Serverless E-Commerce Order Processing System on AWS

# 🎯 Objective

Develop a secure, scalable, and cost-efficient e-commerce order processing system using AWS serverless services. The project includes a web-based UI hosted on S3 & CloudFront, an API built with API Gateway & Lambda, DynamoDB for persistence, and AWS WAF for security.

# 🏗️ High-Level Architecture

The architecture follows AWS serverless best practices. A static frontend website is hosted on Amazon S3 and distributed via CloudFront. API Gateway provides REST endpoints, backed by Lambda functions for business logic. Orders and products are persisted in DynamoDB. AWS WAF provides protection from common exploits.

# ⚙️ AWS Services Used

**• Amazon S3:** Stores static website assets (HTML, CSS, JS) and product images. S3 provides high availability and scales automatically.

**• Amazon CloudFront:** CDN that serves the website globally with low latency, HTTPS, caching, and integrates with AWS WAF.

**• Amazon API Gateway:** Fully managed API management service. Provides REST endpoints (/products, /orders), handles CORS, throttling, and integrates with Lambda.

**• AWS Lambda:** Serverless compute for business logic. Functions: listProducts (read DynamoDB) and createOrder (insert order).

**• Amazon DynamoDB:** NoSQL database storing product catalog and orders. Provides millisecond performance and auto-scaling.

**• AWS WAF v2:** Web Application Firewall attached to CloudFront and API Gateway. Protects against SQLi, XSS, bots using managed rule groups.

**• Amazon CloudWatch:** Monitoring service. Logs from Lambda, API Gateway, and WAF.

# 📖 Step-by-Step Deployment & Steps Followed

1. Designed the high-level architecture using AWS serverless services.
2. Created DynamoDB tables: Products (productId PK), Orders (orderId PK).
3. Configured S3 bucket to host static frontend website and product images.
4. Uploaded frontend (HTML/JS/CSS) to S3 bucket.
5. Implemented Lambda functions: listProducts and createOrder.
6. Assigned least privilege IAM policies to Lambda roles.
7. Built REST APIs using API Gateway with /products and /orders routes.
8. Integrated API Gateway with Lambda functions.
9. Enabled CORS in API Gateway.
10. Created CloudFront distribution with S3 origin (OAC enabled).
11. Attached WAF Web ACL to CloudFront and API Gateway.
12. Tested end-to-end flow: UI → API → DynamoDB.
13. Verified Orders stored in DynamoDB.
14. Configured CloudWatch logging.

🔑 Example IAM Policies (Least Privilege)

Policy for listProducts Lambda:

{  
 "Version": "2012-10-17",  
 "Statement": [{  
 "Effect": "Allow",  
 "Action": ["dynamodb:Scan"],  
 "Resource": "arn:aws:dynamodb:<REGION>:<ACCOUNT\_ID>:table/Products"  
 }]  
}

Policy for createOrder Lambda:

{  
 "Version": "2012-10-17",  
 "Statement": [{  
 "Effect": "Allow",  
 "Action": ["dynamodb:PutItem"],  
 "Resource": "arn:aws:dynamodb:<REGION>:<ACCOUNT\_ID>:table/Orders"  
 }]  
}

