AWS Serverless Project

E-Commerce Order Processing System

A Step-by-Step Console-Based Implementation Guide

Compiled By

Mukhtar Muhammad Lawan PhD

AWS Certified Solutions Architect

Course: Cloud Computing & Serverless Architecture

Duration: 15 hours

Level: Beginner to Intermediate

December 20, 2025

Contents

Project Introduction

Welcome to AWS Serverless

Project Overview

This project guides students through building a complete serverless e-commerce order processing system using AWS Management Console. No command-line interface (CLI) knowledge is required - all steps are performed through the visual web interface.

Learning Objectives

Upon completing this project, students will be able to:

- **Create and secure** an AWS Free Tier account
- **Implement IAM** (Identity and Access Management) best practices
- **Develop and deploy** AWS Lambda functions using Python
- **Design workflows** with AWS Step Functions visual designer
- **Store data** in Amazon S3 (Simple Storage Service)
- **Monitor applications** using Amazon CloudWatch
- **Integrate multiple** AWS services into a complete solution
- **Manage costs** and clean up resources properly

Project Architecture

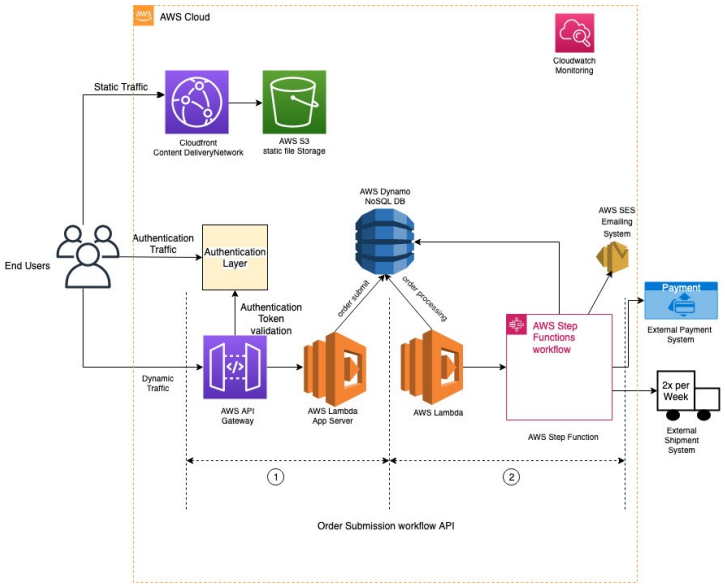

Figure 1: Serverless Order Processing Architecture

The system follows this workflow:

1. **Order Received**: Customer places an order
2. **Validation**: Lambda function validates order data
3. **Payment Processing**: Simulated payment authorization
4. **Decision Point**: Check payment success/failure
5. **Storage**: Save successful orders to S3
6. **Notification**: Send status updates (optional)
7. **Monitoring**: Track all steps in CloudWatch

Project Timeline

Hours	Phase	Topics Covered
Hours 1-3	Setup & Foundation	Account creation, IAM, S3
Hours 4-6	Lambda Functions	Create and test Lambda functions
Hours 7-9	Step Functions	Visual workflow design
Hours 10-12	Integration & Testing	End-to-end testing, monitoring
Hours 13-15	Advanced Features & Cleanup	Notifications, alarms, cleanup

Table 1: 15-hours Project Schedule

AWS Free Tier Information

Warning

Important Free Tier Limits:

- **Lambda**: 1 million requests per month
- **Step Functions**: 4,000 state transitions per month
- **S3**: 5GB standard storage, 20,000 GET requests
- **CloudWatch**: 10 custom metrics, 5GB log ingestion
- **SNS**: 1,000 email notifications per month

Note: These limits are generous enough for this project if resources are cleaned up properly.

Student Deliverables

Students must submit the following at project completion:

1. Screenshots of all AWS resources created
2. Working Lambda function code
3. Step Functions execution results
4. S3 bucket contents verification
5. CloudWatch dashboard screenshots
6. Cleanup verification
7. Project documentation report

1 Phase 1: AWS Account Setup & Security

Note

Time Estimate: 60-90 minutes
Completion Criteria: AWS account created, IAM user configured, billing alarms set

1.1 Step 1: Create AWS Free Tier Account

1.1.1 Account Creation Process

1. **Open AWS Homepage**
   - Go to https://aws.amazon.com
   - Click the **"Create an AWS Account"** button (orange)
2. **Enter Account Information**
   - Email address (use your institutional email)
   - Password (follow AWS requirements)
   - AWS account name (suggested: `student-serverless-project`)
3. **Contact Information**
   - Fill in personal/business information
   - Use your real information for verification
4. **Payment Information**
   - **CRITICAL:** Add a valid credit/debit card
   - You will NOT be charged if you stay within Free Tier
   - AWS uses this for identity verification
5. **Identity Verification**
   - AWS will call the phone number provided
   - Enter the PIN shown on screen when prompted
6. **Support Plan Selection**
   - Choose **"Basic Plan"** - This is FREE
   - Do NOT select Business or Enterprise plans

Warning

Security Reminder:

- Never use root account for daily tasks
- Enable Multi-Factor Authentication (MFA) if possible
- Keep credentials secure and never share them

1.1.2 Verification Checklist

Task	Status
Account creation email received	
Can log into AWS Management Console	
Root account email verified	
Billing information added	

Table 2: Account Setup Checklist

1.2 Step 2: Configure Billing Alerts

Tip

Always set up billing alerts before creating any resources. This is your safety net against unexpected charges.

1.2.1 Enable Billing Preferences

1. **Access Billing Dashboard**
   - Click your account name (top-right corner)
   - Select **"Billing Dashboard"**
2. **Configure Billing Preferences**
   - In left sidebar, click **"Billing Preferences"**
   - Check these boxes:
     - **Receive Free Tier Usage Alerts**
     - **Receive Billing Alerts**
     - **Receive PDF Invoice By Email**
   - Click **"Save preferences"**
3. **Take Screenshot**

- Capture the Billing Preferences page
- Save as `01-billing-preferences.png`

1.2.2 Create CloudWatch Billing Alarm

1. **Open CloudWatch Console**
   - Search "CloudWatch" in AWS Services search bar
   - Click to open CloudWatch
2. **Navigate to Alarms**
   - Left sidebar → **"Alarms"** → **"All Alarms"**
   - Click **"Create alarm"**
3. **Select Metric**
   - Click **"Select metric"**
   - Choose **"Billing"** → **"Total Estimated Charge"**
   - Select currency: **"USD"**
   - Click **"Select metric"**
4. **Configure Conditions**
   - Statistic: **"Maximum"**
   - Period: **"6 hours"**
   - Conditions: **"Greater/Equal than"**
   - Threshold: **"1"** (One US Dollar)
   - Click **"Next"**
5. **Configure Notifications**
   - Notification: **"In alarm"**
   - Create new SNS topic
   - Topic name: `Billing-Alarm-Notification`
   - Email endpoints: Enter your email address
   - Click **"Create topic"**
6. **Name and Create**
   - Alarm name: `Monthly-Billing-Alarm`
   - Description: `"Alert if charges exceed $1"`
   - Click **"Next"**
   - Review and click **"Create alarm"**

7. **Verify Email**
    - Check your email for subscription confirmation
    - Click confirmation link in email

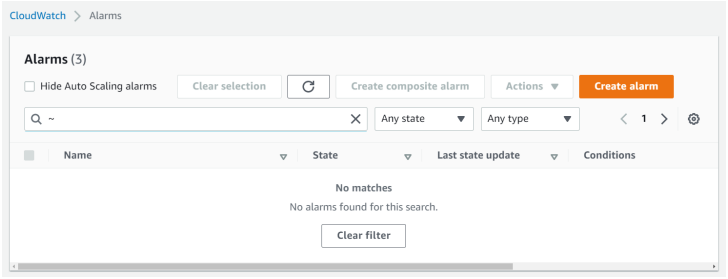

Figure 2: Billing Alarm Configuration

1.3 Step 3: Create IAM User

Warning

Security Best Practice: Never use root account credentials for development. Always create IAM users with appropriate permissions.

1.3.1 IAM User Creation Steps

1. **Open IAM Console**
    - Search "IAM" in AWS Services search
    - Click to open IAM Console
2. **Navigate to Users**
    - Left sidebar → **"Users"**
    - Click **"Create user"**
3. **Set User Details**
    - User name: `student-dev-user`
    - Check: **"Provide user access to the AWS Management Console"**
    - Console password: **"Custom password"**
    - Enter a strong password
    - Uncheck: **"User must create a new password..."**
    - Click **"Next"**
4. **Set Permissions**

- Select: **"Attach policies directly"**
- Search and select these policies:
  - `AWSLambda_FullAccess`
  - `AWSStepFunctionsFullAccess`
  - `AmazonS3FullAccess`
  - `CloudWatchLogsFullAccess`
- Click **"Next"**

5. **Review and Create**
   - Review all settings
   - Click **"Create user"**
6. **Download Credentials**
   - **CRITICAL:** Click **"Download .csv file"**
   - Save securely (this is your only chance to download)
   - File contains:
     - Console login URL
     - Username
     - Password
     - Access key ID (for programmatic access)
     - Secret access key (for programmatic access)
7. **Take Screenshot**
   - Capture the user creation success page
   - Save as `02-iam-user-created.png`

1.3.2 Switch to IAM User

1. **Log Out of Root Account**
   - Click account name (top-right)
   - Select **"Sign out"**
2. **Log In as IAM User**
   - Use the custom sign-in URL from CSV file
   - Or go to: https://[your-account-id].signin.aws.amazon.com/console
   - Enter username and password from CSV
3. **Verify Access**
   - You should see the AWS Management Console
   - Your account name should show in top-right

1.3.3 IAM User Configuration Checklist

Configuration Item	Status
IAM user created successfully	
Policies attached (4 required)	
CSV credentials downloaded	
Successfully logged in as IAM user	
Billing alarm created and confirmed	

Table 3: IAM Setup Verification

1.4 Step 4: Region Selection

Important: Region Consistency

All AWS services in this project MUST be created in the same region. I recommend using **US East (N. Virginia) us-east-1** as it has the most services and best Free Tier availability.

1.4.1 Set Default Region

1. **Check Current Region**
   - Look at top-right corner of AWS Console
   - It should show your current region
2. **Change Region if Needed**
   - Click the region selector
   - Select **"US East (N. Virginia) us-east-1"**
   - The console will refresh
3. **Take Screenshot**
   - Capture the region selector showing us-east-1
   - Save as `03-region-selected.png`

1.5 Phase 1 Completion Checklist

Deliverable	Submitted
AWS account created and verified	
Billing preferences configured	
CloudWatch billing alarm created	
Email confirmed for billing alerts	
IAM user created with proper permissions	
Credentials CSV file downloaded	
Successfully logged in as IAM user	
Region set to us-east-1	
Screenshots collected (3 required)	

Table 4: Phase 1 Completion Requirements

1.6 Troubleshooting Common Issues

1. **Cannot verify phone number**
   - Ensure you're entering the PIN correctly
   - Use a mobile phone for better reception
   - Try the SMS option if available
2. **Credit card declined**
   - Contact your bank to allow international transactions
   - Some prepaid cards may not work
   - Try a different card if possible
3. **Cannot download IAM credentials**
   - If you missed downloading, delete user and recreate
   - Check browser download settings
   - Try a different browser
4. **Cannot log in as IAM user**
   - Use the exact URL from CSV file
   - Check username and password are correct
   - Ensure you're not using root account credentials

1.7 Phase 1 Summary

You have now successfully:

- Created a secure AWS Free Tier account
- Configured billing alerts to prevent unexpected charges
- Created an IAM user with appropriate permissions
- Set up proper security practices
- Selected the correct AWS region

Tip

Best Practice: Store your IAM credentials in a secure password manager. Never commit them to version control or share them publicly.

