# AWS Lambda, SNS, and SQS: Architecture Deep Dive

## Introduction

In modern cloud architectures, event-driven and decoupled systems provide flexibility, scalability, and resilience. AWS offers three key services that, when combined, form a powerful backbone for such architectures:

* **AWS Lambda** - Serverless compute service
* **Amazon SNS (Simple Notification Service)** - Pub/sub messaging service
* **Amazon SQS (Simple Queue Service)** - Fully managed message queuing service

This document explains each service in detail and illustrates how they can be integrated to create robust, event-driven applications.

## AWS Lambda

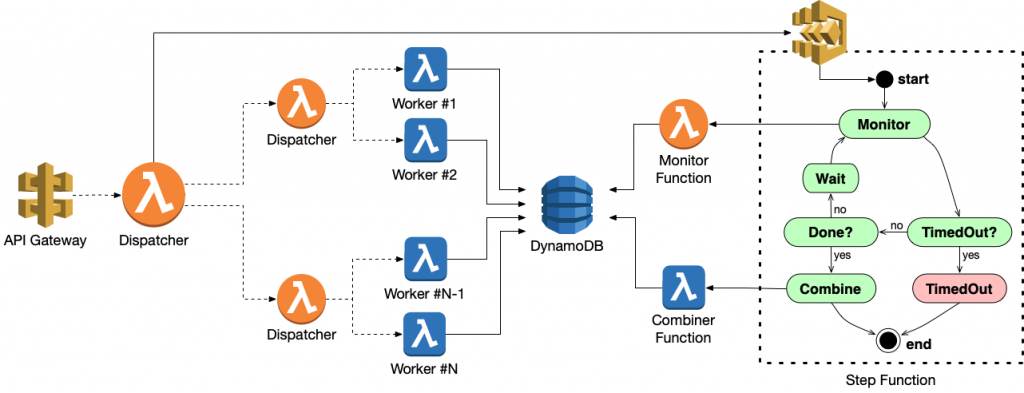
### What is AWS Lambda?

AWS Lambda is a serverless compute service that runs your code in response to events without requiring you to provision or manage servers. Lambda automatically scales your application by running code in response to each trigger.

### Key Features

1. **Event-Driven Execution**: Lambda functions execute in response to events from various AWS services (S3, DynamoDB, API Gateway, etc.)
2. **Serverless**: No server management required
3. **Automatic Scaling**: Lambda automatically scales to handle concurrent requests
4. **Pay-Per-Use**: You pay only for the compute time your code consumes
5. **Multiple Language Support**: Supports Node.js, Python, Java, Go, Ruby, .NET, and custom runtimes
6. **Stateless**: Functions are stateless, allowing for easy scaling
7. **Timeouts**: Maximum execution time of 15 minutes

### Lambda Architecture



Lambda Architecture

A Lambda function consists of:

1. **Handler Function**: Entry point where execution begins
2. **Runtime Environment**: Isolated container with the configured memory and CPU
3. **Event Object**: Data passed to the function when invoked
4. **Context Object**: Methods available to interact with the runtime
5. **Environment Variables**: Key-value pairs accessible within the function

### Lambda Lifecycle

1. **Deployment**: You upload your code and configurations
2. **Initialization**: When first invoked, Lambda initializes a container
3. **Invocation**: Function is called with an event object
4. **Execution**: Your code runs and processes the event
5. **Termination**: Container might be reused or terminated depending on future invocations

### Lambda Invocation Types

1. **Synchronous**: Caller waits for the response
2. **Asynchronous**: Lambda queues the event for processing
3. **Poll-Based**: Lambda polls a stream or queue (Kinesis, DynamoDB Streams, SQS)

## Amazon SNS (Simple Notification Service)

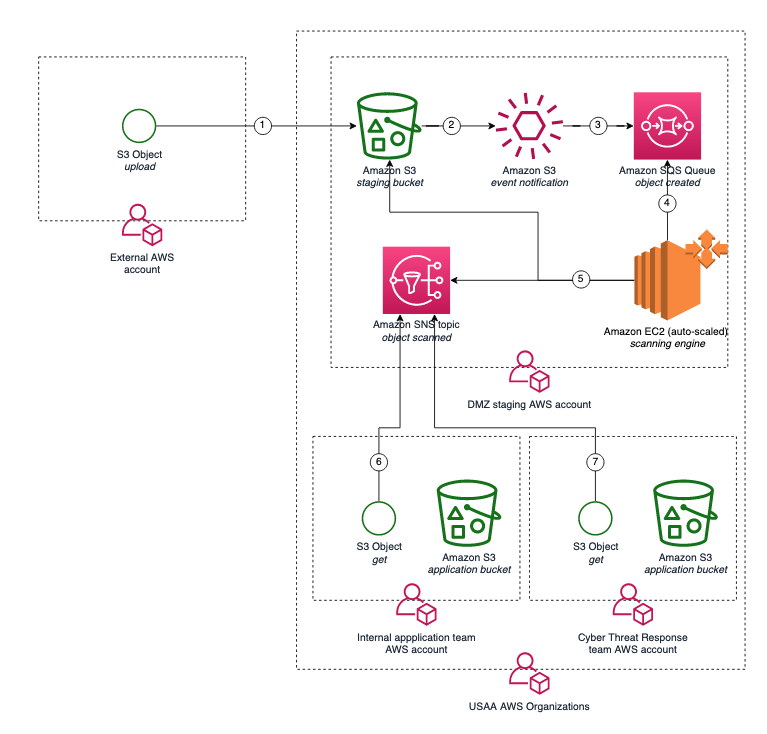
### What is Amazon SNS?

Amazon SNS is a fully managed messaging service for both application-to-application (A2A) and application-to-person (A2P) communication. It decouples message publishers from subscribers.