# 💻 Lambda Function Code

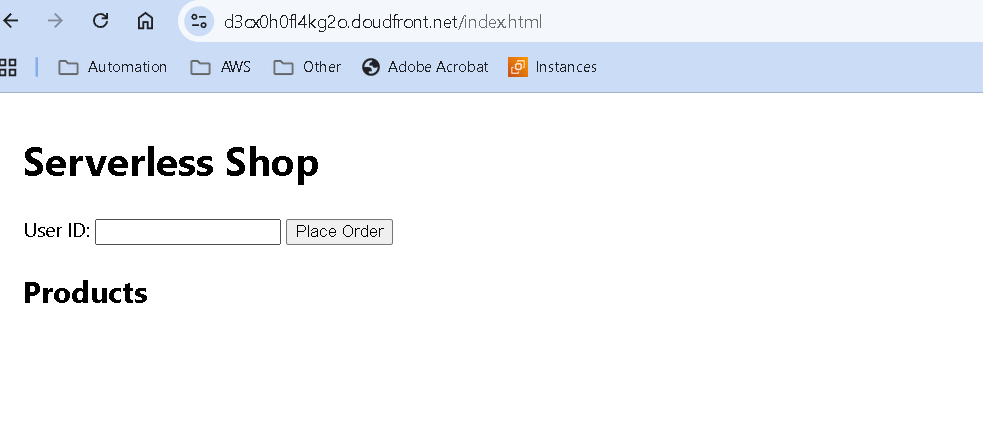
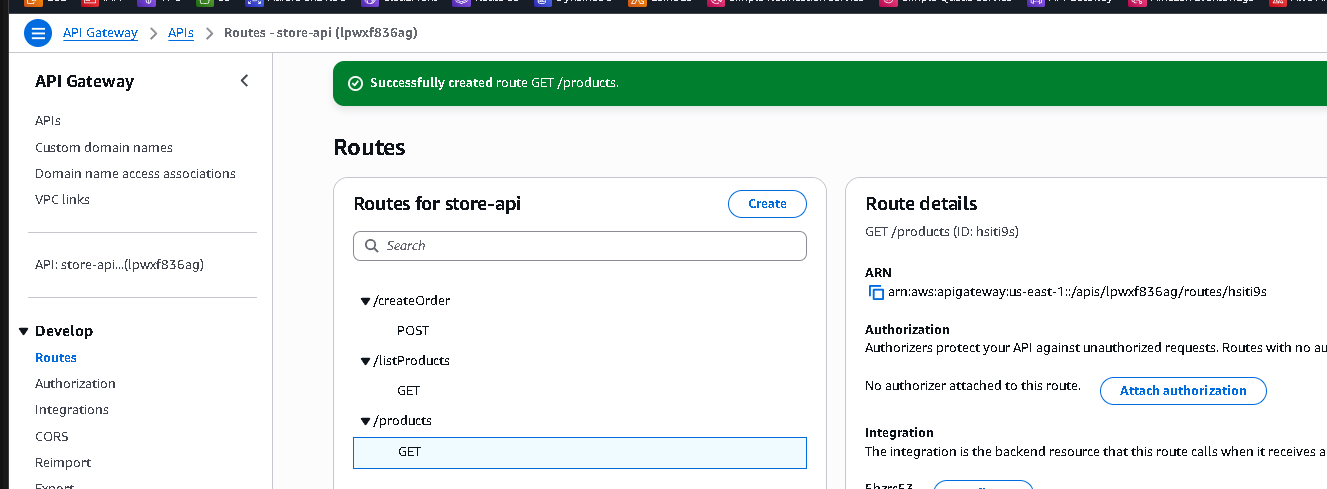