2 Phase 2: Amazon S3 Bucket Creation

Note

Time Estimate: 20-30 minutes
Completion Criteria: S3 bucket created with proper folder structure and permissions

2.1 Introduction to Amazon S3

Amazon Simple Storage Service (S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. In our project, S3 will store:

- Processed order data (JSON files)
- Logs and audit trails
- Backup of successful transactions

S3 Free Tier Limits

- 5 GB of standard storage
- 20,000 GET requests
- 2,000 PUT, COPY, POST, or LIST requests
- 15 GB of data transfer out each month

2.2 Step 1: Create S3 Bucket

2.2.1 Navigation to S3 Console

1. **Open AWS Management Console**
   - Ensure you're logged in as `student-dev-user`
   - Region should be **us-east-1**
2. **Find S3 Service**
   - In the search bar (top), type "S3"
   - Click on **"S3"** in the search results
   - Alternatively, go to: S3 Console

2.2.2 Bucket Creation Process

1. **Start Creation**
   - Click the orange **"Create bucket"** button
2. **General Configuration**
   - **Bucket name:** `order-data-[YOUR-STUDENT-ID]`
     - Replace `[YOUR-STUDENT-ID]` with your actual ID
     - Example: `order-data-s1234567`
     - **Note:** Bucket names must be globally unique
   - **AWS Region:** `US East (N. Virginia) us-east-1`
   - Leave other settings as default
   - Click **"Next"**
3. **Configure Options**
   - **Versioning:** Disable
   - **Server access logging:** Disable
   - **Tags:** Add project tag
     - Click **"Add tag"**
     - Key: `Project`
     - Value: `Serverless-Order-Processing`
   - **Object-level logging:** Disable
   - Click **"Next"**
4. **Set Permissions**
   - **Block Public Access:** Keep ALL boxes checked
     - This ensures your bucket is not publicly accessible
     - Critical for security
   - **Bucket owner preferred:** Leave unchecked
   - **Bucket policy:** Leave empty
   - Click **"Next"**
5. **Review and Create**
   - Review all settings
   - Ensure bucket name is correct
   - Click **"Create bucket"**
6. **Confirmation**
   - You should see "Successfully created bucket"

- The bucket appears in your bucket list

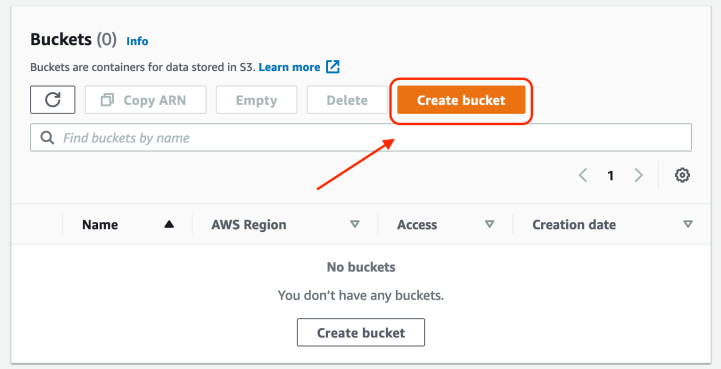

Figure 3: S3 Bucket Creation Screen

2.2.3 Bucket Naming Convention

Requirement	Example
Must be globally unique	`order-data-john-smith-2024`
3-63 characters long	`order-data-s1234567`
Lowercase letters, numbers, hyphens	`order-data-student-01`
Cannot start or end with hyphen	`orders-data-main`
Cannot be an IP address	`order-data-project` (not 192.168.1.1)

Table 5: S3 Bucket Naming Rules

2.3 Step 2: Create Folder Structure

Tip

Organizing your S3 bucket with folders makes it easier to manage different types of data and implement lifecycle policies.

2.3.1 Creating Folders

1. **Open Your Bucket**
   - Click on your bucket name in the S3 console
2. **Create Main Folders**
   - Click "**Create folder**"
   - Folder name: `orders/`

- Click "**Create folder**"
- Click "**Create folder**" again
- Folder name: `logs/`
- Click "**Create folder**"
- Click "**Create folder**" again
- Folder name: `backup/`
- Click "**Create folder**"

3. **Create Subfolders in orders/**
   - Click into the `orders/` folder
   - Create these subfolders:
     - `processed/` - For successfully processed orders
     - `failed/` - For orders that failed processing
     - `pending/` - For orders awaiting processing
4. **Verify Structure**
   - Your bucket should now have this structure:

```
order-data-[YOUR-STUDENT-ID]/
          orders/
                processed/
                failed/
                pending/
          logs/
          backup/
```

Listing 1: S3 Bucket Structure

2.3.2 Taking Screenshot

1. **Capture Bucket List**
   - Take screenshot showing your bucket in S3 console
   - Save as `04-s3-bucket-list.png`
2. **Capture Folder Structure**
   - Navigate into your bucket
   - Take screenshot showing all folders
   - Save as `05-s3-folders.png`

2.4 Step 3: Configure Bucket Properties

2.4.1 Enable Default Encryption

1. **Open Bucket Properties**
   - Click your bucket name
   - Go to **"Properties"** tab
2. **Configure Default Encryption**
   - Scroll to **"Default encryption"**
   - Click **"Edit"**
   - Select: **"Enable"**
   - Encryption type: **"SSE-S3"**
   - Bucket key: **"Enable"**
   - Click **"Save changes"**

2.4.2 Add Lifecycle Rules (Optional)

1. **Go to Management Tab**
   - Click **"Management"** tab
   - Click **"Create lifecycle rule"**
2. **Configure Rule**
   - Rule name: `auto-cleanup-old-files`
   - Scope: **"Apply to all objects in the bucket"**
   - Click **"Next"**
   - Under **"Lifecycle rule actions"**
   - Check: **"Expire current versions of objects"**
   - Days after creation: `30`
   - Click **"Next"** twice
   - Review and click **"Create rule"**

Note

The lifecycle rule will automatically delete files older than 30 days, helping you stay within Free Tier limits and maintain good data hygiene.

2.5 Step 4: Test Bucket Access

2.5.1 Upload Test File

1. **Upload Sample Order**
   - Navigate to `orders/pending/` folder
   - Click "**Upload**"
   - Click "**Add files**"
   - Create a text file `test-order.json` with:

```
{
    "test": "This is a test file",
    "timestamp": "2024-01-01T12:00:00Z",
    "purpose": "Verify S3 bucket access"
}
```

Listing 2: Test Order JSON

2. **Complete Upload**
   - Click "**Next**"
   - Storage class: "**Standard**"
   - Encryption: "**Use bucket settings**"
   - Click "**Next**" twice
   - Click "**Upload**"
3. **Verify Upload**
   - You should see the file in `orders/pending/`
   - Click on the file
   - Click "**Open**" to view contents

2.5.2 Delete Test File

1. **Clean Up Test File**
   - Check the box next to `test-order.json`
   - Click "**Delete**"
   - Confirm deletion

2.6 Step 5: Record Bucket Information

2.6.1 Important Details to Record

Create a text file called `bucket-info.txt` with:

```
S3 BUCKET INFORMATION
=====================
Bucket Name: order-data-[YOUR-STUDENT-ID]
Bucket ARN: arn:aws:s3:::order-data-[YOUR-STUDENT-ID
Region: us-east-1
Creation Date: [Date of creation]
Folders Created:
  - orders/
  - orders/processed/
  - orders/failed/
  - orders/pending/
  - logs/
  - backup/

Encryption: Enabled (SSE-S3)
Public Access: Blocked
Lifecycle Rule: Delete after 30 days (optional)
```

Listing 3: Bucket Information File

2.7 Phase 2 Completion Checklist

Deliverable	**Status**
S3 bucket created with unique name	
Bucket created in us-east-1 region	
All public access blocked	
Default encryption enabled	
Folder structure created	
Test file uploaded and deleted	
Bucket information documented	
Screenshots taken (2 required)	

Table 6: Phase 2 Completion Requirements

2.8 Troubleshooting S3 Issues

1. **"Bucket name already exists"**
   - S3 bucket names are globally unique
   - Try adding your student ID or timestamp
   - Example: `order-data-s1234567-20240101`

2. **Cannot upload files**
    - Check IAM user has S3 permissions
    - Verify you're in the correct folder
    - Check file size (should be small for testing)
3. **Cannot see encryption settings**
    - Refresh the page
    - Check you're in "Properties" tab
    - Ensure you have proper permissions

2.9 Phase 2 Summary

You have successfully:

- Created a secure S3 bucket with encryption
- Organized data with logical folder structure
- Configured security settings (no public access)
- Tested basic bucket operations
- Documented bucket information for future reference

Tip

Remember: Your bucket name is unique. Write it down as you'll need it in Phase 3 for the Lambda function code.