### Key Features

1. **Pub/Sub Model**: Publishers send messages to topics, and subscribers receive them
2. **Fan-Out Pattern**: One message can be delivered to multiple subscribers
3. **Message Filtering**: Subscribers can filter messages based on attributes
4. **Delivery Protocols**: HTTP/S, Email, SMS, SQS, Lambda, and mobile push notifications
5. **Message Attributes**: Metadata for message filtering
6. **Durability**: Messages are stored redundantly across multiple AZs
7. **Large Message Size**: Up to 256KB per message

### SNS Architecture



SNS Architecture

SNS consists of:

1. **Topics**: Named access points for publishers to send messages
2. **Subscriptions**: Endpoints that receive messages from a topic
3. **Publishers**: Entities that send messages to topics
4. **Message Format**: JSON with message body and attributes

### SNS Message Flow

1. **Publish**: Publisher sends a message to a topic
2. **Fanout**: SNS duplicates the message for each subscription
3. **Delivery**: SNS attempts to deliver the message to each endpoint
4. **Retry Policy**: Failed deliveries are retried based on configured policies
5. **Dead-Letter Queue**: Failed deliveries can be sent to a DLQ after retry policy is exhausted

## Amazon SQS (Simple Queue Service)

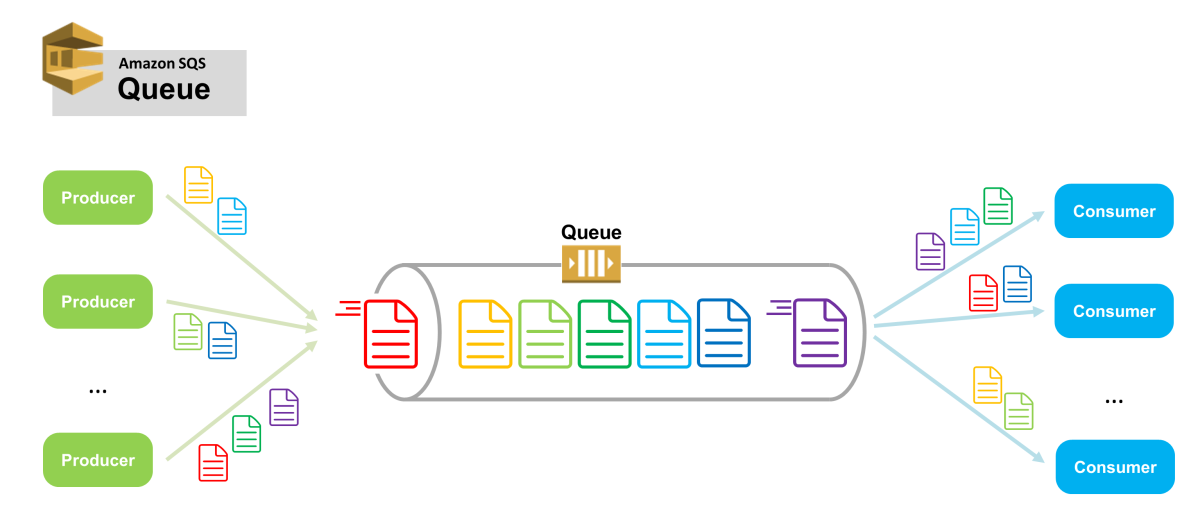
### What is Amazon SQS?

Amazon SQS is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications.

### Key Features

1. **Reliable Delivery**: Messages are stored redundantly across multiple AZs
2. **Queue Types**: Standard (at-least-once delivery, best-effort ordering) and FIFO (exactly-once processing, strict ordering)
3. **Message Retention**: Messages can be retained for up to 14 days
4. **Long Polling**: Reduces empty responses and API calls
5. **Visibility Timeout**: Prevents multiple consumers from processing the same message
6. **Dead-Letter Queues**: Captures messages that can’t be processed
7. **Large Message Size**: Up to 256KB per message (larger via S3 pointer)

### SQS Architecture



SQS Architecture

SQS consists of:

1. **Queue**: The container for messages
2. **Producers**: Applications that send messages to the queue
3. **Consumers**: Applications that process messages from the queue
4. **Message Lifecycle**: Send, Store, Receive, Delete

### SQS Message Flow

1. **Send**: Producer sends message to queue
2. **Store**: Message is stored redundantly across multiple AZs
3. **Receive**: Consumer polls queue and receives message
4. **Process**: Consumer processes the message
5. **Delete**: Consumer explicitly deletes the message after successful processing
6. **Visibility Timeout**: If not deleted, message becomes visible again after timeout