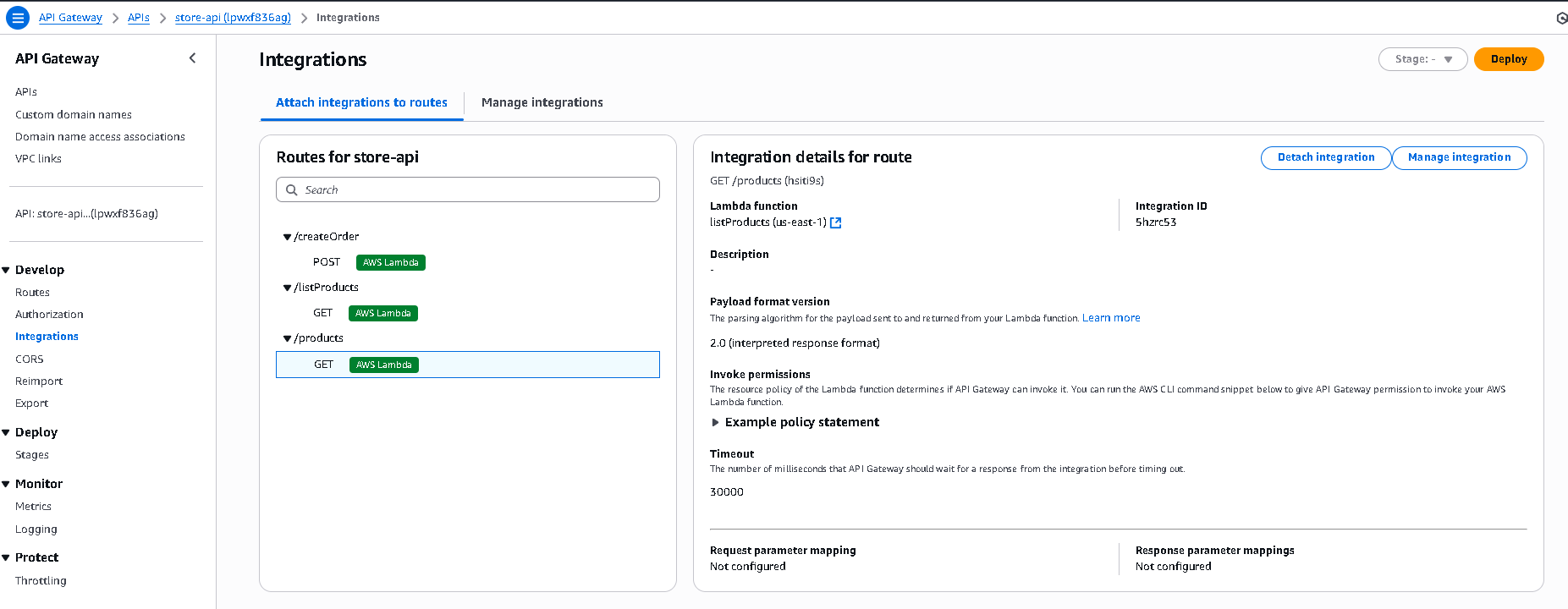
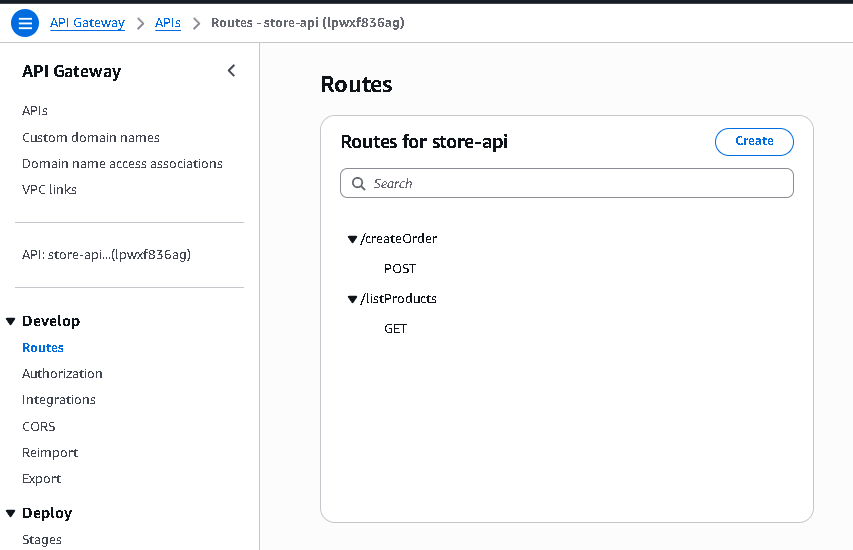
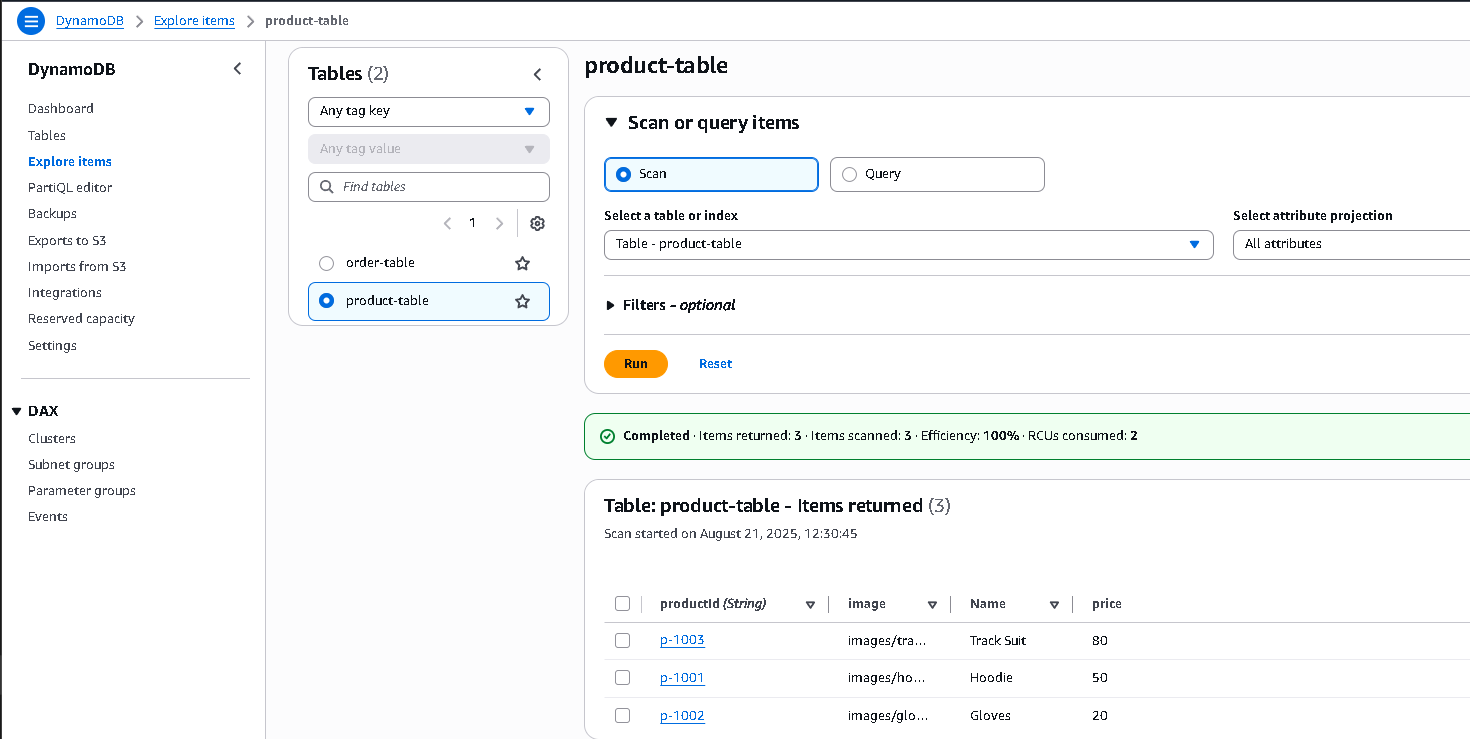
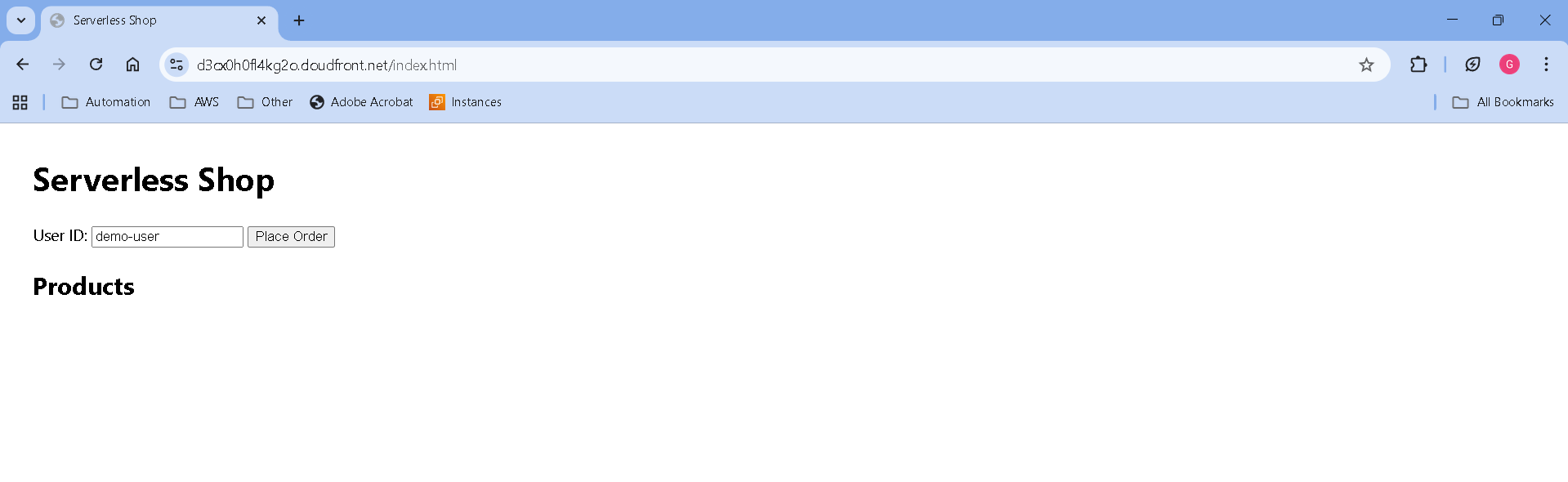