3 Phase 3: AWS Lambda Functions

Note

Time Estimate: 90-120 minutes
Completion Criteria: Three Lambda functions created, configured, and tested

3.1 Introduction to AWS Lambda

AWS Lambda is a serverless compute service that runs your code in response to events. In our project, we'll create three Lambda functions:

Function	Purpose
validate-order	Validates incoming order data structure
process-payment	Simulates payment processing (85% success rate)
save-to-s3	Saves processed orders to S3 bucket

Table 7: Lambda Functions Overview

3.2 Step 1: Create First Lambda Function

3.2.1 Navigation to Lambda Console

1. **Open AWS Management Console**
   - Ensure you're logged in as `student-dev-user`
   - Region: **us-east-1**
2. **Find Lambda Service**
   - Search for "Lambda" in services search bar
   - Click on **"Lambda"**
   - Or go to: Lambda Console

3.2.2 Create validate-order Function

1. **Start Creation**
   - Click **"Create function"**
2. **Choose Function Type**
   - Select **"Author from scratch"**

3. **Basic Information**
    - **Function name:** `validate-order`
    - **Runtime:** `Python 3.9`
    - **Architecture:** `x86_64`
    - Click **"Create function"**
4. **Configure Function Code**
    - In the function page, find **"Code source"** section
    - Delete the default code in `lambda_function.py`
    - Copy and paste the following code:

```python
import json
import boto3
from datetime import datetime
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    """
    Validates incoming order data
    Returns: Validated order or error message
    """

    logger.info(f"Received event: {json.dumps(event)}")

    # Check if event comes from Step Functions
    if 'body' in event:
        try:
            event = json.loads(event['body'])
        except:
            pass

    # Required fields for order validation
    required_fields = [
        'order_id',
        'customer_id',
        'items',
        'total_amount'
    ]

    # Check for missing fields
    missing_fields = []
    for field in required_fields:
        if field not in event:
            missing_fields.append(field)

    if missing_fields:
        error_message = f"Missing required fields: {missing_fields}"
        logger.error(error_message)
        return {
            'statusCode': 400,
```

```
            'error': error_message,
            'missing_fields': missing_fields
        }

    # Validate items array
    if not isinstance(event['items'], list) or len(event['items']) ==
        0:
        return {
            'statusCode': 400,
            'error': 'Items must be a non-empty array'
        }

    # Validate total amount
    try:
        total = float(event['total_amount'])
        if total <= 0:
            raise ValueError("Total must be positive")
    except ValueError as e:
        return {
            'statusCode': 400,
            'error': f'Invalid total_amount: {str(e)}'
        }

    # Add metadata
    event['validation_timestamp'] = datetime.now().isoformat()
    event['status'] = 'VALIDATED'
    event['validation_id'] = context.aws_request_id

    logger.info(f"Order validated: {event['order_id']}")

    # Return format suitable for Step Functions
    return {
        'statusCode': 200,
        'order': event,
        'message': 'Order validated successfully'
    }
```

Listing 4: validate-order Lambda Function Code

3.2.3 Deploy and Configure

1. **Deploy Function**
   - Click "**Deploy**" button (orange)
2. **Configure Basic Settings**
   - Click "**Configuration**" tab
   - Click "**General configuration**"
   - Click "**Edit**"
   - **Description:** `Validates e-commerce order data`
   - **Timeout:** `30 seconds`

- **Memory:** `128 MB`
- Click "**Save**"

3. **Take Screenshot**
   - Capture the function overview page
   - Save as `06-lambda-validate-created.png`

3.3 Step 2: Create Second Lambda Function

3.3.1 Create process-payment Function

1. **Return to Lambda Home**
   - Click "**Functions**" in left sidebar
   - Click "**Create function**"
2. **Function Configuration**
   - **Author from scratch**
   - **Function name:** `process-payment`
   - **Runtime:** `Python 3.9`
   - Click "**Create function**"
3. **Add Code**
   - Delete default code
   - Paste this code:

```python
import json
import random
from datetime import datetime
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    """
    Simulates payment processing
    Returns: Updated order with payment status
    """

    logger.info(f"Processing payment for order")

    # Handle input from Step Functions or direct invocation
    if 'order' in event:
        order_data = event['order']
    elif 'body' in event:
```

```
        try:
            order_data = json.loads(event['body'])
        except:
            order_data = event
    else:
        order_data = event

    # Log the order ID if available
    order_id = order_data.get('order_id', 'UNKNOWN')
    logger.info(f"Processing payment for order: {order_id}")

    # Simulate payment processing with 85% success rate
    payment_statuses = ['SUCCESS', 'FAILED']
    weights = [0.85, 0.15]  # 85% success, 15% failure

    chosen_status = random.choices(
        payment_statuses,
        weights=weights,
        k=1
    )[0]

    # Add payment information
    order_data['payment_status'] = chosen_status
    order_data['payment_timestamp'] = datetime.now().isoformat()
    order_data['transaction_id'] = f"TXN-{random.randint(10000, 99999)}
        "

    # Update overall status
    if chosen_status == 'SUCCESS':
        order_data['status'] = 'PAYMENT_SUCCESS'
        logger.info(f"Payment successful for order: {order_id}")
    else:
        order_data['status'] = 'PAYMENT_FAILED'
        order_data['failure_reason'] = 'Insufficient funds or card
            declined'
        logger.warning(f"Payment failed for order: {order_id}")

    # Return format for Step Functions
    return {
        'statusCode': 200,
        'order': order_data,
        'message': f'Payment processing completed: {chosen_status}'
    }
```

Listing 5: process-payment Lambda Function Code

3.3.2 Deploy and Configure

1. **Deploy Function**
   - Click "**Deploy**"
2. **Configure Settings**
   - Go to "**Configuration**" → "**General configuration**"

- Click "**Edit**"
- **Description:** `Simulates payment processing for orders`
- **Timeout:** `30 seconds`
- **Memory:** `128 MB`
- Click "**Save**"

3. **Take Screenshot**
   - Save as `07-lambda-payment-created.png`

3.4 Step 3: Create Third Lambda Function

3.4.1 Create save-to-s3 Function

1. **Create New Function**
   - **Functions → "Create function"**
   - **Author from scratch**
   - **Function name:** `save-to-s3`
   - **Runtime:** `Python 3.9`
   - Click "**Create function**"
2. **Important: Update Bucket Name**
   - In the code below, replace `order-data-[YOUR-STUDENT-ID]` with your actual bucket name
   - This is on line 22 of the code
3. **Add Code**

```
1  import json
2  import boto3
3  from datetime import datetime
4  import logging
5
6  logger = logging.getLogger()
7  logger.setLevel(logging.INFO)
8
9  s3 = boto3.client('s3')
10
11 def lambda_handler(event, context):
12     """
13     Saves order data to S3 bucket
14     """
15
16     logger.info(f"Saving order to S3: {event}")
17
18     # Extract order data
```

```
    if 'order' in event:
        order_data = event['order']
    else:
        order_data = event

    order_id = order_data.get('order_id', 'UNKNOWN')

    # IMPORTANT: Update this with your actual bucket name
    bucket_name = 'order-data-[YOUR-STUDENT-ID]'  # CHANGE THIS!

    # Create file name with timestamp
    timestamp = datetime.now().strftime('%Y-%m-%d-%H-%M-%S')
    file_name = f"orders/processed/{order_id}-{timestamp}.json"

    try:
        # Save to S3
        s3.put_object(
            Bucket=bucket_name,
            Key=file_name,
            Body=json.dumps(order_data, indent=2),
            ContentType='application/json',
            Metadata={
                'order-id': order_id,
                'processed-by': 'save-to-s3-lambda'
            }
        )

        logger.info(f"Successfully saved order {order_id} to S3: {
            file_name}")

        # Add S3 location to order data
        order_data['s3_location'] = {
            'bucket': bucket_name,
            'key': file_name,
            'url': f"s3://{bucket_name}/{file_name}"
        }

        return {
            'statusCode': 200,
            'order': order_data,
            's3_location': f"s3://{bucket_name}/{file_name}",
            'message': f'Order {order_id} saved to S3 successfully'
        }

    except Exception as e:
        logger.error(f"Failed to save order to S3: {str(e)}")
        return {
            'statusCode': 500,
            'error': str(e),
            'message': 'Failed to save order to S3'
        }
```

Listing 6: save-to-s3 Lambda Function Code

Warning

Critical Step: You MUST update line 22 with your actual S3 bucket name. If you skip this step, the function will fail when trying to save to S3.

3.4.2 Deploy and Configure

1. **Deploy Function**
   - Click "**Deploy**"
2. **Configure Settings**
   - **Configuration** → "**General configuration**" → "**Edit**"
   - **Description:** `Saves processed orders to S3 bucket`
   - **Timeout:** `1 minute`
   - **Memory:** `128 MB`
   - Click "**Save**"
3. **Take Screenshot**
   - Save as `08-lambda-s3-created.png`

3.5 Step 4: Test Lambda Functions

3.5.1 Test validate-order Function

1. **Navigate to validate-order**
   - Click "**Functions**" in left sidebar
   - Click `validate-order`
2. **Create Test Event**
   - Click "**Test**" tab
   - Click "**Create new event**"
   - **Event name:** `TestValidOrder`
   - Paste this JSON:

```
{
  "order_id": "ORD-001",
  "customer_id": "CUST-001",
  "items": [
    {
      "product_id": "P-001",
      "name": "Wireless Headphones",
```

```
      "quantity": 1,
      "price": 99.99
    }
  ],
  "total_amount": 99.99,
  "customer_email": "test@example.com"
}
```

Listing 7: Test Event for validate-order

1. **Run Test**
   - Click "**Test**"
   - Observe the execution result
   - Should show "Order validated successfully"
2. **Test Invalid Order**
   - Create another test event `TestInvalidOrder`
   - Remove "order_id" field from JSON
   - Click "**Test**"
   - Should show error about missing fields

3.5.2 Test process-payment Function

1. **Navigate to process-payment**
2. **Create Test Event**
   - Event name: `TestPayment`
   - Use this JSON:

```
{
  "order": {
    "order_id": "ORD-001",
    "customer_id": "CUST-001",
    "total_amount": 99.99,
    "status": "VALIDATED"
  }
}
```

Listing 8: Test Event for process-payment

1. **Run Multiple Tests**
   - Click "**Test**" several times
   - Observe that sometimes payment succeeds, sometimes fails
   - This is expected (85% success rate simulation)

3.5.3 Test save-to-s3 Function

1. **Navigate to save-to-s3**
2. **Create Test Event**
   - Event name: `TestSaveToS3`
   - Use this JSON:

```
{
  "order": {
    "order_id": "ORD-TEST-001",
    "customer_id": "CUST-TEST-001",
    "total_amount": 50.00,
    "status": "PAYMENT_SUCCESS",
    "payment_status": "SUCCESS"
  }
}
```

Listing 9: Test Event for save-to-s3

1. **Run Test**
   - Click **"Test"**
   - Should show "Order saved to S3 successfully"
2. **Verify in S3**
   - Go to S3 Console
   - Navigate to your bucket → `orders/processed/`
   - You should see a JSON file
   - Click to open and verify contents

3.6 Step 5: Verify Function Details

3.6.1 Record Function ARNs

Create a file `lambda-info.txt` with:

```
LAMBDA FUNCTION INFORMATION
===========================

1. validate-order
   - ARN: arn:aws:lambda:us-east-1:[ACCOUNT-ID]:function:validate-order
   - Runtime: Python 3.9
   - Memory: 128 MB
   - Timeout: 30 seconds

2. process-payment
```

```
   - ARN: arn:aws:lambda:us-east-1:[ACCOUNT-ID]:function:process-
     payment
   - Runtime: Python 3.9
   - Memory: 128 MB
   - Timeout: 30 seconds

3. save-to-s3
   - ARN: arn:aws:lambda:us-east-1:[ACCOUNT-ID]:function:save-to-s3
   - Runtime: Python 3.9
   - Memory: 128 MB
   - Timeout: 1 minute

Note: Replace [ACCOUNT-ID] with your 12-digit AWS account ID
```

Listing 10: Lambda Function Information

3.6.2 Find Your Account ID

1. **Click your account name** (top-right corner)
2. **Select "Switch Roles"** (if available)
3. **Your account ID** is displayed
4. **Or go to IAM Console** → It shows in the URL

3.7 Phase 3 Completion Checklist

Deliverable	Status
validate-order function created	
process-payment function created	
save-to-s3 function created	
All functions deployed successfully	
Bucket name updated in save-to-s3 code	
Each function tested individually	
Test results verified	
S3 file created from save-to-s3 test	
Function ARNs recorded	
Screenshots taken (3+ required)	

Table 8: Phase 3 Completion Requirements

3.8 Troubleshooting Lambda Issues

1. **"Deployment failed"**
   - Check Python syntax in your code

- Ensure proper indentation
- Check for missing quotes or brackets

2. **"Access Denied" when saving to S3**
   - Verify IAM role has S3 permissions
   - Check bucket name is correct
   - Ensure bucket exists in us-east-1
3. **Test events not working**
   - Ensure JSON is valid
   - Check event structure matches what function expects
   - Try simpler test events first
4. **Timeout errors**
   - Increase timeout in configuration
   - 30 seconds is usually sufficient

3.9 Phase 3 Summary

You have successfully:

- Created three Lambda functions with Python code
- Configured appropriate memory and timeout settings
- Tested each function individually
- Verified S3 integration works
- Documented function details for next phase

Tip

Pro Tip: Use CloudWatch Logs to debug Lambda functions. Click "Monitor" tab in any Lambda function to view execution logs.