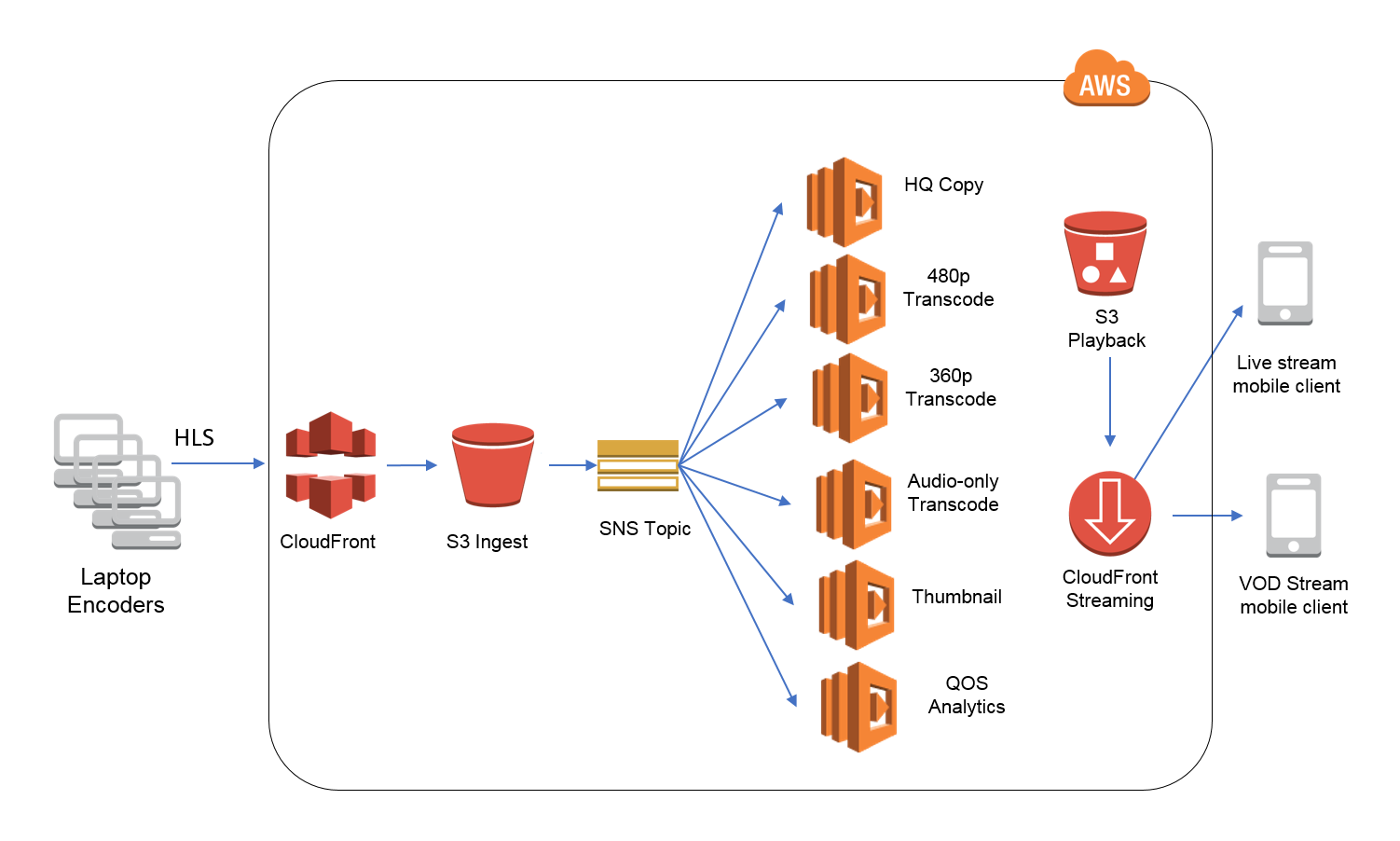
### Standard vs. FIFO Queues

| Feature | Standard Queue | FIFO Queue |
| --- | --- | --- |
| Delivery | At least once | Exactly once |
| Ordering | Best-effort | Strict (first-in-first-out) |
| Throughput | Unlimited | 300 TPS (3000 with batching) |
| Deduplication | None built-in | Supported |
| Use Case | High throughput, order not critical | Order critical, no duplicates |

## Integrating Lambda, SNS, and SQS

### Common Integration Patterns

#### 1. Lambda → SNS (Fan-out pattern)

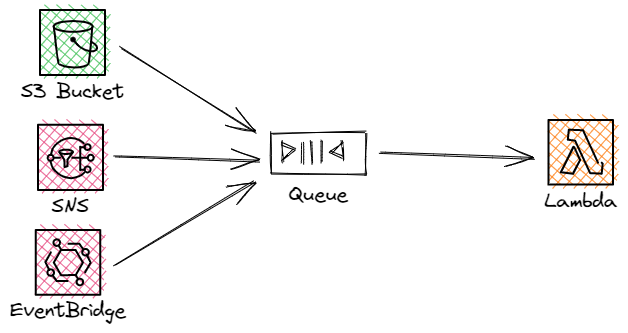


Lambda to SNS

**Flow**: 1. Lambda processes an event 2. Lambda publishes message to SNS topic 3. SNS delivers to multiple subscribers

**Benefits**: - One Lambda execution can trigger multiple downstream processes - Decoupled architecture - Easy to add new subscribers without changing Lambda code

#### 2. SNS → Lambda (Direct invocation)



SNS to Lambda

**Flow**: 1. Publisher sends message to SNS topic 2. SNS invokes Lambda function with message payload 3. Lambda processes the message

**Benefits**: - Immediate processing - Simple architecture - Good for non-critical processing

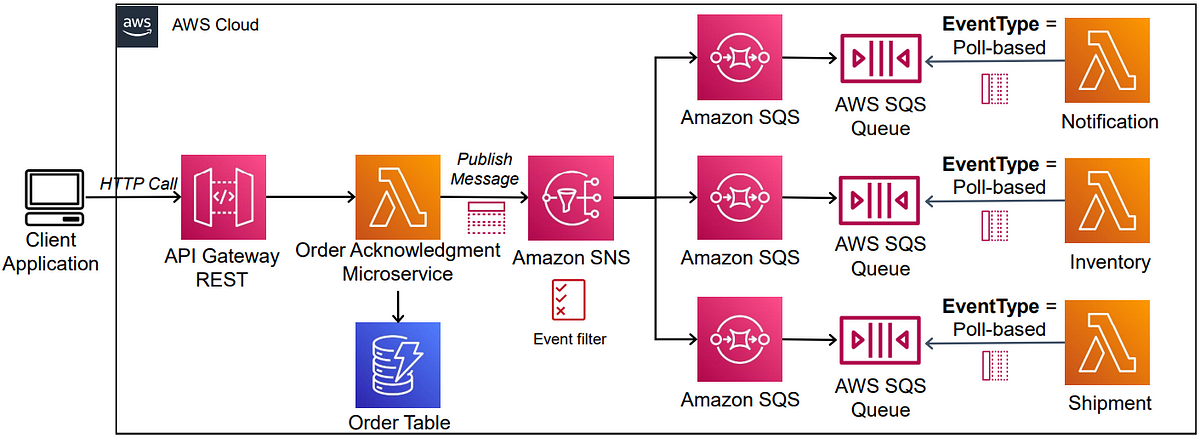
#### 3. SNS → SQS → Lambda (Durable processing)

SNS to SQS to Lambda

**Flow**: 1. Publisher sends message to SNS topic 2. SNS delivers message to SQS queue(s) 3. Lambda polls SQS queue for messages 4. Lambda processes messages and deletes from queue

**Benefits**: - Durability: Messages persist in SQS if Lambda fails - Throttling: SQS buffers messages if Lambda can’t keep up - Retry: Failed processing can be retried - DLQ: Persistently failed messages can be moved to DLQ

#### 4. API Gateway → Lambda → SNS → SQS → Lambda (Full event processing)



Full Architecture

**Flow**: 1. Client sends request to API Gateway 2. API Gateway triggers Lambda 3. Lambda processes request and publishes to SNS 4. SNS delivers to multiple SQS queues 5. Lambda functions poll respective SQS queues 6. Each Lambda performs specific processing

**Benefits**: - Complete decoupling - Independent scaling - Fault isolation - Specialized processing

### Best Practices for Lambda, SNS, and SQS Integration

#### Lambda Best Practices

1. **Keep Functions Focused**: Each function should have a single responsibility
2. **Manage Timeouts**: Set timeout values appropriate for the task
3. **Handle Partial Failures**: Implement idempotent processing for retries
4. **Size Appropriately**: Allocate memory based on performance needs
5. **Use Environment Variables**: Store configuration outside code
6. **Implement Error Handling**: Capture and process errors appropriately

#### SNS Best Practices

1. **Topic Design**: Create topics around business events, not consumers
2. **Message Attributes**: Use attributes for filtering instead of creating multiple topics
3. **Message Format**: Use a consistent JSON schema within messages
4. **Error Handling**: Configure DLQs for failed deliveries
5. **Monitoring**: Set up CloudWatch alarms for delivery failures