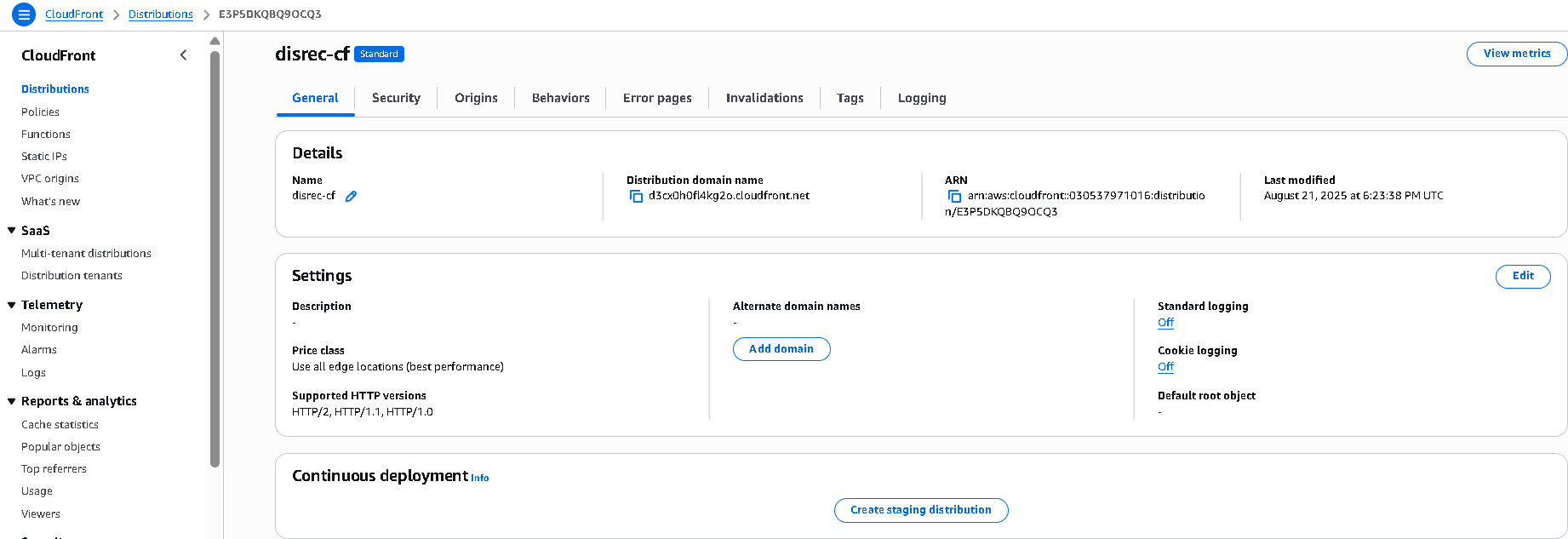
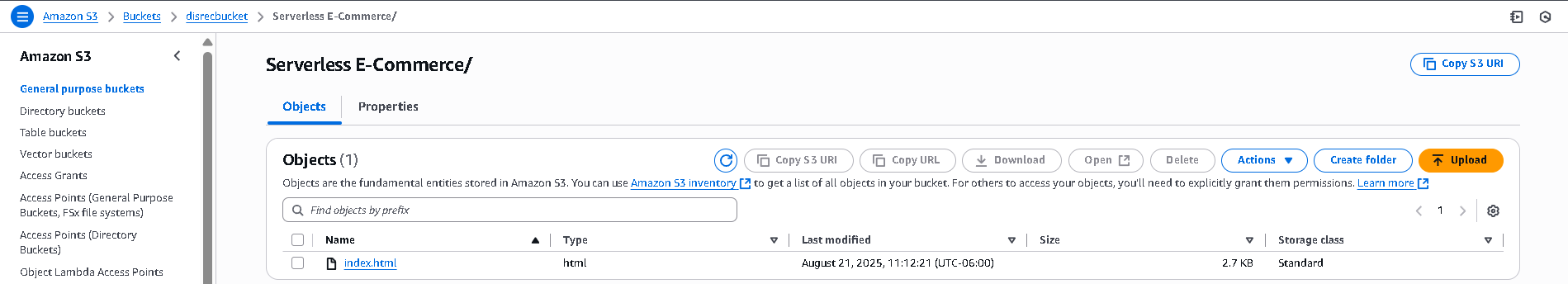
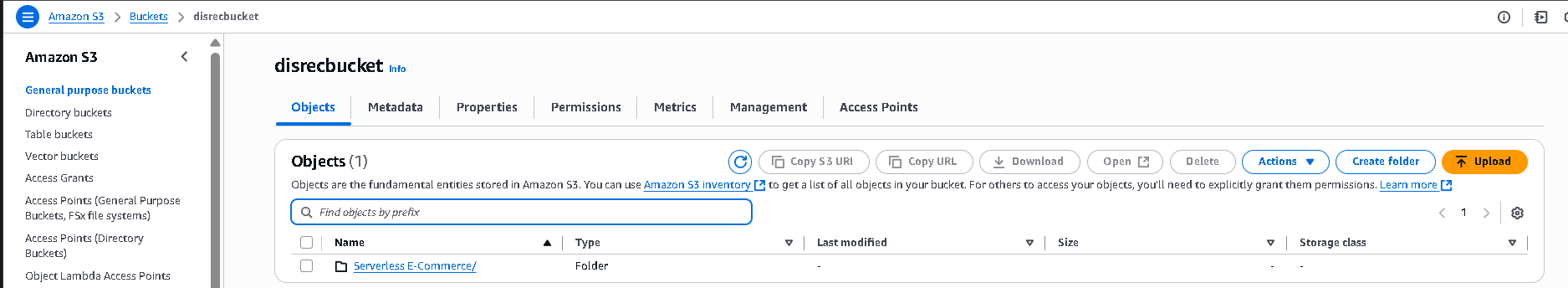