4 Phase 4: AWS Step Functions

Note

Time Estimate: 60-90 minutes
Completion Criteria: Step Functions state machine created, configured, and tested

4.1 Introduction to AWS Step Functions

AWS Step Functions is a visual workflow service that orchestrates AWS services. It allows you to build applications from individual components that each perform a discrete function.

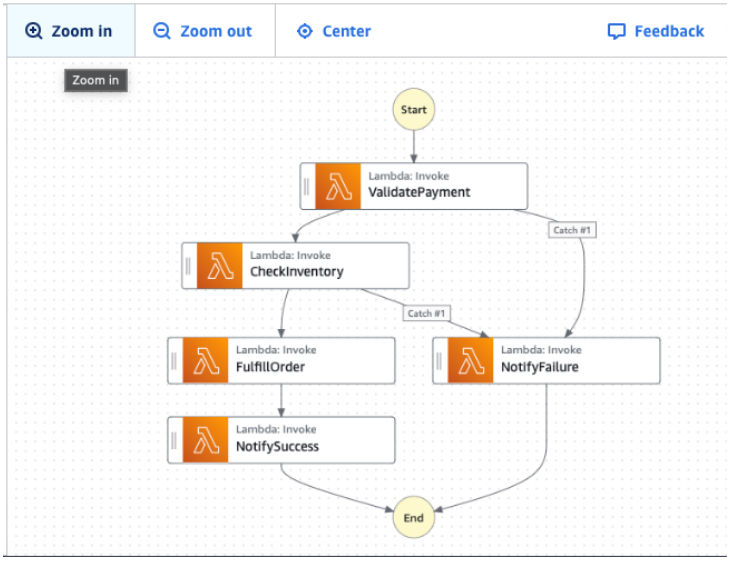

Figure 4: Step Functions Visual Workflow Example

4.2 Step 1: Create Step Functions State Machine

4.2.1 Navigation to Step Functions Console

1. **Open AWS Management Console**
   - Logged in as `student-dev-user`
   - Region: **us-east-1**
2. **Find Step Functions Service**
   - Search for "Step Functions"
   - Click on **"Step Functions"**

- Or go to: Step Functions Console

4.2.2 Create New State Machine

1. **Start Creation**
   - Click **"Create state machine"**
2. **Choose Design Method**
   - Select **"Create from blank"**
   - **State Machine Details**
   - **State machine name:** `OrderProcessingWorkflow`
   - **Type:** `Standard`
   - Click **"continue"**

4.3 Step 2: Design Visual Workflow

4.3.1 Add First State - Validate Order

1. **Add Task State**
   - From left panel, under **Actions** drag **"AWS Lambda task"** to the canvas
2. **Configure State**
   - Click the new state on canvas
   - Right panel opens
   - **State name:** `ValidateOrder`
   - **Integration type: Optimized**
3. **Configure API Arguments**
   - **Function name:** Click dropdown, select `validate-order`
   - The ARN will auto-populate

4.3.2 Add Second State - Process Payment

1. **Add Another Task**
   - From left panel, under **Actions** drag another **"AWS Lambda task"** to the canvas and add it below ValidateOrder. The states will connect automatically
2. **Configure State**

- Click the new state
- **State name:** `ProcessPayment`
- **Integration type: Optimized**
- **Function name:** Select `process-payment`

4.3.3 Add Choice State - Payment Decision

1. **Add Choice State**
   - From left panel, under **flow** drag **"Choice"** to canvas
   - Connect ProcessPayment to Choice state
2. **Configure Choice State**
   - Click the Choice state
   - **State name:** `PaymentDecision`
3. **Add First Rule**
   - Click **"Rule** #1"
   - Choice rules, click **Add conditions**
   - **Expression:** add `$order.payment_status`
   - **Operator:** `is equal to`
   - **Value:** `String`
   - **Enter value:** `SUCCESS`
   - **Next state:** `SaveToS3` (we'll create this next)
4. **Add Second Rule**
   - Click **"Add new choice rule"**
   - **Expression:** add `$order.payment_status`
   - **Operator:** `is equal to`
   - **Value:** `String`
   - **Enter value:** `FAILED`

4.3.4 Add Success Path - Save to S3

1. **Add Task State**
   - Drag **"Task"** to canvas
   - Connect from PaymentDecision (SUCCESS rule) to this state
   - **State name:** `SaveToS3`

- **Service:** `AWS Lambda`
- **Function name:** Select `save-to-s3`

2. **Add Success State**
   - From left panel, under flow, drag **"Succees"** to canvas
   - Connect SaveToS3 to Succees state
   - **State name:** `OrderProcessedSuccessfully`

4.3.5 Add Failure Path

1. **Add Fail State**
   - From left panel, under flow drag **"Fail"** to canvas
   - Connect from PaymentDecision (FAILED rule) to this state
   - Connect Default rule to this state
   - **State name:** `PaymentFailed`
   - **Error:** `PaymentProcessingError`
   - **Cause:** `Payment failed for the order`

4.3.6 Add Error Handling

1. **Configure ValidateOrder Errors**
   - Click `ValidateOrder` state
   - In right panel, click **"Error handling"**
   - Click **"Add error handler"**
   - **Errors:** Select `States.ALL`
   - **Fallback state:** `ValidationFailed`
   - Click **"Apply"**

2. **Create Validation Failed State**
   - Drag fail state from the left panel and Add **"Fail"** state
   - **State name:** `ValidationFailed`
   - **Error:** `ValidationError`
   - **Cause:** `Order validation failed`
   - Connect error handler from ValidateOrder to this state

4.4 Step 3: Review and Create State Machine

4.4.1 Final Workflow Structure

Your workflow should look like this:

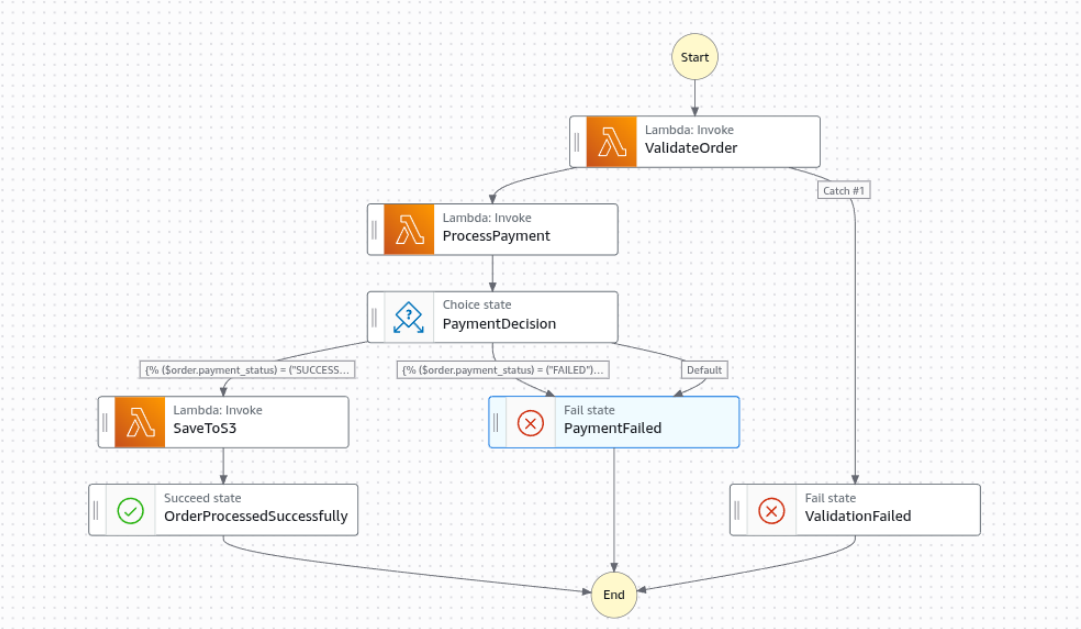

Figure 5: Complete Step Functions Workflow

4.4.2 Create State Machine

1. **Click "create"**
2. **State Machine Details**
   - **State machine name:** `OrderProcessingWorkflow`
   - **Type:** `Standard`
3. **Success Confirmation**
   - You should see "State machine created successfully"
   - Take screenshot: `09-stepfunctions-created.png`

4.5 Step 4: Test the Workflow

4.5.1 Prepare Test Input

1. **Copy Test JSON**

- Use this sample order data for testing:

```
{
  "order_id": "ORD-001",
  "customer_id": "CUST-001",
  "items": [
    {
      "product_id": "P-001",
      "name": "Wireless Headphones",
      "quantity": 1,
      "price": 99.99
    },
    {
      "product_id": "P-002",
      "name": "USB-C Cable",
      "quantity": 2,
      "price": 12.50
    }
  ],
  "total_amount": 124.99,
  "customer_email": "student@example.edu",
  "shipping_address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

Listing 11: Test Order for Step Functions

4.5.2 Start Execution

1. **Start New Execution**
   - On state machine page, click **"Start execution"**
2. **Enter Execution Details**
   - **Input:** Paste the test JSON above
   - **Execution name:** (optional) `TestExecution-1`
   - Click **"Start execution"**
3. **Monitor Execution**
   - Watch the visual execution flow
   - States will light up as they execute
   - Wait for completion (30-60 seconds)
4. **Take Screenshot**
   - Capture the successful execution
   - Save as `10-stepfunctions-success.png`

4.5.3 Test Failure Scenario

1. **Create Invalid Order**
   - Start new execution
   - Use this invalid JSON (missing total_amount):