#### SQS Best Practices

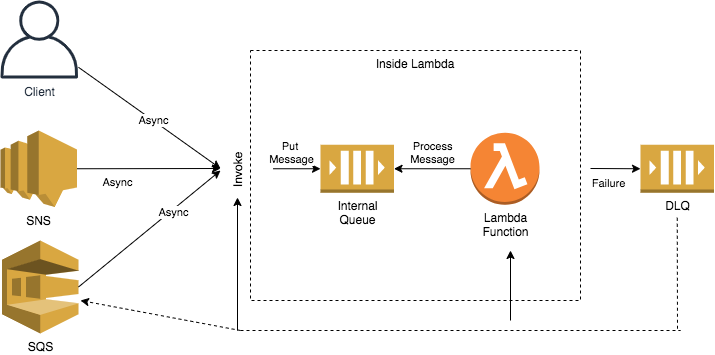
1. **Queue Design**: Use standard queues for high throughput, FIFO when order matters
2. **Message Size**: Keep messages small or use S3 for large payloads
3. **Visibility Timeout**: Set based on expected processing time
4. **Batch Processing**: Use batch operations when possible
5. **Long Polling**: Enable to reduce empty responses
6. **DLQ Configuration**: Set up DLQs with appropriate retry thresholds

## Advanced Considerations

### Handling Large Messages

For messages exceeding the 256KB limit: 1. Store the payload in S3 2. Pass the S3 object reference in SNS/SQS 3. Lambda retrieves the full payload from S3

### Error Handling and Dead-Letter Queues



DLQ Architecture

**Implementation**: 1. Configure DLQs for both SNS subscriptions and SQS queues 2. Set appropriate retry policies 3. Create a separate Lambda function to process DLQ messages 4. Implement alerting on DLQ message arrival

### Security Considerations

1. **IAM Policies**: Use least privilege principle
2. **Encryption**: Enable encryption at rest and in transit
3. **VPC Integration**: Run Lambda in VPC when accessing private resources
4. **Resource Policies**: Control who can invoke Lambda or publish to SNS

### Monitoring and Troubleshooting

1. **CloudWatch Metrics**: Monitor invocation counts, errors, and latency
2. **CloudWatch Logs**: Analyze Lambda execution logs
3. **X-Ray**: Trace requests through the distributed system
4. **CloudWatch Alarms**: Alert on abnormal conditions
5. **CloudTrail**: Audit API calls

### Cost Optimization

1. **Right-size Lambda**: Allocate appropriate memory
2. **Optimize Code**: Reduce execution time
3. **Batch Processing**: Reduce invocations by processing in batches
4. **Reserved Concurrency**: Control maximum concurrency for critical functions

## Common Use Cases

### Event-Driven Data Processing

Example: E-commerce order processing - Customer places order via API Gateway - Order Lambda validates and publishes to SNS - SNS fans out to multiple SQS queues - Specialized Lambdas process different aspects (inventory, payment, shipping)

### Microservices Communication

Example: User registration flow - Registration Lambda creates user - Lambda publishes “UserCreated” event to SNS - Multiple services subscribe: email verification, welcome sequence, analytics

### Asynchronous API Processing

Example: Document processing - User uploads document via API - Lambda acknowledges receipt and publishes to SNS - Processing Lambda consumes from SQS queue - When complete, notification sent to user

### Real-Time Data Transformation

Example: IoT data pipeline - IoT devices publish to MQTT topics in IoT Core - Rules Engine routes to SNS - Lambda functions process different data types - Processed data stored in data lake

## Conclusion

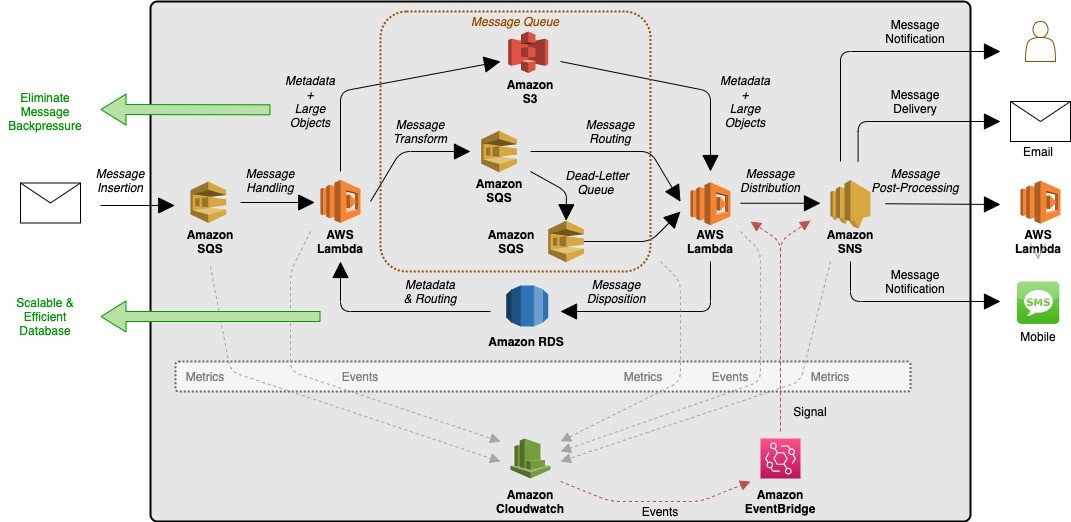
The combination of AWS Lambda, SNS, and SQS provides a powerful foundation for building serverless, event-driven architectures. By leveraging the strengths of each service and following established patterns and best practices, you can create systems that are:

* Highly scalable
* Fault-tolerant
* Cost-efficient
* Loosely coupled
* Easy to maintain and extend

This approach enables you to focus on business logic rather than infrastructure management, accelerating development and reducing operational overhead.

# 

# Example: Building an Order Processing System with AWS Services



I’ll walk you through creating a complete order processing system using AWS Lambda, SNS, and SQS as requested. This system will process orders from a React form through to separate order and billing inventory systems.