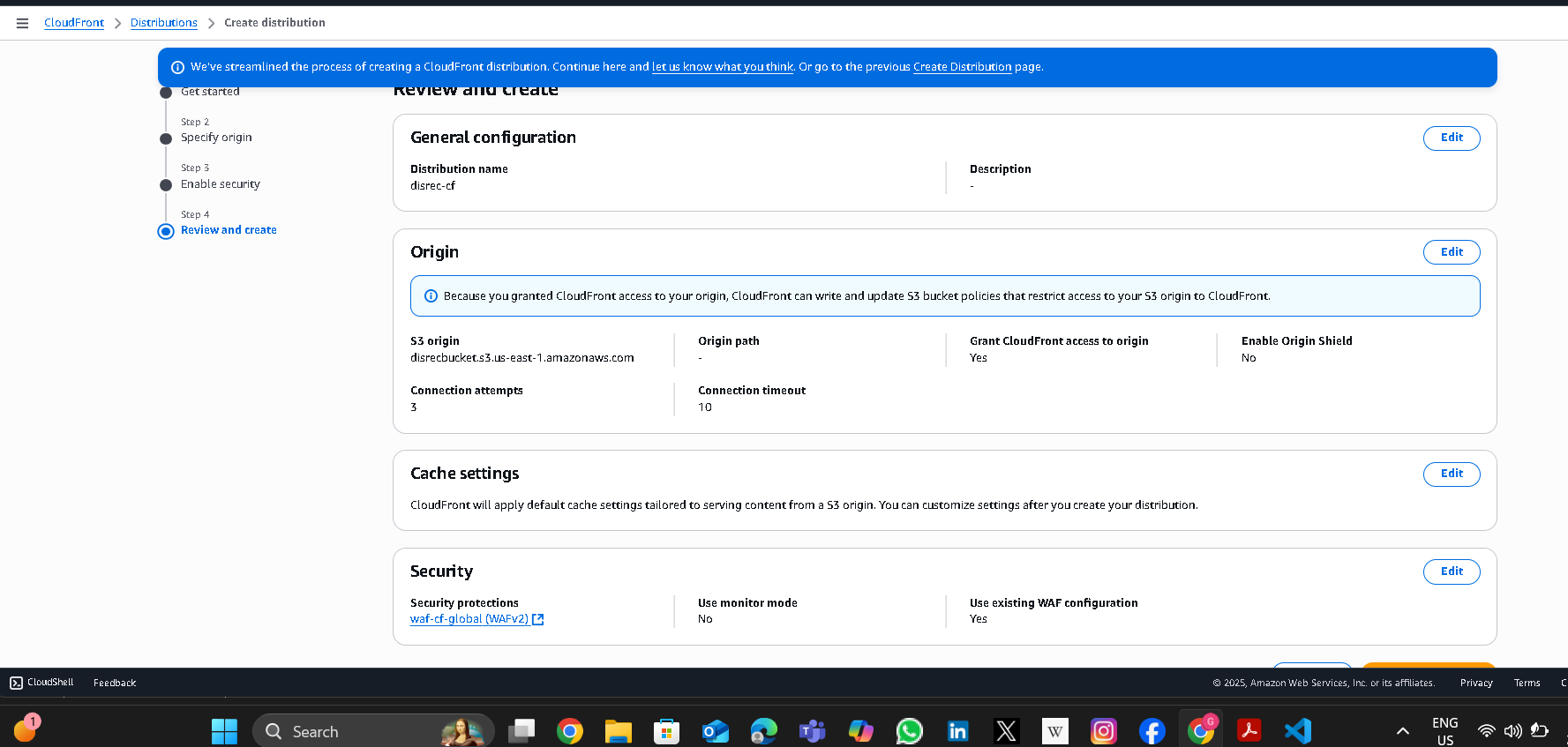
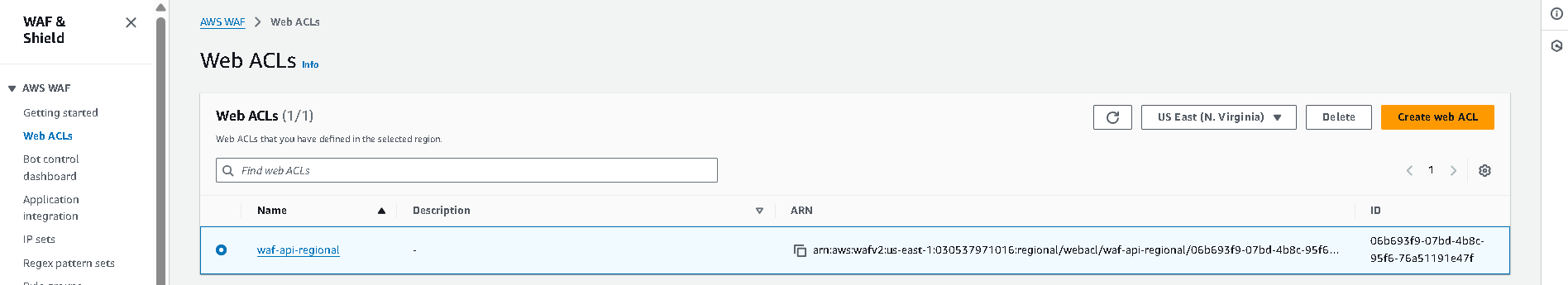
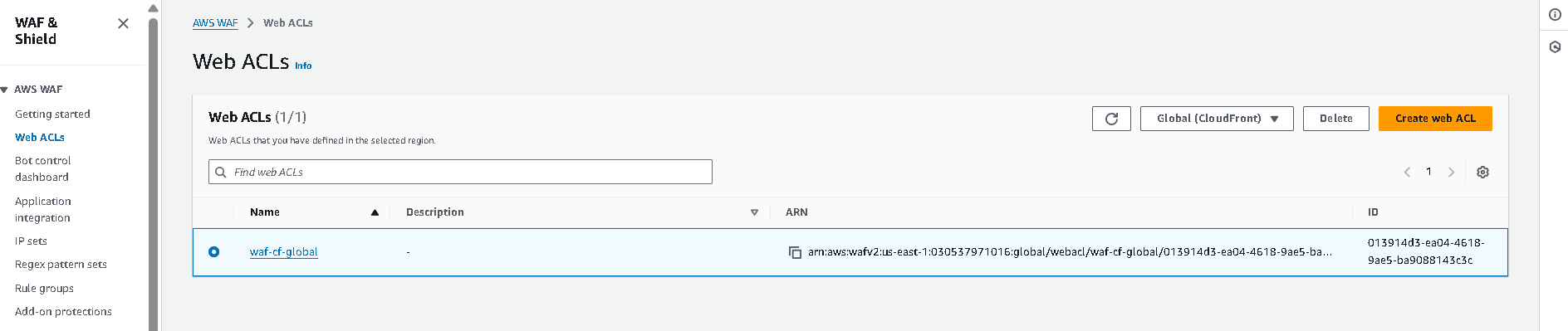