```
{
  "order_id": "ORD-002",
  "customer_id": "CUST-002",
  "items": [
    {
      "product_id": "P-003",
      "name": "Laptop Case",
      "quantity": 1
    }
  ]
}
```

Listing 12: Invalid Test Order

1. **Observe Failure**
   - Execution should fail at ValidateOrder
   - Should go to ValidationFailed state
   - Take screenshot: `11-stepfunctions-failure.png`

4.6 Step 5: Verify Results

4.6.1 Check S3 for Successful Orders

1. **Open S3 Console**
   - Navigate to your bucket
   - Go to `orders/processed/` folder
   - You should see JSON files for successful orders
2. **View File Contents**
   - Click on a file
   - Click "**Open**"
   - Verify it contains complete order data
   - Should include S3 location metadata

4.6.2 Check CloudWatch Logs

1. **Open Lambda Console**
   - Go to any Lambda function
   - Click **"Monitor"** tab
   - Click **"View CloudWatch logs"**
2. **Review Logs**
   - You should see logs for each execution
   - Logs show function execution details
   - Helpful for debugging

4.7 Step 6: Record State Machine Information

4.7.1 Create Documentation File

Create `stepfunctions-info.txt`:

```
STEP FUNCTIONS INFORMATION
==========================

State Machine Name: OrderProcessingWorkflow
State Machine ARN: arn:aws:states:us-east-1:[ACCOUNT-ID]:stateMachine:
    OrderProcessingWorkflow
Type: Standard
Creation Date: [Date]

States:
1. ValidateOrder - Calls validate-order Lambda
2. ProcessPayment - Calls process-payment Lambda
3. PaymentDecision - Choice state based on payment_status
4. SaveToS3 - Calls save-to-s3 Lambda (success path)
5. OrderProcessedSuccessfully - Succeed state
6. PaymentFailed - Fail state for payment failures
7. ValidationFailed - Fail state for validation errors

Error Handling:
- ValidateOrder catches States.ALL     ValidationFailed
- ProcessPayment errors     Propagate to parent

Test Results:
- Successful execution: [Yes/No]
- Failure execution: [Yes/No]
- S3 files created: [Number] files in orders/processed/
```

Listing 13: Step Functions Information

4.8 Phase 4 Completion Checklist

Deliverable	Status
State machine created successfully	
All states configured correctly	
Lambda functions connected properly	
Error handling configured	
Successful execution tested	
Failure scenario tested	
S3 files verified for successful orders	
State machine ARN recorded	
Screenshots taken (3+ required)	

Table 9: Phase 4 Completion Requirements

4.9 Troubleshooting Step Functions Issues

1. **"State machine creation failed"**
   - Check IAM permissions for Step Functions
   - Ensure Lambda functions exist in same region
   - Verify no syntax errors in workflow definition
2. **"Execution stuck at a state"**
   - Check CloudWatch logs for the Lambda function
   - Verify Lambda function timeout settings
   - Check input format matches function expectations
3. **"Cannot connect Lambda function"**
   - Ensure Lambda function exists
   - Check function name spelling
   - Verify both services in same region
4. **"Choice state not working"**
   - Check variable path is correct ($.order.payment_status)
   - Verify the variable actually exists in the data
   - Check value matches exactly (case-sensitive)

4.10 Phase 4 Summary

You have successfully:

- Created a Step Functions state machine visually
- Connected three Lambda functions in a workflow
- Implemented decision logic with Choice state
- Added error handling for failures
- Tested both success and failure scenarios
- Verified end-to-end integration works

Tip

Best Practice: Use meaningful state names and add comments in your workflow. This makes it easier to understand and maintain.

5 Phase 5: Testing & Monitoring

Note

Time Estimate: 45-60 minutes
Completion Criteria: Comprehensive testing completed, monitoring dashboard created

5.1 Introduction to Testing Strategy

A robust testing strategy ensures your serverless application works correctly under various conditions. We'll test:

- Individual Lambda functions
- End-to-end workflow
- Error conditions
- Integration points

5.2 Step 1: Comprehensive Testing

5.2.1 Test Case 1: Valid Order with Successful Payment

1. **Test Data**
    - Use this JSON for testing:

```
{
  "order_id": "TEST-SUCCESS-001",
  "customer_id": "CUST-TEST-001",
  "items": [
    {
      "product_id": "P-TEST-001",
      "name": "Test Product",
      "quantity": 3,
      "price": 25.00
    }
  ],
  "total_amount": 75.00,
  "customer_email": "test-success@example.com",
  "shipping_address": {
    "street": "456 Test Ave",
    "city": "Testville",
    "state": "TS",
    "zip": "54321"
```

```
  }
}
```

Listing 14: Test Case 1: Valid Order

1. **Execute Test**
   - Go to Step Functions Console
   - Start execution with above input
   - Record execution ID
2. **Expected Results**
   - All states should execute successfully
   - Payment should succeed (85% probability)
   - File should be saved to S3
   - Execution should end at OrderProcessedSuccessfully
3. **Verification Steps**
   - Check S3 for new file in `orders/processed/`
   - Verify file contains complete order data
   - Check CloudWatch logs for each Lambda function

5.2.2 Test Case 2: Valid Order with Failed Payment

1. **Test Strategy**
   - Since payment failure is random (15% chance)
   - We may need multiple executions to trigger failure
   - Alternative: Temporarily modify process-payment to always fail
2. **Temporary Modification (Optional)**
   - Go to process-payment Lambda function
   - Change line with random.choices weights to:
   - `weights = [0.00, 1.00]` # 0% success, 100% failure
   - Deploy the function
   - Run test
   - Change back to `weights = [0.85, 0.15]` after test
3. **Expected Results**
   - ValidateOrder should succeed
   - ProcessPayment should fail
   - Execution should go to PaymentFailed state
   - No file should be saved to S3

5.2.3 Test Case 3: Invalid Order Data

1. **Test Data**
   - Create order missing required fields:

```
{
  "order_id": "TEST-INVALID-001",
  "customer_id": "CUST-INVALID-001",
  "items": [],  # Empty array should fail validation
  "total_amount": 0  # Zero amount should fail
}
```

Listing 15: Test Case 3: Invalid Order

1. **Expected Results**
   - ValidateOrder should fail
   - Execution should go to ValidationFailed state
   - Error should indicate the specific validation failure

5.2.4 Test Case 4: Edge Cases

1. **Large Order Amount**
   - Test with total_amount: 999999.99
   - Should process normally
2. **Many Items**
   - Test with 10+ items in array
   - Should process normally
3. **Special Characters**
   - Test with special characters in names/addresses
   - Should handle UTF-8 encoding properly

5.3 Step 2: Create CloudWatch Dashboard

5.3.1 Navigation to CloudWatch

1. **Open CloudWatch Console**
   - Search "CloudWatch" in services

- Or go to: CloudWatch Console

2. **Create Dashboard**
   - Left sidebar → **"Dashboards"**
   - Click **"Create dashboard"**
   - **Dashboard name:** `Serverless-Order-Monitoring`
   - Click **"Create dashboard"**

5.3.2 Add Lambda Metrics Widget

1. **Add First Widget**
   - Select **"Line"** widget type
   - Click **"Metrics"**
2. **Select Metrics**
   - Select **"AWS/Lambda"**
   - Select all three Lambda functions
   - Select **"Invocations"** metric
   - Click **"Create widget"**
3. **Configure Widget**
   - Title: `Lambda Invocations`
   - Click **"Create widget"**

5.3.3 Add Step Functions Widget

1. **Add Another Widget**
   - Click **"Add widget"**
   - Select **"Number"** widget type
2. **Select Metrics**
   - Click **"Metrics"**
   - Select **"AWS/States"**
   - Select your state machine
   - Select **"ExecutionsSucceeded"** and **"ExecutionsFailed"**
   - Click **"Create widget"**
3. **Configure Widget**
   - Title: `Step Functions Executions`
   - Click **"Create widget"**

5.3.4 Add S3 Metrics Widget

1. **Add Third Widget**
   - **"Add widget"** → **"Number"**
   - Click **"Metrics"**
2. **Select Metrics**
   - Select **"AWS/S3"**
   - Select your bucket
   - Select **"NumberOfObjects"** metric
   - Click **"Create widget"**
3. **Configure Widget**
   - Title: `S3 Objects Count`
   - Click **"Create widget"**

5.3.5 Final Dashboard Layout

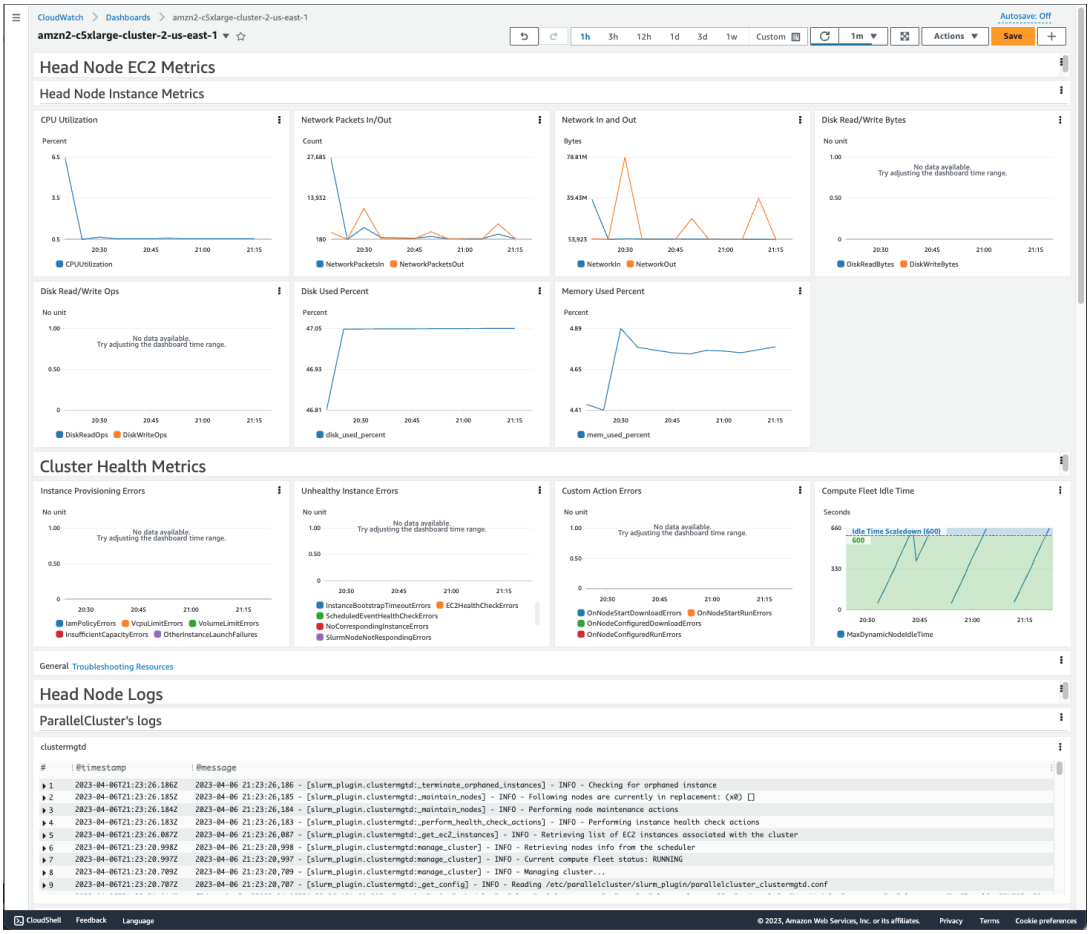

Figure 6: CloudWatch Monitoring Dashboard

5.4 Step 3: Create CloudWatch Alarms

5.4.1 Alarm for Step Functions Failures

1. **Create New Alarm**
   - CloudWatch Console → **"Alarms"** → **"All Alarms"**
   - Click **"Create alarm"**
2. **Select Metric**
   - Click **"Select metric"**
   - Select **"AWS/States"**
   - Select your state machine
   - Select **"ExecutionsFailed"**
   - Click **"Select metric"**
3. **Configure Conditions**
   - Statistic: **"Sum"**
   - Period: **"5 minutes"**
   - Conditions: **"Greater/Equal"** → **"1"**
   - Click **"Next"**
4. **Configure Actions**
   - Notification: **"In alarm"**
   - Select existing SNS topic: `Billing-Alarm-Notification`
   - Click **"Next"**
5. **Name and Create**
   - Alarm name: `StepFunctions-Failure-Alarm`
   - Description: `Alert when Step Functions executions fail`
   - Click **"Next"**
   - Review and click **"Create alarm"**

5.4.2 Alarm for Lambda Errors

1. **Create Another Alarm**
   - Similar process for Lambda errors
   - Metric: **"AWS/Lambda"** → **"Errors"** (sum)
   - Select all three Lambda functions
   - Condition: **"Greater than"** → **"0"**
   - Alarm name: `Lambda-Error-Alarm`

5.5 Step 4: Review CloudWatch Logs

5.5.1 Access Lambda Logs

1. **Open Lambda Console**
   - Select any Lambda function
   - Click **"Monitor"** tab
   - Click **"View CloudWatch logs"**
2. **Analyze Log Streams**
   - You'll see multiple log streams (one per execution)
   - Click on a log stream to view details
   - Look for:
     - START and END messages
     - INFO level messages
     - ERROR messages (if any)
     - Execution duration

5.5.2 Access Step Functions Logs

1. **CloudWatch Logs Console**
   - Left sidebar → **"Log groups"**
   - Search for `/aws/states/OrderProcessingWorkflow`
   - Click to view log streams
2. **Review Execution History**
   - Each execution creates a log stream
   - Contains detailed state transition information
   - Useful for debugging workflow issues

5.6 Step 5: Performance Testing

5.6.1 Test Multiple Concurrent Executions

1. **Create Test Script (Optional)**
   - Advanced students can create a Python script
   - Use boto3 to start multiple executions

- Test with 5-10 concurrent orders

2. **Monitor Performance**
   - Watch CloudWatch metrics during load
   - Check Lambda concurrent executions
   - Monitor execution durations

5.6.2 Analyze Cost Implications

1. **Check AWS Cost Explorer**
   - Go to Billing Console
   - Click **"Cost Explorer"**
   - View service costs (should be 0.00)

2. **Estimate Monthly Costs**
   - Based on your usage, estimate if you'd exceed Free Tier
   - Lambda: 1M requests free
   - Step Functions: 4,000 state transitions free
   - S3: 5GB storage free

5.7 Step 6: Documentation

5.7.1 Create Test Report

Create `test-report.txt`:

```
TESTING REPORT
==============

Date: [Date of testing]
Tester: [Your Name]

Test Cases Executed:
1. Valid Order with Successful Payment
   - Status: PASS/FAIL
   - Execution ID: [ID]
   - Notes: [Any observations]

2. Valid Order with Failed Payment
   - Status: PASS/FAIL
   - Execution ID: [ID]
   - Notes: [Any observations]

3. Invalid Order Data
   - Status: PASS/FAIL
```

```
   - Execution ID: [ID]
   - Notes: [Any observations]

4. Edge Cases
   - Status: PASS/FAIL
   - Notes: [Any observations]

Monitoring Setup:
- CloudWatch Dashboard: Created [Yes/No]
- Dashboard Name: [Name]
- Widgets Created: [Number]
- Alarms Configured: [Number]

Performance Observations:
- Average Lambda execution time: [Time]
- Step Functions workflow duration: [Time]
- S3 write performance: [Observations]

Issues Found:
1. [Issue description]
   - Severity: High/Medium/Low
   - Resolution: [How it was fixed]

2. [Issue description]
   - Severity: High/Medium/Low
   - Resolution: [How it was fixed]

Recommendations:
1. [Suggestion for improvement]
2. [Suggestion for improvement]

Overall Assessment:
- Application Stability: Excellent/Good/Fair/Poor
- Ready for Production: Yes/No (with conditions)
- Free Tier Compliance: Yes/No
```

Listing 16: Test Report Template

5.8 Phase 5 Completion Checklist

Deliverable	**Status**
All test cases executed	
Test results documented	
CloudWatch dashboard created	
CloudWatch alarms configured	
Logs reviewed for all services	
Performance observations recorded	
Test report completed	
Screenshots taken (dashboard, logs, alarms)	

Table 10: Phase 5 Completion Requirements

5.9 Troubleshooting Testing Issues

1. **Metrics not showing in CloudWatch**
   - Wait 1-2 minutes for metrics to appear
   - Ensure you're in correct region
   - Verify services are actually being used
2. **Alarms not triggering**
   - Check alarm threshold is appropriate
   - Verify SNS topic is confirmed
   - Check alarm is in "Insufficient data" state initially
3. **Logs not appearing**
   - Ensure Lambda functions have CloudWatch permissions
   - Check log group retention settings
   - Verify functions are actually being executed

5.10 Phase 5 Summary

You have successfully:

- Executed comprehensive test cases
- Created monitoring dashboard in CloudWatch
- Configured alarms for failures and errors
- Reviewed and analyzed logs
- Documented test results and observations
- Assessed application performance and stability

Tip

Continuous Monitoring: Regularly check your CloudWatch dashboard during development. It helps catch issues early and understand system behavior.

6 Phase 6: Advanced Features (Bonus)

Note

Time Estimate: 60-90 minutes
Completion Criteria: Optional advanced features implemented and tested

6.1 Introduction to Advanced Features

This phase covers optional enhancements that demonstrate more advanced AWS serverless capabilities. These features are not required for project completion but provide valuable learning opportunities.

6.2 Feature 1: Email Notifications with Amazon SNS

6.2.1 Create SNS Topic

1. **Open SNS Console**
   - Search "SNS" in services
   - Or go to: SNS Console
2. **Create Topic**
   - Click "**Create topic**"
   - **Type:** `Standard`
   - **Name:** `OrderNotifications`
   - Click "**Create topic**"
3. **Create Subscription**
   - Click into the new topic
   - Click "**Create subscription**"
   - **Protocol:** `Email`
   - **Endpoint:** Your email address
   - Click "**Create subscription**"
4. **Confirm Subscription**
   - Check your email for confirmation message
   - Click confirmation link

6.2.2 Create Notification Lambda Function

1. **Create New Lambda Function**
   - Name: `send-notification`
   - Runtime: Python 3.9
2. **Add Code**

```python
import json
import boto3
from datetime import datetime

sns = boto3.client('sns')

def lambda_handler(event, context):
    """
    Sends notification via SNS based on order status
    """

    # Extract order data
    if 'order' in event:
        order_data = event['order']
    else:
        order_data = event

    order_id = order_data.get('order_id', 'UNKNOWN')
    status = order_data.get('status', 'UNKNOWN')

    # SNS Topic ARN - UPDATE WITH YOUR TOPIC ARN
    topic_arn = 'arn:aws:sns:us-east-1:[ACCOUNT-ID]:OrderNotifications'

    # Create message based on status
    if status == 'PAYMENT_SUCCESS':
        subject = f'Order Confirmed: {order_id}'
        message = f"""
        Order {order_id} has been successfully processed!

        Order Details:
        - Customer: {order_data.get('customer_id')}
        - Total Amount: ${order_data.get('total_amount', 0):.2f}
        - Items: {len(order_data.get('items', []))}
        - Processed at: {datetime.now().isoformat()}

        Thank you for your order!
        """
    elif status == 'PAYMENT_FAILED':
        subject = f'Order Failed: {order_id}'
        message = f"""
        Order {order_id} failed during payment processing.

        Reason: {order_data.get('failure_reason', 'Unknown error')}

        Please check your payment details and try again.
        """
    else:
```

```
48          subject = f'Order Update: {order_id}'
49          message = f"""
50          Order {order_id} status: {status}
51
52          Details: {json.dumps(order_data, indent=2)}
53          """
54
55      # Send notification
56      try:
57          response = sns.publish(
58              TopicArn=topic_arn,
59              Subject=subject,
60              Message=message
61          )
62
63          print(f"Notification sent for order {order_id}: {response['
                MessageId']}")
64
65          return {
66              'statusCode': 200,
67              'message': f'Notification sent: {subject}',
68              'sns_message_id': response['MessageId']
69          }
70
71      except Exception as e:
72          print(f"Failed to send notification: {str(e)}")
73          return {
74              'statusCode': 500,
75              'error': str(e),
76              'message': 'Failed to send notification'
77          }
```

Listing 17: SNS Notification Lambda Function

Warning

Important: Update line 20 with your actual SNS Topic ARN. You can find this in the SNS Topic details page.

6.2.3 Update Step Functions Workflow

1. **Edit State Machine**
   - Open Step Functions Console
   - Select your state machine
   - Click "**Edit**"
2. **Add Notification State**
   - Add new Task state after SaveToS3
   - Name: `SendSuccessNotification`
   - Connect SaveToS3 to this state

- Connect this state to OrderProcessedSuccessfully
- Configure to call `send-notification` Lambda

3. **Add Failure Notification**
   - Add Task state after PaymentFailed
   - Name: `SendFailureNotification`
   - Connect PaymentFailed to this state
   - Configure to call `send-notification` Lambda
4. **Deploy Changes**
   - Click **"Save"**
   - Update state machine

6.3 Feature 2: Database Storage with DynamoDB

6.3.1 Create DynamoDB Table

1. **Open DynamoDB Console**
   - Search "DynamoDB" in services
   - Or go to: DynamoDB Console
2. **Create Table**
   - Click **"Create table"**
   - **Table name:** `Orders`
   - **Partition key:** `order_id` (String)
   - **Sort key:** `timestamp` (String)
   - Click **"Create table"**
3. **Configure Table**
   - Wait for table creation (2-3 minutes)
   - Go to **"Additional settings"**
   - Enable TTL (Time to Live) for automatic cleanup
   - TTL attribute: `ttl`

6.3.2 Create DynamoDB Lambda Function

1. **Create New Lambda Function**
   - Name: `save-to-dynamodb`
   - Runtime: Python 3.9
2. **Add Code**

```python
import json
import boto3
from datetime import datetime, timedelta
import uuid

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Orders')

def lambda_handler(event, context):
    """
    Saves order data to DynamoDB table
    """

    # Extract order data
    if 'order' in event:
        order_data = event['order']
    else:
        order_data = event

    order_id = order_data.get('order_id', str(uuid.uuid4()))

