Distributed Order Fulfillment System (DOFS)

The Distributed Order Fulfillment System (DOFS) is a fully event-driven, serverless order processing system built on AWS services. It implements a microservices architecture with the following key characteristics:

- **Event-Driven**: Uses AWS Step Functions for orchestration and SQS for asynchronous messaging
- **Serverless**: Built entirely on AWS Lambda functions with no server management
- **Fault-Tolerant**: Implements retry logic, dead letter queues, and error handling
- **Scalable**: Auto-scales based on demand using AWS managed services
- **Observable**: Includes comprehensive monitoring and alerting

System Components

1. API Layer

- **API Gateway (HTTP API v2)**: Entry point for all client requests
- **API Handler Lambda**: Processes incoming HTTP requests and initiates the order workflow

2. Orchestration Layer

- **Step Functions**: Coordinates the order processing workflow
- **Validator Lambda**: Validates incoming order data
- **Order Storage Lambda**: Persists valid orders to the database

3. Processing Layer

- **SQS Order Queue**: Handles asynchronous order processing
- **Fulfill Order Lambda**: Processes order fulfillment with simulated failure rates
- **SQS Dead Letter Queue**: Captures failed messages after max retries
- **DLQ Handler Lambda**: Processes failed orders and stores them for analysis

4. Data Layer

- **Orders DynamoDB Table**: Stores all order records
- **Failed Orders DynamoDB Table**: Stores failed order records for analysis
- **IAM Roles**: Provides least-privilege access control

5. Monitoring & Observability

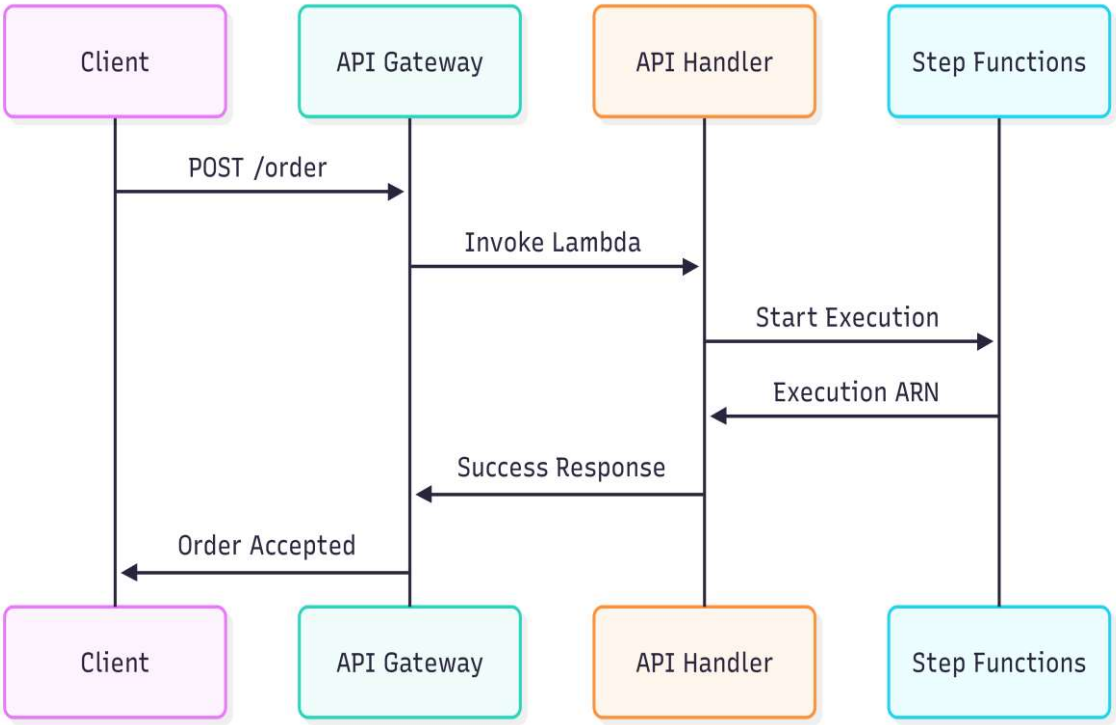
- **CloudWatch Logs**: Centralized logging for all Lambda functions
- **CloudWatch Metrics**: Performance and operational metrics
- **CloudWatch Alarms**: Alerts when DLQ depth exceeds threshold