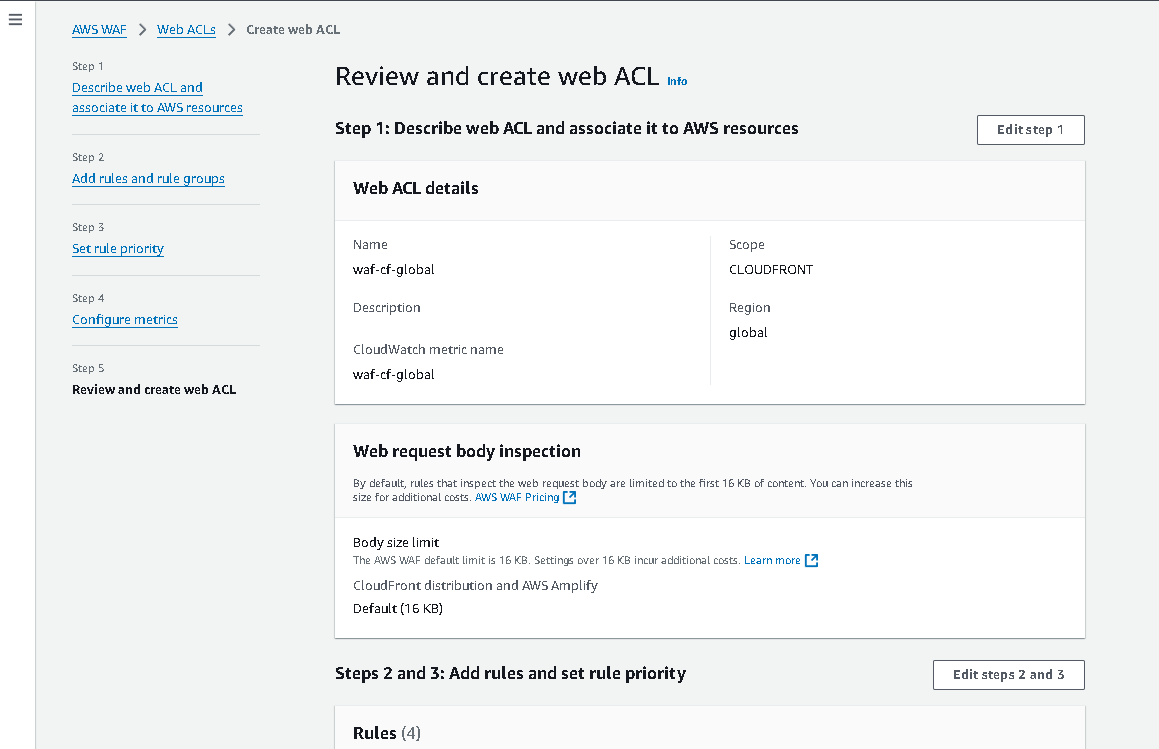
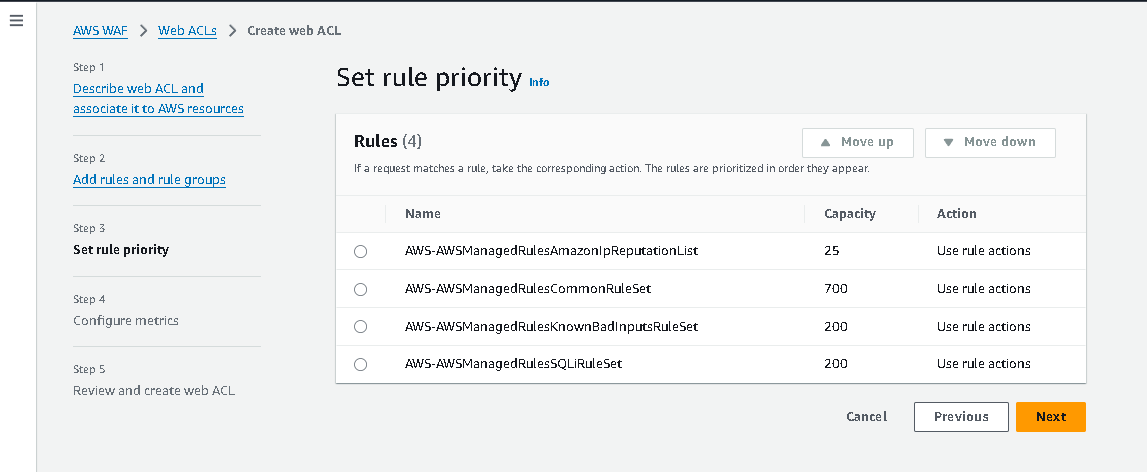
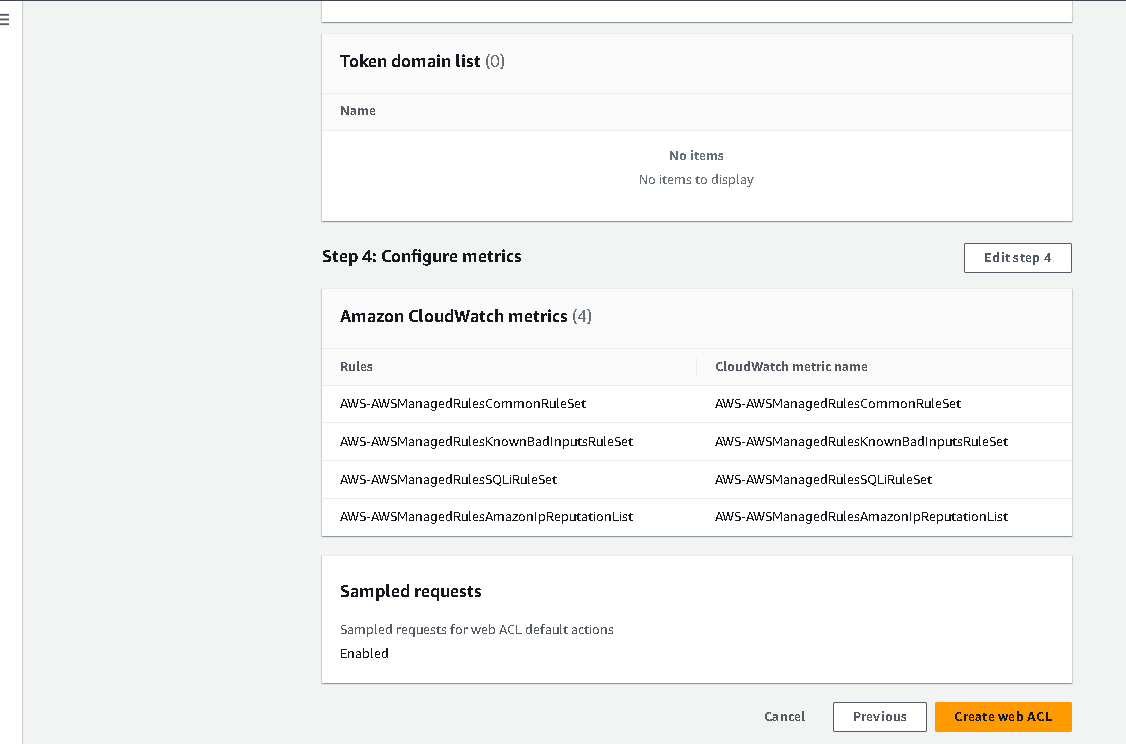