    # Prepare item for DynamoDB
    item = {
        'order_id': order_id,
        'timestamp': datetime.now().isoformat(),
        'data': order_data,
        'status': order_data.get('status', 'UNKNOWN'),
        'customer_id': order_data.get('customer_id', 'UNKNOWN'),
        'total_amount': str(order_data.get('total_amount', 0)),
        # TTL: Delete after 30 days (2592000 seconds)
        'ttl': int((datetime.now() + timedelta(days=30)).timestamp())
    }

    try:
        # Save to DynamoDB
        response = table.put_item(Item=item)

        print(f"Order saved to DynamoDB: {order_id}")

        # Add DynamoDB info to order data
        order_data['dynamodb_id'] = order_id
        order_data['dynamodb_timestamp'] = item['timestamp']

        return {
            'statusCode': 200,
            'order': order_data,
            'dynamodb_response': response,
```

```
            'message': f'Order {order_id} saved to DynamoDB'
        }

    except Exception as e:
        print(f"Failed to save to DynamoDB: {str(e)}")
        return {
            'statusCode': 500,
            'error': str(e),
            'message': 'Failed to save order to DynamoDB'
        }
```

Listing 18: DynamoDB Lambda Function

6.3.3 Update IAM Permissions

1. **Add DynamoDB Permissions**
   - Go to IAM Console
   - Find the role used by Lambda functions
   - Attach policy: `AmazonDynamoDBFullAccess`

6.4 Feature 3: API Gateway Trigger

6.4.1 Create REST API

1. **Open API Gateway Console**
   - Search "API Gateway" in services
   - Or go to: API Gateway Console
2. **Create API**
   - Click "**Create API**"
   - Choose "**REST API**" → "**Build**"
   - **API name:** `OrderProcessingAPI`
   - Description: `API for submitting orders`
   - Click "**Create API**"
3. **Create Resource and Method**
   - Click "**Actions**" → "**Create Resource**"
   - Resource name: `orders`
   - Click "**Create Resource**"
   - Select `/orders` resource
   - Click "**Actions**" → "**Create Method**"

- Select `POST` → click checkmark

4. **Configure Integration**
   - Integration type: `AWS Service`
   - AWS Region: `us-east-1`
   - AWS Service: `Step Functions`
   - HTTP method: `POST`
   - Action Type: `Use path override`
   - Path override: `/stateMachines/[YOUR-STATE-MACHINE-ARN]/executions`
   - Replace with your state machine ARN
5. **Deploy API**
   - Click **"Actions"** → **"Deploy API"**
   - Deployment stage: `[New Stage]`
   - Stage name: `prod`
   - Click **"Deploy"**

6.5 Feature 4: X-Ray Tracing

6.5.1 Enable X-Ray for Lambda

1. **Open Lambda Console**
   - Select a Lambda function
   - Go to **"Configuration"** tab
   - Click **"Monitoring and operations tools"**
2. **Enable X-Ray**
   - Check **"Enable active tracing"**
   - Click **"Save"**
3. **Repeat for All Functions**
   - Enable for all three Lambda functions

6.5.2 Enable X-Ray for Step Functions

1. **Open Step Functions Console**
   - Select your state machine
   - Click **"Edit"**

- Go to "**Logging**" section

2. **Configure Tracing**
   - Check "**Enable X-Ray tracing**"
   - Click "**Save**"

6.5.3 View X-Ray Traces

1. **Open X-Ray Console**
   - Search "X-Ray" in services
   - Or go to: X-Ray Console
2. **View Service Map**
   - You'll see a visual map of your application
   - Shows connections between services
   - Displays latency and error information

6.6 Feature 5: Environment Variables

6.6.1 Configure Lambda Environment Variables

1. **Open Lambda Configuration**
   - Select a Lambda function
   - **Configuration** → "**Environment variables**"
   - Click "**Edit**"
2. **Add Variables**
   - Add these variables:
     - `ENVIRONMENT: development`
     - `LOG_LEVEL: INFO`
     - `S3_BUCKET:` Your bucket name
     - `MAX_RETRIES: 3`
   - Click "**Save**"
3. **Update Lambda Code**
   - Modify code to read environment variables:

```python
import os

# Read environment variables
environment = os.environ.get('ENVIRONMENT', 'development')
log_level = os.environ.get('LOG_LEVEL', 'INFO')
s3_bucket = os.environ.get('S3_BUCKET', 'default-bucket')
max_retries = int(os.environ.get('MAX_RETRIES', 3))

# Use in your code
print(f"Environment: {environment}")
print(f"Log level: {log_level}")
print(f"S3 Bucket: {s3_bucket}")
```

Listing 19: Reading Environment Variables

6.7 Advanced Features Completion Checklist

Feature	Implemented
SNS Email Notifications	
DynamoDB Storage	
API Gateway REST API	
X-Ray Tracing	
Environment Variables	
Updated Step Functions Workflow	
Tested All New Features	
Screenshots of New Services	

Table 11: Advanced Features Checklist

6.8 Cost Considerations for Advanced Features

Service	Free Tier Limit	Cost if Exceeded
SNS	1,000 emails/month	$0.50 per 100,000
DynamoDB	25 GB storage	$0.25 per GB-month
API Gateway	1M API calls/month	$3.50 per million
X-Ray	First 100,000 traces free	$5.00 per 1M traces

Table 12: Cost of Advanced Features

Warning

Cost Warning: Advanced features may incur costs if heavily used. Monitor your usage and clean up resources promptly.

6.9 Phase 6 Summary

You have optionally implemented:

- Email notifications with SNS
- Database storage with DynamoDB
- REST API with API Gateway
- Distributed tracing with X-Ray
- Configuration with environment variables

These features demonstrate real-world serverless application patterns and provide valuable experience with additional AWS services.

Tip

Learning Path: Each advanced feature corresponds to a common production requirement. Understanding these patterns will help you design robust serverless applications.

7 Phase 7: Cleanup & Documentation

Note

Time Estimate: 30-45 minutes
Completion Criteria: All resources deleted, documentation completed, project submitted

7.1 Importance of Cleanup

Warning

Critical: AWS resources continue to incur costs until deleted. Even "free tier" resources can become chargeable if limits are exceeded or if the free tier period expires.

7.2 Step 1: Create Cleanup Plan

7.2.1 Cleanup Order

Delete resources in this order to avoid dependency issues:

1. API Gateway (if created in Phase 6)
2. Step Functions State Machines
3. Lambda Functions
4. CloudWatch Alarms and Dashboards
5. S3 Buckets
6. DynamoDB Tables (if created)
7. SNS Topics (if created)
8. IAM Roles (created by services)
9. CloudWatch Log Groups

7.3 Step 2: Delete Step Functions Resources

7.3.1 Delete State Machine

1. **Open Step Functions Console**
   - Navigate to your state machine
2. **Delete State Machine**
   - Click **"Delete"**
   - Type state machine name to confirm
   - Click **"Delete"**
3. **Take Screenshot**
   - Capture confirmation message
   - Save as `12-stepfunctions-deleted.png`

7.4 Step 3: Delete Lambda Functions

7.4.1 Bulk Delete Functions

1. **Open Lambda Console**
   - Click **"Functions"** in left sidebar
2. **Select All Functions**
   - Check boxes next to all your functions:
     - validate-order
     - process-payment
     - save-to-s3
     - send-notification (if created)
     - save-to-dynamodb (if created)
3. **Delete Functions**
   - Click **"Actions"** → **"Delete"**
   - Type "delete" to confirm
   - Click **"Delete"**
4. **Verify Deletion**
   - Refresh page
   - Functions should not appear in list
5. **Take Screenshot**
   - Capture empty functions list
   - Save as `13-lambda-deleted.png`

7.5 Step 4: Delete S3 Bucket

7.5.1 Empty and Delete Bucket

1. **Open S3 Console**
   - Select your bucket
2. **Empty Bucket**
   - Click **"Empty"**
   - Type bucket name to confirm
   - Click **"Empty"**
3. **Delete Bucket**
   - Click **"Delete"**
   - Type bucket name to confirm
   - Click **"Delete"**
4. **Take Screenshot**
   - Capture empty buckets list
   - Save as `14-s3-deleted.png`

7.6 Step 5: Delete CloudWatch Resources

7.6.1 Delete Dashboard

1. **Open CloudWatch Console**
   - **Dashboards** → Select your dashboard
2. **Delete Dashboard**
   - Click **"Actions"** → **"Delete dashboard"**
   - Confirm deletion

7.6.2 Delete Alarms

1. **Navigate to Alarms**
   - **Alarms** → **"All Alarms"**
2. **Select and Delete**

- Select all alarms you created
- Click **"Actions"** → **"Delete"**
- Confirm deletion

7.6.3 Delete Log Groups

1. **Navigate to Log Groups**
   - **Logs"** → **"Log groups"**
2. **Delete Lambda Log Groups**
   - Search for `/aws/lambda/`
   - Select all your Lambda log groups
   - Click **"Actions"** → **"Delete"**
   - Confirm deletion
3. **Delete Step Functions Log Groups**
   - Search for `/aws/states/`
   - Delete Step Functions log groups

7.7 Step 6: Delete Advanced Resources (if created)

7.7.1 Delete DynamoDB Table

1. **Open DynamoDB Console**
2. Select `Orders` table
3. Click **"Delete table"**
4. Confirm deletion

7.7.2 Delete SNS Topic

1. **Open SNS Console**
2. Select your topic
3. Click **"Delete"**
4. Confirm deletion

7.7.3 Delete API Gateway

1. **Open API Gateway Console**
2. Select your API
3. Click **"Actions"** → **"Delete"**
4. Confirm deletion

7.8 Step 7: Delete IAM Roles

7.8.1 Find and Delete Roles

1. **Open IAM Console**
   - **Roles"** in left sidebar
2. **Search for Roles**
   - Search for roles containing:
     - lambda
     - states
     - OrderProcessing
3. **Delete Roles**
   - Select each role
   - Click **"Delete role"**
   - Confirm deletion

Warning

Important: Do NOT delete your `student-dev-user` IAM user. You may need it for future projects.

7.9 Step 8: Final Verification

7.9.1 Check All Services

Service	Status (Should be Empty/Zero)
Step Functions State Machines	0
Lambda Functions	0
S3 Buckets (project-specific)	0
CloudWatch Dashboards (project-specific)	0
CloudWatch Alarms (project-specific)	0
DynamoDB Tables (if created)	0
SNS Topics (if created)	0
API Gateway APIs (if created)	0
IAM Roles (created by services)	0

Table 13: Final Verification Checklist

7.9.2 Check Billing Dashboard

1. **Open Billing Dashboard**
2. Check **"Cost Explorer"**
3. Verify no ongoing charges
4. Take screenshot: `15-billing-zero.png`

7.10 Step 9: Complete Project Documentation

7.10.1 Final Project Report

Create `final-report.md`:

```
# AWS Serverless Project - Final Report

## Student Information
- Name: [Your Name]
- Student ID: [Your ID]
- Date: [Completion Date]
- Course: [Course Name]

## Project Summary
Brief description of what was accomplished in this project.

## Services Used
- [ ] AWS IAM (Identity and Access Management)
- [ ] Amazon S3 (Simple Storage Service)
- [ ] AWS Lambda
- [ ] AWS Step Functions
- [ ] Amazon CloudWatch
```