## Architecture Overview

Your architecture will follow this flow:

1. React Form → API Gateway

2. API Gateway → Lambda (Order Processor)

3. Lambda → SNS Topic

4. SNS Topic → Two SQS Queues (Order Queue & Billing Queue)

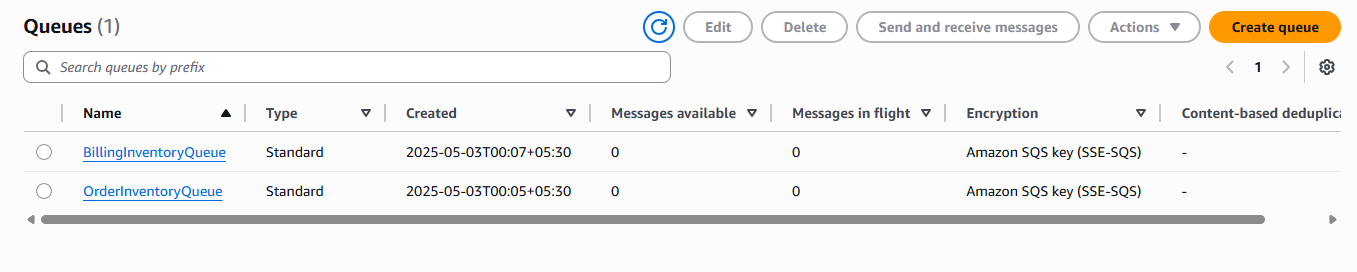
5. SQS Queues → Respective Lambda Functions (Order Inventory & Billing Inventory)

Let me break this down into detailed implementation steps:

## Step 1: Create SQS Queues

First, let’s set up the destination queues:

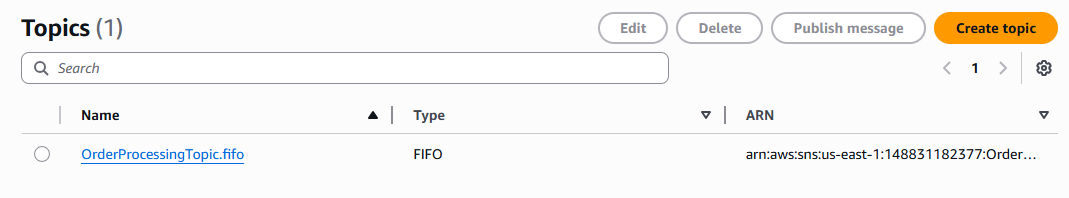
1. Log into AWS Management Console and search for “SQS”
2. Click “Create queue”
3. For the Order queue:
   * Select “Standard Queue”
   * Name it “OrderInventoryQueue”
   * Keep default settings for now
   * Click “Create queue”
4. Repeat for the Billing queue:
   * Select “Standard Queue”
   * Name it “BillingInventoryQueue”
   * Keep default settings
   * Click “Create queue”
5. Take screenshots of both created queues for your documentation



## Step 2: Create an SNS Topic

Next, create the SNS topic that will fan out messages to both queues:

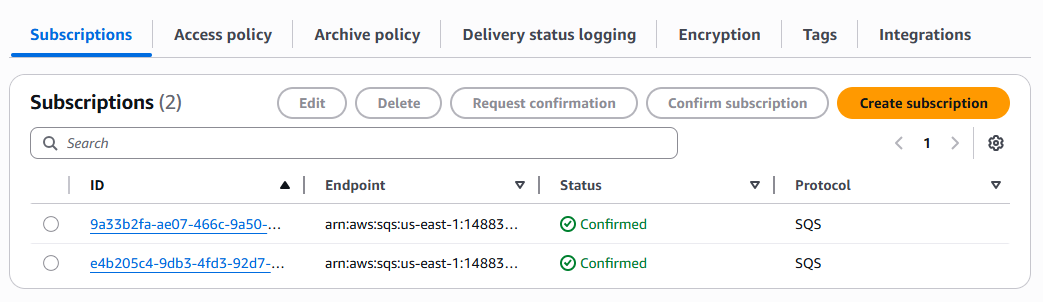
1. In the AWS Console, search for “SNS”
2. Click “Topics” in the left navigation
3. Click “Create topic”
4. Configure:
   * Type: Standard
   * Name: “OrderProcessingTopic”
   * Display name: “OrderProcessing”
5. Click “Create topic”
6. Take a screenshot of the created topic



## Step 3: Subscribe SQS Queues to SNS Topic

Now connect your SNS topic to both SQS queues:

1. Select your “OrderProcessingTopic”
2. Click “Create subscription”
3. For the first subscription:
   * Protocol: “Amazon SQS”
   * Endpoint: Select your “OrderInventoryQueue” from the dropdown (or paste ARN)
   * Click “Create subscription”
4. Repeat for the second subscription:
   * Protocol: “Amazon SQS”
   * Endpoint: Select your “BillingInventoryQueue”
   * Click “Create subscription”
5. Take screenshots showing both subscriptions active



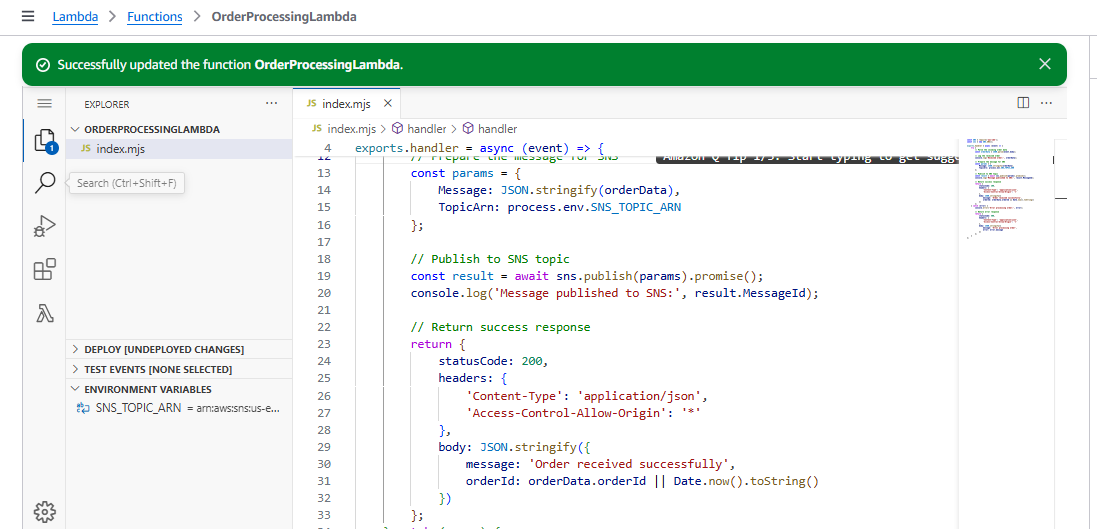
## Step 4: Create Order Processing Lambda

This Lambda will receive API Gateway requests and publish to SNS:

1. In AWS Console, search for “Lambda”
2. Click “Create function”
3. Select “Author from scratch”
4. Configure:
   * Function name: “OrderProcessingLambda”
   * Runtime: Node.js 18.x
   * Architecture: x86\_64
5. Click “Create function”
6. Replace the code with:

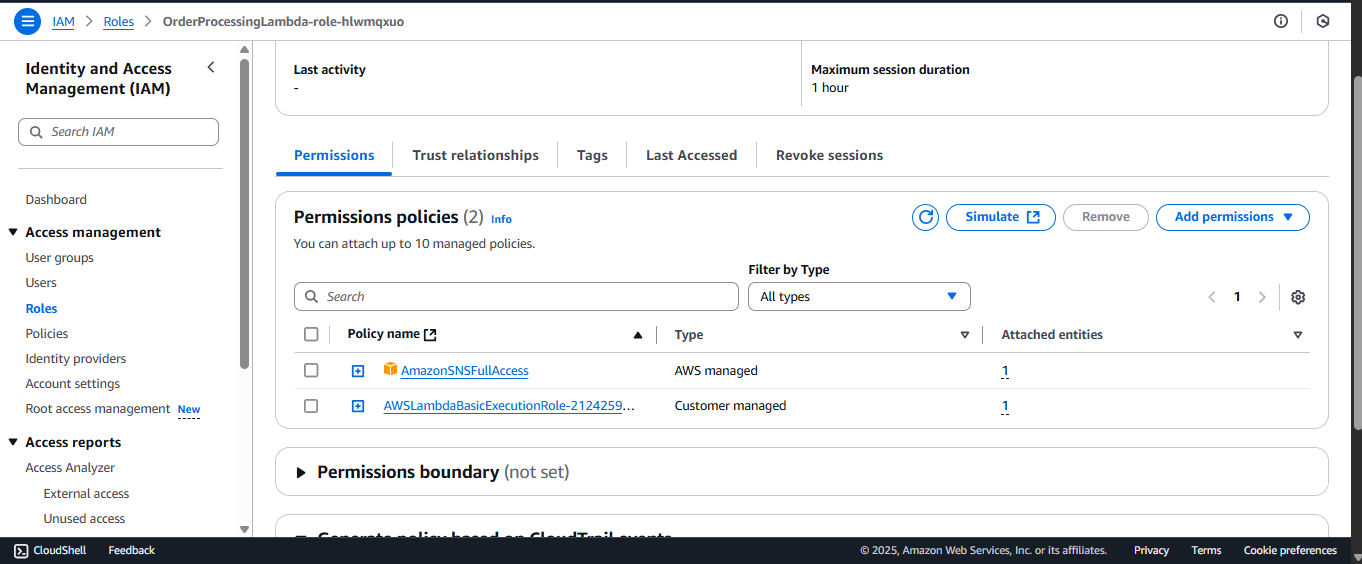
const AWS = require('aws-sdk');  
const sns = new AWS.SNS();  
  
exports.handler = async (event) => {  
 try {  
 // Parse the incoming order data  
 const orderData = JSON.parse(event.body);  
   
 // Log the received order  
 console.log('Received order:', orderData);  
   
 // Prepare the message for SNS  
 const params = {  
 Message: JSON.stringify(orderData),  
 TopicArn: process.env.SNS\_TOPIC\_ARN  
 };  
   
 // Publish to SNS topic  
 const result = await sns.publish(params).promise();  
 console.log('Message published to SNS:', result.MessageId);  
   
 // Return success response  
 return {  
 statusCode: 200,  
 headers: {  
 'Content-Type': 'application/json',  
 'Access-Control-Allow-Origin': '\*'  
 },  
 body: JSON.stringify({  
 message: 'Order received successfully',  
 orderId: orderData.orderId || Date.now().toString()  
 })  
 };  
 } catch (error) {  
 console.error('Error processing order:', error);  
   
 // Return error response  
 return {  
 statusCode: 500,  
 headers: {  
 'Content-Type': 'application/json',  
 'Access-Control-Allow-Origin': '\*'  
 },  
 body: JSON.stringify({  
 message: 'Error processing order',  
 error: error.message  
 })  
 };  
 }  
};

1. Add an environment variable:
   * Key: SNS\_TOPIC\_ARN
   * Value: [Your SNS Topic ARN] (Copy from the SNS topic you created)
2. Take a screenshot of the function code and configuration



## Step 5: Setup IAM Permissions for Order Lambda

1. In your Lambda function, go to the “Configuration” tab
2. Click on “Permissions”
3. Click on the IAM Role name
4. In the IAM console, click “Attach policies”
5. Search for and attach “AmazonSNSFullAccess” (in production, use more restricted permissions)
6. Take a screenshot of the permissions



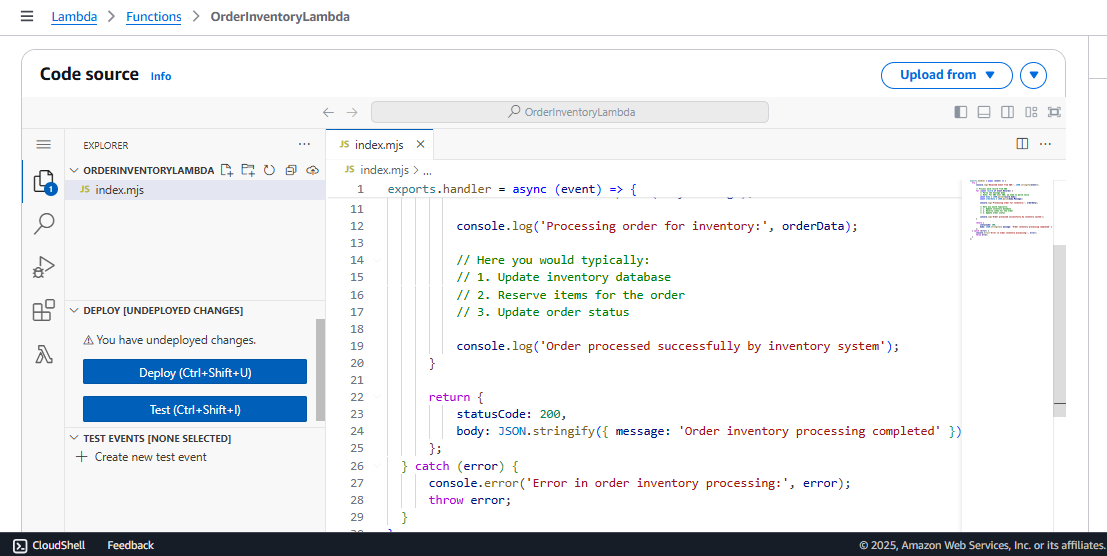
## Step 6: Create Order Inventory Lambda

This Lambda processes messages from the Order queue:

1. Create a new Lambda function:
   * Name: “OrderInventoryLambda”
   * Runtime: Node.js 18.x
2. Add this code:

exports.handler = async (event) => {  
 try {  
 console.log('Received event from SQS:', JSON.stringify(event));  
   
 // Process each record from SQS  
 for (const record of event.Records) {  
 // Parse the message body  
 // Note: For SNS over SQS, we need to parse twice  
 const body = JSON.parse(record.body);  
 const orderData = JSON.parse(body.Message);  
   
 console.log('Processing order for inventory:', orderData);  
   
 // Here you would typically:  
 // 1. Update inventory database  
 // 2. Reserve items for the order  
 // 3. Update order status  
   
 console.log('Order processed successfully by inventory system');  
 }  
   
 return {  
 statusCode: 200,  
 body: JSON.stringify({ message: 'Order inventory processing completed' })  
 };  
 } catch (error) {  
 console.error('Error in order inventory processing:', error);  
 throw error;  
 }  
};

1. Take a screenshot of the function



## Step 7: Create Billing Inventory Lambda

Similar to the Order Lambda, create one for billing:

1. Create another Lambda function:
   * Name: “BillingInventoryLambda”
   * Runtime: Node.js 18.x
2. Add this code:

exports.handler = async (event) => {  
 try {  
 console.log('Received event from SQS:', JSON.stringify(event));  
   
 // Process each record from SQS  
 for (const record of event.Records) {  
 // Parse the message body  
 // Note: For SNS over SQS, we need to parse twice  
 const body = JSON.parse(record.body);  
 const orderData = JSON.parse(body.Message);  
   
 console.log('Processing order for billing:', orderData);  
   
 // Here you would typically:  
 // 1. Calculate final price  
 // 2. Generate invoice  
 // 3. Record transaction in billing system  
   
 console.log('Order processed successfully by billing system');  
 }  
   
 return {  
 statusCode: 200,  
 body: JSON.stringify({ message: 'Billing processing completed' })  
 };  
 } catch (error) {  
 console.error('Error in billing processing:', error);  
 throw error;  
 }  
};

1. Take a screenshot of the function

## Step 8: Configure SQS Triggers for Inventory Lambdas

Note: This permission should be added AmazonSQSFullAccess for Lambda function to get triggered

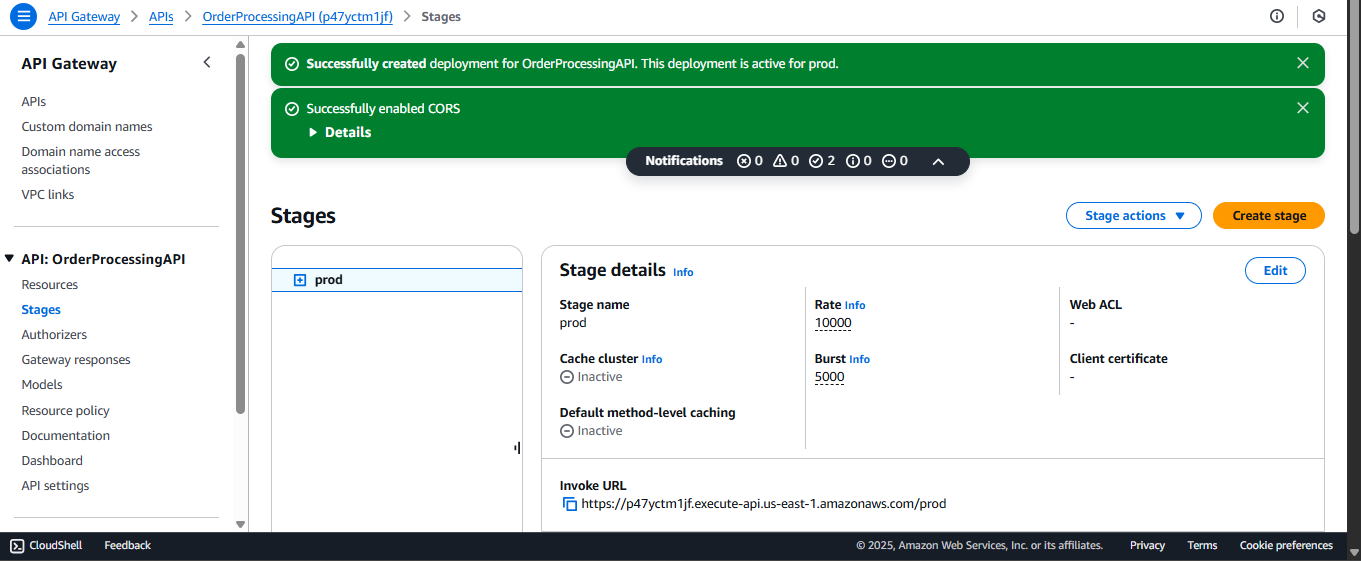
1. For the OrderInventoryLambda:
   * Go to “Configuration” tab
   * Click “Triggers” > “Add trigger”
   * Select “SQS” from the dropdown
   * Select your “OrderInventoryQueue”
   * Batch size: 10 (default)
   * Click “Add”
2. For the BillingInventoryLambda:
   * Go to “Configuration” tab
   * Click “Triggers” > “Add trigger”
   * Select “SQS” from the dropdown
   * Select your “BillingInventoryQueue”
   * Batch size: 10 (default)
   * Click “Add”
3. Take screenshots of both Lambda triggers



## Step 9: Create API Gateway

Create an API Gateway to receive requests from your React form:

1. In AWS Console, search for “API Gateway”
2. Click “Create API”
3. Select “REST API” and click “Build”
4. Configure:
   * API name: “OrderProcessingAPI”
   * Endpoint Type: Regional
5. Click “Create API”
6. Create a resource:
   * Click “Create Resource”
   * Resource Name: “orders”
   * Click “Create Resource”
7. Create a POST method:
   * With “orders” selected, click “Create Method”
   * Select “POST” from the dropdown
   * Click the checkmark
   * Integration type: Lambda Function
   * Lambda Function: OrderProcessingLambda
   * Click “Save”
8. Enable CORS:
   * Select the “orders” resource
   * Click “Actions” > “Enable CORS”
   * Check “Access-Control-Allow-Origin” and enter “\*”
   * Click “Enable CORS and replace existing CORS headers”
9. Deploy the API:
   * Click “Actions” > “Deploy API”
   * Deployment stage: [New Stage]
   * Stage name: “prod”
   * Click “Deploy”
10. Take note of the Invoke URL shown
11. Take screenshots of the API Gateway configuration