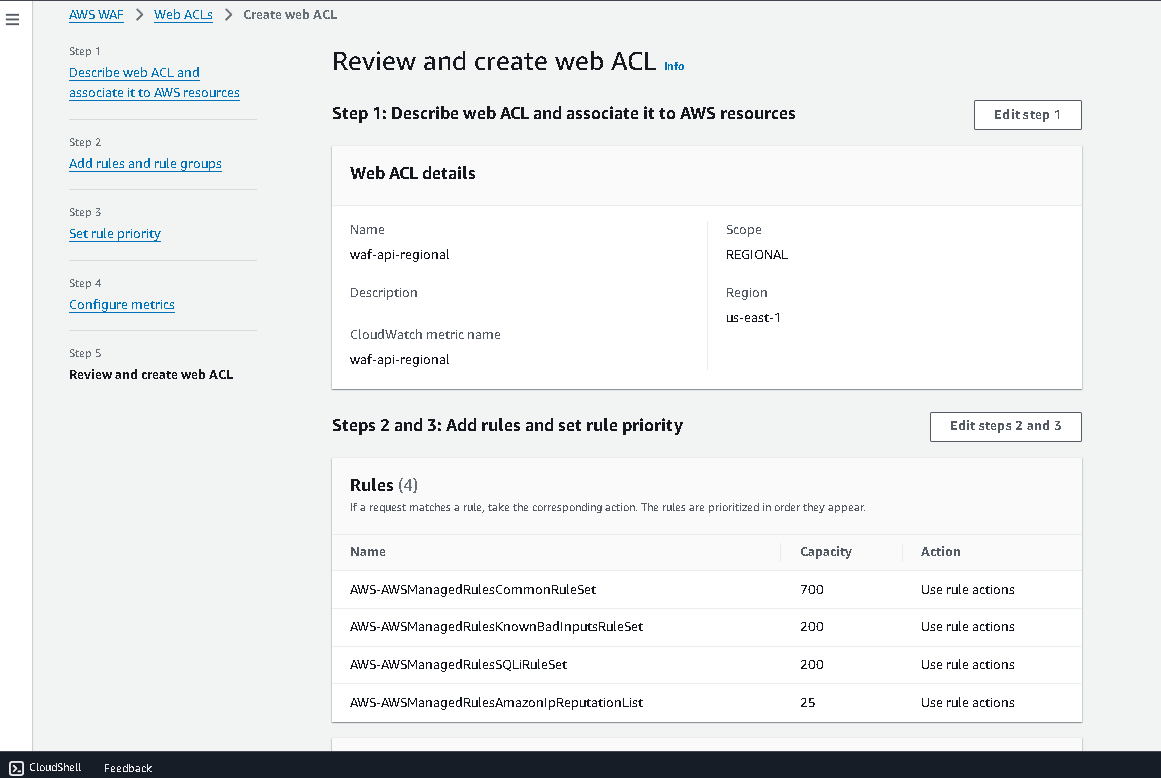
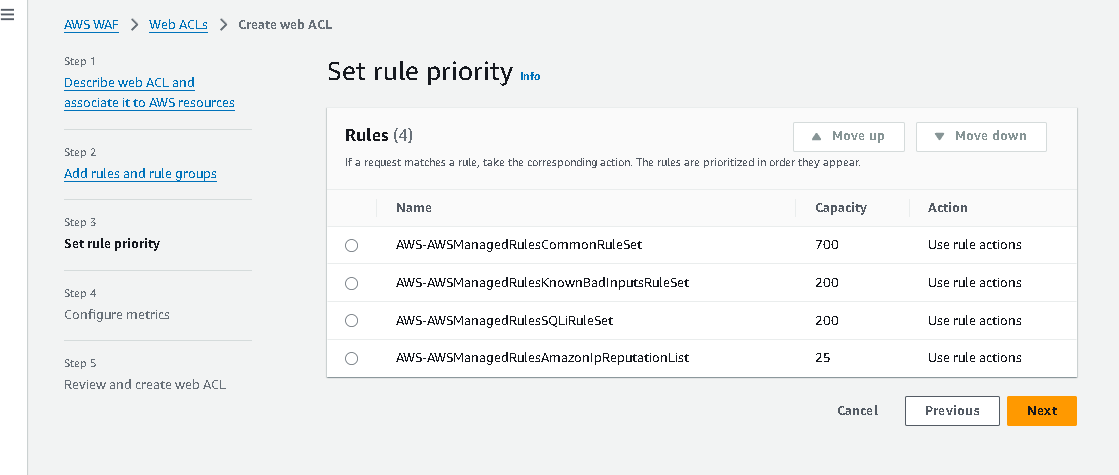
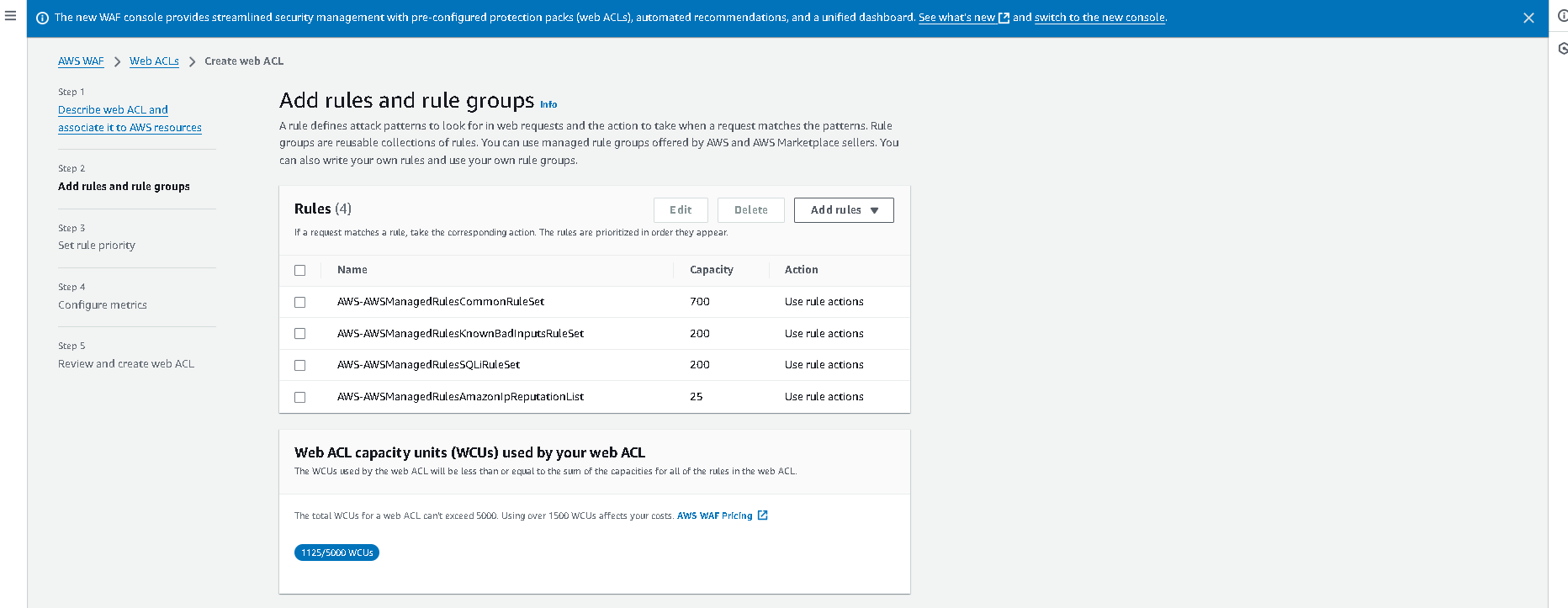