```
- [ ] (Optional) Amazon SNS
- [ ] (Optional) Amazon DynamoDB
- [ ] (Optional) Amazon API Gateway
- [ ] (Optional) AWS X-Ray

## Key Learnings
1. [What you learned about serverless architecture]
2. [What you learned about AWS services]
3. [Challenges faced and how you overcame them]
4. [Best practices discovered]

## Code Quality Assessment
- Code organization: [Excellent/Good/Fair/Poor]
- Error handling: [Excellent/Good/Fair/Poor]
- Documentation: [Excellent/Good/Fair/Poor]
- Testing completeness: [Excellent/Good/Fair/Poor]

## Architecture Diagram
[Describe or include your architecture diagram]

## Performance Observations
- Average Lambda execution time: [Time]
- Step Functions workflow duration: [Time]
- Most time-consuming step: [Which step and why]

## Cost Analysis
- Estimated monthly cost at current usage: [$ amount]
- Free tier utilization: [Percentage]
- Cost optimization opportunities: [Suggestions]

## Security Considerations
- IAM best practices implemented: [List]
- Data encryption: [Yes/No]
- Public access restrictions: [Yes/No]
- Additional security measures taken: [List]

## Recommendations for Improvement
1. [Technical improvements]
2. [Architectural improvements]
3. [Cost optimization suggestions]
4. [Security enhancements]

## Conclusion
[Overall assessment of the project and personal growth]

## Screenshots Included
1. [Screenshot 1 description]
2. [Screenshot 2 description]
3. [Screenshot 3 description]
... [Continue list]

## Appendix
- Lambda function code: [Attached]
- Test cases: [Attached]
- Configuration files: [Attached]
```

Listing 20: Final Project Report Template

7.10.2 Create Submission Package

Organize your files in this structure:

```
Project-Submission-[Your-Name]/

          Documentation/
                final-report.md
                test-report.txt
                architecture-diagram.png
                learning-journal.md

          Screenshots/
                01-billing-preferences.png
                02-iam-user-created.png
                03-region-selected.png
                04-s3-bucket-list.png
                05-s3-folders.png
                06-lambda-validate-created.png
                07-lambda-payment-created.png
                08-lambda-s3-created.png
                09-stepfunctions-created.png
                10-stepfunctions-success.png
                11-stepfunctions-failure.png
                12-stepfunctions-deleted.png
                13-lambda-deleted.png
                14-s3-deleted.png
                15-billing-zero.png

          Code/
                validate-order.py
                process-payment.py
                save-to-s3.py
                send-notification.py (optional)
                save-to-dynamodb.py (optional)

          Configuration/
              bucket-info.txt
              lambda-info.txt
              stepfunctions-info.txt
              test-events.json
```

Listing 21: Submission Package Structure

7.11 Step 10: Submit Project

7.11.1 Submission Requirements

1. **Compress Files**
   - Create ZIP file of your submission package
   - Name: `AWS-Project-[Your-Name]-[Date].zip`

2. **Submit Through Learning Platform**
    - Upload ZIP file
    - Include any required forms
    - Meet submission deadline
3. **Keep Local Backup**
    - Keep a copy of all files locally
    - Useful for portfolio or future reference

7.12 Final Phase Checklist

Task	Completed
All AWS resources deleted	
Billing verified as $0.00	
Final report completed	
All screenshots collected and organized	
Code files saved and documented	
Submission package created	
Project submitted on time	
Local backup maintained	

Table 14: Final Phase Completion

7.13 Troubleshooting Cleanup Issues

1. **"Cannot delete bucket - not empty"**
    - Ensure you emptied the bucket first
    - Check for versioned objects
    - Use S3 console "Empty bucket" feature
2. **"Cannot delete IAM role - in use"**
    - Wait 5-10 minutes after deleting services
    - Check if any service still references the role
    - Try deleting again later
3. **"Resources reappear"**
    - Check if you have automation recreating resources
    - Verify you're in correct region
    - Clear browser cache and check again

7.14 Phase 7 Summary

You have successfully:

- Safely deleted all AWS resources
- Verified no ongoing charges
- Created comprehensive documentation
- Organized project deliverables
- Prepared professional submission package
- Demonstrated responsible cloud resource management

Tip

Career Tip: This project makes an excellent portfolio piece. Consider creating a GitHub repository with your code and documentation to showcase to potential employers.

7.15 Final Reflection Questions

Consider these questions as you complete your project:

1. What was the most challenging aspect of this project?
2. How would you scale this system for production use?
3. What additional AWS services would you integrate next?
4. How has this project changed your understanding of serverless architecture?
5. What would you do differently if starting over?

7.16 Congratulations!

You have successfully completed the AWS Serverless Order Processing System project. You have gained practical experience with:

- AWS account management and security
- Serverless compute with AWS Lambda
- Workflow orchestration with Step Functions

- Cloud storage with S3
- Monitoring with CloudWatch
- Integration of multiple AWS services
- Cost management and resource cleanup

These skills are highly valuable in today's cloud computing landscape and will serve you well in your future cloud endeavors.

Appendix

A. AWS Free Tier Details

Service	Free Tier Limit	Duration
AWS Lambda	1M requests/month	12 months
Step Functions	4,000 state transitions/-month	12 months
Amazon S3	5GB storage	12 months
CloudWatch	10 custom metrics	12 months
SNS	1,000 publishes/month	12 months
DynamoDB	25GB storage	Always free
API Gateway	1M API calls/month	12 months

Table 15: AWS Free Tier Limits (us-east-1)

B. Common Error Messages and Solutions

Error Message	Solution
"User: arn:aws:... is not authorized to perform:..."	Check IAM permissions, attach required policies
"The specified bucket does not exist"	Verify bucket name spelling and region
"Function not found"	Check Lambda function exists in same region
"Timeout after 3 seconds"	Increase Lambda timeout in configuration
"Invalid JSON"	Validate JSON syntax, check quotes and brackets
"Resource already exists"	Choose different name or delete existing resource
"Access Denied"	Check resource policies and IAM permissions

Table 16: Common AWS Errors and Solutions

C. AWS Console URLs

- **AWS Management Console:** `https://console.aws.amazon.com/`
- **IAM Console:** `https://console.aws.amazon.com/iam/`
- **Lambda Console:** `https://console.aws.amazon.com/lambda/`
- **Step Functions Console:** `https://console.aws.amazon.com/states/`
- **S3 Console:** `https://s3.console.aws.amazon.com/`
- **CloudWatch Console:** `https://console.aws.amazon.com/cloudwatch/`

- **Billing Console:** https://console.aws.amazon.com/billing/
- **SNS Console:** https://console.aws.amazon.com/sns/
- **DynamoDB Console:** https://console.aws.amazon.com/dynamodb/
- **API Gateway Console:** https://console.aws.amazon.com/apigateway/
- **X-Ray Console:** https://console.aws.amazon.com/xray/

D. Python Code Examples

Complete validate-order.py

```
import json
import boto3
from datetime import datetime
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    """
    Validates incoming order data
    Returns: Validated order or error message
    """

    logger.info(f"Received event: {json.dumps(event)}")

    # Check if event comes from Step Functions
    if 'body' in event:
        try:
            event = json.loads(event['body'])
        except:
            pass

    # Required fields for order validation
    required_fields = [
        'order_id',
        'customer_id',
        'items',
        'total_amount'
    ]

    # Check for missing fields
    missing_fields = []
    for field in required_fields:
        if field not in event:
            missing_fields.append(field)

    if missing_fields:
        error_message = f"Missing required fields: {missing_fields}"
        logger.error(error_message)
        return {
```

```
            'statusCode': 400,
            'error': error_message,
            'missing_fields': missing_fields
        }

    # Validate items array
    if not isinstance(event['items'], list) or len(event['items']) ==
        0:
        return {
            'statusCode': 400,
            'error': 'Items must be a non-empty array'
        }

    # Validate total amount
    try:
        total = float(event['total_amount'])
        if total <= 0:
            raise ValueError("Total must be positive")
    except ValueError as e:
        return {
            'statusCode': 400,
            'error': f'Invalid total_amount: {str(e)}'
        }

    # Add metadata
    event['validation_timestamp'] = datetime.now().isoformat()
    event['status'] = 'VALIDATED'
    event['validation_id'] = context.aws_request_id

    logger.info(f"Order validated: {event['order_id']}")

    # Return format suitable for Step Functions
    return {
        'statusCode': 200,
        'order': event,
        'message': 'Order validated successfully'
    }
```

Listing 22: Complete validate-order.py

Complete process-payment.py

```
import json
import random
from datetime import datetime
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    """
    Simulates payment processing
    Returns: Updated order with payment status
    """

```

```
    logger.info(f"Processing payment for order")

    # Handle input from Step Functions or direct invocation
    if 'order' in event:
        order_data = event['order']
    elif 'body' in event:
        try:
            order_data = json.loads(event['body'])
        except:
            order_data = event
    else:
        order_data = event

    # Log the order ID if available
    order_id = order_data.get('order_id', 'UNKNOWN')
    logger.info(f"Processing payment for order: {order_id}")

    # Simulate payment processing with 85% success rate
    payment_statuses = ['SUCCESS', 'FAILED']
    weights = [0.85, 0.15]  # 85% success, 15% failure

    chosen_status = random.choices(
        payment_statuses,
        weights=weights,
        k=1
    )[0]

    # Add payment information
    order_data['payment_status'] = chosen_status
    order_data['payment_timestamp'] = datetime.now().isoformat()
    order_data['transaction_id'] = f"TXN-{random.randint(10000, 99999)}
        "

    # Update overall status
    if chosen_status == 'SUCCESS':
        order_data['status'] = 'PAYMENT_SUCCESS'
        logger.info(f"Payment successful for order: {order_id}")
    else:
        order_data['status'] = 'PAYMENT_FAILED'
        order_data['failure_reason'] = 'Insufficient funds or card
            declined'
        logger.warning(f"Payment failed for order: {order_id}")

    # Return format for Step Functions
    return {
        'statusCode': 200,
        'order': order_data,
        'message': f'Payment processing completed: {chosen_status}'
    }
```

Listing 23: Complete process-payment.py

E. Sample Test Events

Valid Order Test Event

```
{
  "order_id": "ORD-001",
  "customer_id": "CUST-001",
  "items": [
    {
      "product_id": "P-001",
      "name": "Wireless Headphones",
      "quantity": 1,
      "price": 99.99
    },
    {
      "product_id": "P-002",
      "name": "USB-C Cable",
      "quantity": 2,
      "price": 12.50
    }
  ],
  "total_amount": 124.99,
  "customer_email": "customer@example.com",
  "shipping_address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

Listing 24: Valid Order Test Event

Invalid Order Test Event

```
{
  "order_id": "ORD-002",
  "customer_id": "CUST-002",
  "items": [],  # Empty array - should fail
  "total_amount": 0  # Zero amount - should fail
}
```

Listing 25: Invalid Order Test Event