']['Value']

    print(password)
```

Enterprise Benefit:

Secrets stored securely.

PART 9 – Lambda Trigger Example (S3 Trigger)

Enterprise Scenario:

File upload triggers Lambda.

Architecture:

```
csharp
```

Copy code

```
User uploads file
     ↓
S3 Bucket
     ↓
Lambda Triggered
     ↓
Process file
```

Lab 5 – Create S3 Trigger

Go to:

Lambda → Configuration → Triggers

Add trigger:

Select:

S3

Choose bucket.

Save.

Test by uploading file.

Lambda executes automatically.

PART 10 – Encryption using KMS

Enterprise uses KMS to encrypt secrets.

Architecture:

```
nginx
```

Copy code

```
Secrets Manager
   ↓
KMS Encryption
   ↓
Lambda Access
```

Professional Demo 5 – Enable KMS Encryption

Go to:

Lambda → Configuration → Environment Variables

Enable encryption using KMS key.

Enterprise End-to-End Architecture Scenario

sql Copy code

```
User uploads file
     ↓
S3 Bucket
     ↓
Lambda Function
     ↓
Fetch secret from Parameter Store
     ↓
Connect to Aurora RDS
     ↓
Store processed data
```

End-to-End Professional Lab

Participants will:

Step 1

Create Lambda function

Step 2

Configure IAM role

Step 3

Configure environment variables

Step 4

Store secrets in Parameter Store

Step 5

Configure VPC access

Step 6

Add S3 trigger

Step 7

Test Lambda execution

Professional Troubleshooting

Error:

```nginx
AccessDeniedException
```

Copy code

Fix:

Add proper IAM permissions.

Error:

```nginx
Timeout error
```

Copy code

Fix:

Increase timeout or fix networking.

Error:

```pgsql
Cannot connect to database
```

Fix:

Check security group rules.

Enterprise Best Practices

Always:

- Use IAM roles
- Use Parameter Store or Secrets Manager
- Use private VPC for database access
- Use KMS encryption
- Use environment variables

Enterprise Architecture Summary

```vbnet
S3 Event
   ↓
Lambda Function
   ↓
IAM Role
   ↓
Secrets Manager
   ↓
Aurora RDS
```

Professional Capstone Lab

Build secure serverless file processing system.

If you want, I can prepare **Day 6 – Lambda Part 2 with API Gateway integration, real-time enterprise REST API, and production-grade architecture**, which is the most important enterprise use case.

I have DAY 6 toc below --> I wnated professional Demos and Lab steps -> also examples, scenarios: -- Day 6 – KMS (Key Management Service) & RDS Integration ()

Why Encryption Matters

- Data at rest vs data in transit.
- Compliance (GDPR, HIPAA, PCI DSS) use cases.
- Shared responsibility for data protection.

Introduction to AWS KMS

- What is AWS Key Management Service?
- Types of Keys:
  - AWS Managed Keys (e.g., aws/rds)
  - Customer Managed Keys (CMK)
  - AWS Owned Keys
- Key components: Key ID, Key Policy, Grants, Aliases, Rotation.

KMS Key Management Concepts

- Envelope encryption concept.
- Key hierarchy (CMK → DEK → Resource Encryption).
- Automatic key rotation.
- Logging with CloudTrail for audit visibility.

Access Control in KMS

- How IAM and Key Policies work together.
- Least privilege for KMS usage (separate encrypt/decrypt roles).

Below is a **Day 6 – AWS KMS & RDS Integration (Enterprise-Level Training)** designed for **senior professionals**, with **security architecture, encryption design, compliance scenarios, professional demos, hands-on labs, and real-world integration with RDS**.

This is critical for **enterprise security, compliance, and production-grade encryption architecture**.

Day 6 – AWS KMS & RDS Integration (Enterprise-Level)

Learning Outcomes

By end of Day 6, participants will:

- Understand enterprise encryption architecture
- Understand KMS key types and usage
- Implement encryption for RDS using KMS
- Implement secure key policies and IAM access
- Understand envelope encryption deeply
- Enable audit logging using CloudTrail
- Implement least privilege access model

PART 1 – Why Encryption Matters (Enterprise Security Foundation)

Enterprise Threat Model

Without encryption:

pgsql Copy code

```
Database → Data stored in plain text
```

If attacker gains access → Data exposed.

With encryption:

pgsql Copy code

```
Database
   ↓
Encrypted using KMS key
   ↓
Unreadable without key
```

Data at Rest vs Data in Transit

Type	Description	Example
Data at rest	Stored data encryption	RDS, S3
Data in transit	Network encryption	HTTPS, TLS

Enterprise Example:

pgsql Copy code

```
Application → HTTPS → RDS (encrypted at rest)
```

Compliance Requirements (Enterprise)

Encryption required for:

- GDPR → User data protection
- HIPAA → Healthcare data
- PCI DSS → Payment data

Enterprise Example:

Payment database MUST use encryption.

PART 2 – AWS Shared Responsibility Model (Encryption Perspective)

AWS responsible for:

- Infrastructure security

Customer responsible for:

- Data encryption
- Key management
- Access control

PART 3 – AWS KMS Introduction

What is AWS KMS?

AWS KMS is key management service.

Used to:

- Create encryption keys
- Encrypt resources
- Control access to encryption keys

Architecture:

```pgsql
Resource (RDS, S3, Lambda)
     ↓
Data Encryption Key (DEK)
     ↓
Encrypted using CMK (in KMS)
```

PART 4 – Types of KMS Keys

1. AWS Managed Keys

Example:

```bash
aws/rds
```

```
aws/s3
```

Fully managed by AWS.

Limited control.

---

2. Customer Managed Keys (CMK) – Enterprise Recommended

Full control:

- Rotation
- Permissions
- Policies

---

3. AWS Owned Keys

Used internally by AWS.

Not visible to customer.

Professional Demo 1 – Create Customer Managed Key

Lab 1 – Create CMK

Go to:

AWS Console → KMS → Create Key

Select:

```sql
Symmetric
```

Choose:

```mathematica
```

```
Encrypt and decrypt
```

Enter:

Alias:

vbnet Copy code

```
company-rds-key
```

Click Create.

PART 5 – Key Components

Component	Purpose
Key ID	Unique identifier
Alias	Human readable name
Key Policy	Defines access
Grants	Temporary access
Rotation	Automatic key change

PART 6 – Envelope Encryption (Critical Enterprise Concept)

Enterprise encryption uses 2-layer encryption.

Architecture:

pgsql Copy code

```
Data
 ↓
Encrypted using DEK (Data Encryption Key)
 ↓
```

```
DEK encrypted using CMK
 ↓
Stored securely
```

Why?

Performance + security.

PART 7 – Key Hierarchy Architecture

mathematica Copy code

```
Customer Managed Key (CMK)
        ↓
Data Encryption Key (DEK)
        ↓
Encrypt Resource
        ↓
RDS Database
```

PART 8 – Enable Key Rotation

Enterprise security best practice.

Lab 2 – Enable Rotation

Go to:

KMS → Select key → Key rotation

Enable:

mathematica Copy code

```
Automatic rotation
```

Rotation happens yearly.

PART 9 – Audit Logging using CloudTrail

Enterprise compliance requires audit.

CloudTrail logs:

- Who used key
- When used
- Which resource used

Architecture:

```vbnet
User
 ↓
KMS Key Usage
 ↓
CloudTrail logs event
```

Professional Demo 2 – View KMS Logs

Go to:

CloudTrail → Event History

Filter:

```css
Event Source: kms.amazonaws.com
```

View logs.

PART 10 – Access Control in KMS

Access controlled by:

- IAM policies
- Key policies

Both must allow access.

Architecture:

vbnet Copy code

```
IAM Role
   ↓
Key Policy
   ↓
KMS Key
```

Professional Demo 3 – IAM Policy for KMS

Example Policy:

json Copy code

```
{
 "Effect": "Allow",
 "Action": [
   "kms:Encrypt",
   "kms:Decrypt"
 ],
 "Resource": "*"
}
```

PART 11 – RDS Encryption using KMS (Enterprise Scenario)

Enterprise Architecture:

pgsql Copy code

```
Application
   ↓
```

```
RDS Database
   ↓
Encrypted using KMS CMK
```

Lab 3 – Create Encrypted RDS Database

Go to:

RDS → Create database

Select:

```makefile
Engine: MySQL
```

Enable:

```nginx
Encryption
```

Choose:

```vbnet
company-rds-key
```

Create database.

Verify Encryption

Go to:

RDS → Database → Configuration

Check:

```vbnet
```

Copy code

```
Encryption: Enabled
KMS key: company-rds-key
```

PART 12 – Lambda Access to Encrypted RDS

Enterprise architecture:

pgsql Copy code

```
Lambda Function
   ↓
IAM Role
   ↓
KMS Key access
   ↓
Encrypted RDS
```

Lab 4 – Add Lambda Access to KMS

Attach policy to Lambda role:

json Copy code

```
{
 "Effect": "Allow",
 "Action": [
   "kms:Decrypt"
 ],
 "Resource": "*"
}
```

PART 13 – CLI Demo for KMS

Create key using CLI:

bash Copy code

```
aws kms create-key \
--description "Enterprise encryption key"
```

List keys:

bash Copy code

```
aws kms list-keys
```

PART 14 – SDK Demo using Python

Encrypt data:

python Copy code

```
import boto3

kms = boto3.client('kms')

response = kms.encrypt(
    KeyId='alias/company-rds-key',
    Plaintext='Sensitive Data'.encode()
)

print(response['CiphertextBlob'])
```

Enterprise Scenario – Banking Application

Architecture:

pgsql Copy code

```
Banking Application
       ↓
Lambda
       ↓
Encrypted RDS
```

```
        ↓
KMS Key
        ↓
CloudTrail Audit
```

Compliance satisfied.

End-to-End Enterprise Lab

Participants will:

Step 1
Create CMK

Step 2
Enable key rotation

Step 3
Create encrypted RDS

Step 4
Grant IAM access

Step 5
Test encryption using SDK

Step 6
View audit logs

Professional Troubleshooting

Error:

scss Copy code

```
AccessDeniedException (KMS)
```

Fix:

Add IAM permission and key policy.

Error:

```pgsql
Cannot access encrypted database
```

Fix:

Grant role access to KMS.

Enterprise Best Practices

Always:

- Use Customer Managed Keys
- Enable rotation
- Enable CloudTrail logging
- Use least privilege access
- Separate encrypt and decrypt roles

Enterprise Architecture Summary

```mathematica
Application
   ↓
IAM Role
   ↓
KMS CMK
   ↓
Encrypt DEK
   ↓
Encrypt RDS Data
```

Professional Capstone Lab

Build secure encrypted database architecture using KMS and RDS.

If you want, I can prepare **Day 7 – Real Enterprise Project integrating S3 + Lambda + RDS + KMS + IAM + Terraform**, which simulates actual production system design.