## Step 10: Create a Simple React Form

Now let’s create a basic React form to test the system:

import React, { useState } from 'react';

function OrderForm() {

  const [formData, setFormData] = useState({

    customerName: '',

    email: '',

    productId: '',

    quantity: 1,

    shippingAddress: ''

  });

  const [submitting, setSubmitting] = useState(false);

  const [result, setResult] = useState(null);

  const [error, setError] = useState(null);

  // Replace with your actual API Gateway URL

  const API\_URL = 'https://your-api-gateway-url.amazonaws.com/prod/orders';

  const handleChange = (e) => {

    const { name, value } = e.target;

    setFormData({

      ...formData,

      [name]: name === 'quantity' ? parseInt(value) : value

    });

  };

  const handleSubmit = async (e) => {

    e.preventDefault();

    setSubmitting(true);

    setError(null);

    try {

      const orderData = {

        ...formData,

        orderId: `ORD-${Date.now()}`,

        orderDate: new Date().toISOString()

      };

      const response = await fetch(API\_URL, {

        method: 'POST',

        headers: {

          'Content-Type': 'application/json'

        },

        body: JSON.stringify(orderData)

      });

      const data = await response.json();

      if (!response.ok) {

        throw new Error(data.message || 'Failed to submit order');

      }

      setResult(data);

      // Reset form

      setFormData({

        customerName: '',

        email: '',

        productId: '',

        quantity: 1,

        shippingAddress: ''

      });

    } catch (err) {

      setError(err.message || 'An error occurred while submitting your order');

    } finally {

      setSubmitting(false);

    }

  };

  return (

    <div>

      <h2>Place Your Order</h2>

      {error && <div className="error-message">{error}</div>}

      {result && (

        <div className="success-message">

          Order successful! Your order ID is: {result.orderId}

        </div>

      )}

      <form onSubmit={handleSubmit}>

        <div className="form-group">

          <label htmlFor="customerName">Name:</label>

          <input

            type="text"

            id="customerName"

            name="customerName"

            value={formData.customerName}

            onChange={handleChange}

            required

          />

        </div>

        <div className="form-group">

          <label htmlFor="email">Email:</label>

          <input

            type="email"

            id="email"

            name="email"

            value={formData.email}

            onChange={handleChange}

            required

          />

        </div>

        <div className="form-group">

          <label htmlFor="productId">Product ID:</label>

          <input

            type="text"

            id="productId"

            name="productId"

            value={formData.productId}

            onChange={handleChange}

            required

          />

        </div>

        <div className="form-group">

          <label htmlFor="quantity">Quantity:</label>

          <input

            type="number"

            id="quantity"

            name="quantity"

            min="1"

            value={formData.quantity}

            onChange={handleChange}

            required

          />

        </div>

        <div className="form-group">

          <label htmlFor="shippingAddress">Shipping Address:</label>

          <textarea

            id="shippingAddress"

            name="shippingAddress"

            value={formData.shippingAddress}

            onChange={handleChange}

            required

          />

        </div>

        <button type="submit" disabled={submitting}>

          {submitting ? 'Submitting...' : 'Place Order'}

        </button>

      </form>

    </div>

  );

}

export default OrderForm;

## Step 11: Test the Complete Flow

Now it’s time to test the entire system:

1. Deploy your React form to a hosting service or run locally
2. Make sure to update the API\_URL in the React form with your actual API Gateway URL
3. Fill out the form and submit an order
4. Check the CloudWatch logs for each component:
   * OrderProcessingLambda logs
   * SNS delivery logs
   * OrderInventoryLambda logs
   * BillingInventoryLambda logs
5. Take screenshots of successful logs showing the data flow

## Step 12: Add CSS for the React Form (Optional)

.order-form-container { max-width: 600px; margin: 0 auto; padding: 20px; background-color: #f9f9f9; border-radius: 8px; box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1); }

h2 { text-align: center; color: #333; margin-bottom: 20px; }

.form-group { margin-bottom: 15px; }

label { display: block; margin-bottom: 5px; font-weight: 600; color: #555; }

input, textarea { width: 100%; padding: 10px; border: 1px solid #ddd; border-radius: 4px; font-size: 16px; }

textarea { min-height: 100px; resize: vertical; }

button { display: block; width: 100%; padding: 12px; background-color: #4caf50; color: white; border: none; border-radius: 4px; font-size: 16px; font-weight: bold; cursor: pointer; transition: background-color 0.3s; }

button:hover { background-color: #3e8e41; }

button:disabled { background-color: #cccccc; cursor: not-allowed; }

.error-message { padding: 10px; background-color: #ffebee; color: #c62828; border-radius: 4px; margin-bottom: 15px; border-left: 4px solid #c62828; }

.success-message { padding: 10px; background-color: #e8f5e9; color: #2e7d32; border-radius: 4px; margin-bottom: 15px; border-left: 4px solid #2e7d32; }

## Step 13: Error Handling and Monitoring

Enhance your system with proper error handling:

1. Add Dead Letter Queues (DLQs) for your SQS queues:
   * Go to each SQS queue
   * Click “Edit”
   * Scroll to “Dead-letter queue” section
   * Select “Enable” and choose to create a new DLQ or use an existing one
   * Take screenshots of the DLQ configuration
2. Set up CloudWatch Alarms:
   * Go to CloudWatch in AWS Console
   * Click “Alarms” > “Create Alarm”
   * Select metrics for Lambda errors and SQS queue depth
   * Configure thresholds and notifications
   * Take screenshots of the configured alarms

## Complete Implementation Checklist

As you implement each component, check them off and include screenshots in your Word document:

* ☐ SQS Queues (OrderInventoryQueue, BillingInventoryQueue)
* ☐ SNS Topic (OrderProcessingTopic)
* ☐ SNS Subscriptions for both queues
* ☐ Order Processing Lambda
* ☐ Order Inventory Lambda
* ☐ Billing Inventory Lambda
* ☐ SQS triggers for Inventory Lambdas
* ☐ API Gateway configuration
* ☐ React Order Form implementation
* ☐ End-to-end testing with sample order
* ☐ CloudWatch logs showing the complete flow
* ☐ Error handling with DLQs (optional)
* ☐ CloudWatch Alarms (optional)

## Final Notes for Documentation

When capturing screenshots for your Word document:

1. Include AWS service configuration pages showing the connections between services
2. Show the code for each Lambda function
3. Include the React form UI
4. Capture CloudWatch logs showing message flow through the system
5. Document any permissions or IAM roles created
6. Include the final API Gateway URL

This implementation creates a robust, scalable order processing system that separates concerns between order processing and billing while ensuring reliable message delivery through AWS’s managed services.