6. CI/CD Pipeline

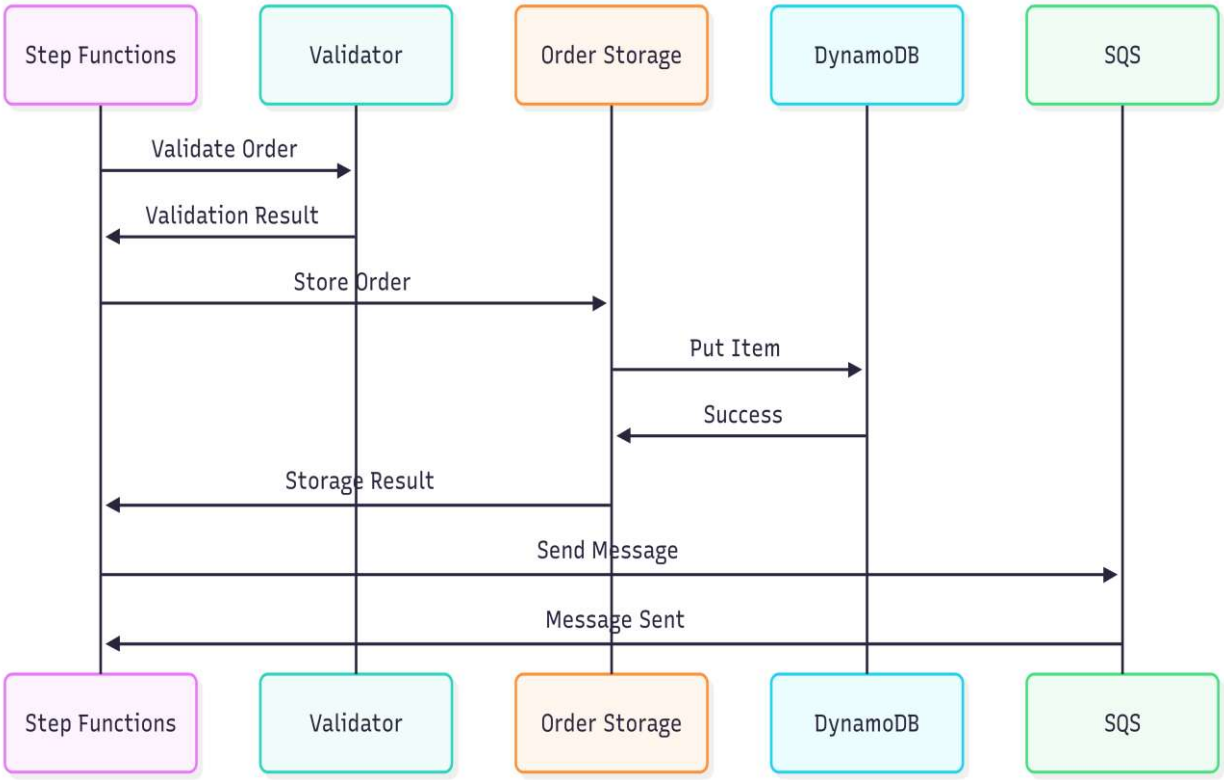
- **GitHub Repository**: Source code management
- **CodePipeline**: Automated deployment pipeline
- **CodeBuild**: Build and deployment automation

ORDER PROCESSING FLOW

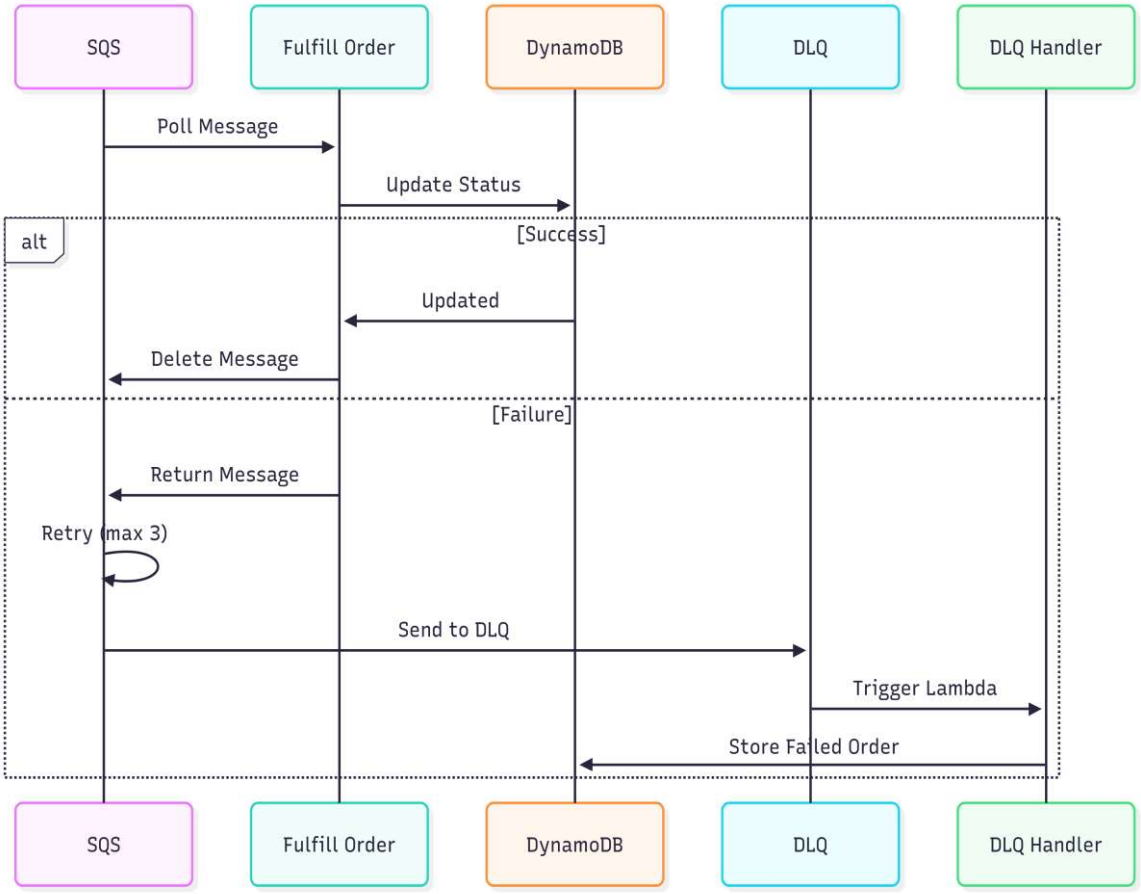
1. Order Submission


Client
API Gateway
API Handler
Step Functions
POST /order
Invoke Lambda
Start Execution
Execution ARN
Success Response
Order Accepted
Client
API Gateway
API Handler
Step Functions

2. Order Validation and Storage



3. Order Fulfilllment



Deployment Guide

Prerequisites

- AWS CLI configured with appropriate permissions
- Terraform 1.6+ installed
- Python 3.12+ installed
- GitHub personal access token

Step 1: Clone Repository

bash

```
git clone https://github.com/adityaoncloud/dofs-project.git
cd dofs-project
```

Step 2: Configure Backend

Edit `terraform/backend.tf`:

hcl

```
terraform {
  backend "s3" {
    bucket = "your-terraform-state-bucket"
    key    = "dofs/dev/terraform.tfstate"
    region = "ap-south-1"
    encrypt = true
  }
}
```

Step 3: Initialize Terraform

bash

```
cd terraform
terraform init
```

Step 4: Package Lambda Functions

bash

```
# Create ZIP files for each Lambda function
cd ../lambdas/api_handler
zip -r function.zip handler.py

cd ../validator
zip -r function.zip handler.py

cd ../order_storage
zip -r function.zip handler.py

cd ../fulfill_order
zip -r function.zip handler.py
```

```
cd ../dlq_handler
zip -r function.zip handler.py
```

Step 5: Deploy Infrastructure

bash

```
cd ../../terraform
terraform plan
terraform apply
```

Step 6: Configure CI/CD

The pipeline is automatically configured and will trigger on pushes to the main branch.

Monitoring & Observability

CloudWatch Metrics

- **Lambda Duration**: Execution time for each function
- **Lambda Errors**: Error count and error rate
- **SQS Queue Depth**: Number of messages in queue
- **DLQ Depth**: Number of failed messages

CloudWatch Alarms

- **DLQ Depth Alarm**: Triggers when DLQ has more than 1 message
- **Lambda Error Rate**: Alerts on high error rates
- **API Gateway 4xx/5xx**: Monitors API errors

Logging Strategy

Each Lambda function logs to CloudWatch with structured logging:

CI/CD Pipeline

Pipeline Stages

1. **Source Stage**
    a. Monitors GitHub repository
    b. Triggers on push to main branch
    c. Uses CodeStar connection for authentication
2. **Build Stage**
    a. Runs in CodeBuild environment
    b. Installs Terraform
    c. Validates and applies infrastructure changes

BuildSpec Configuration

yaml

```yaml
version: 0.2
phases:
  install:
    runtime-versions:
      python: 3.12
    commands:
      - echo Installing Terraform...
      - curl -O
https://releases.hashicorp.com/terraform/1.6.6/terraform_1.6.6_linux_amd64.zip
      - unzip terraform_1.6.6_linux_amd64.zip
      - sudo mv terraform /usr/local/bin/
  pre_build:
    commands:
      - cd terraform
      - terraform init
  build:
    commands:
      - terraform plan
      - terraform apply -auto-approve
```

System Architecture