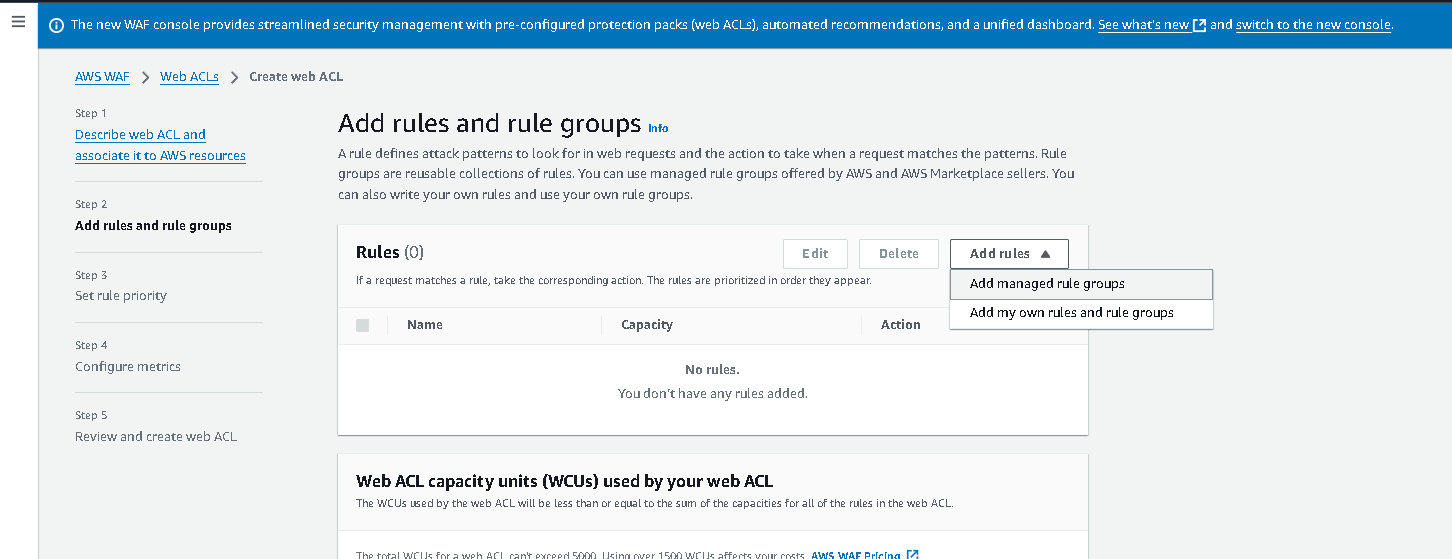
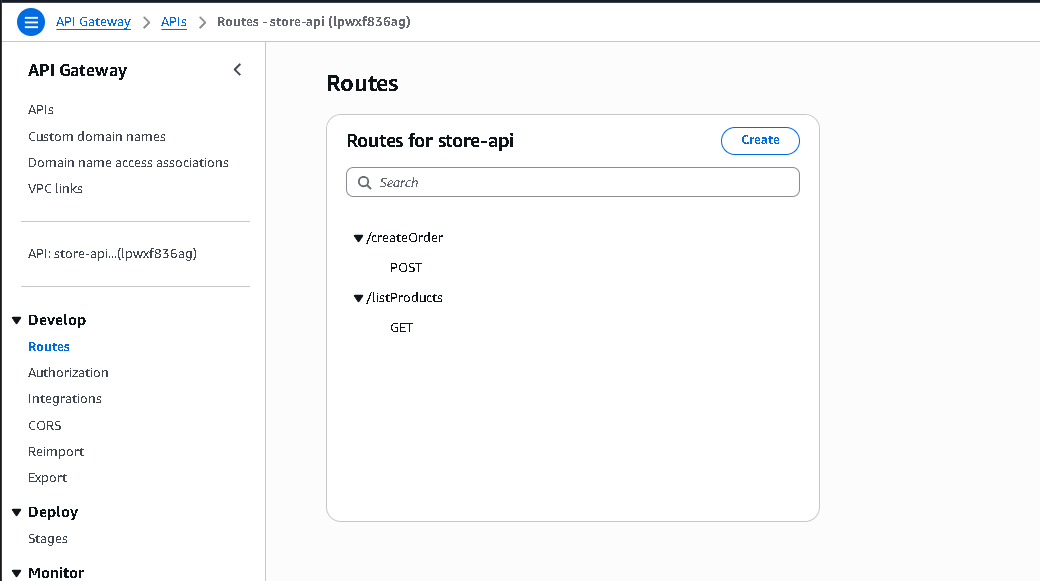
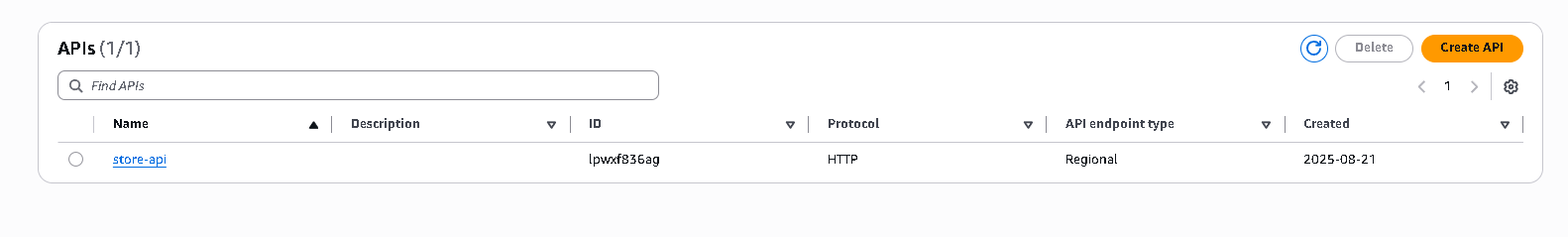
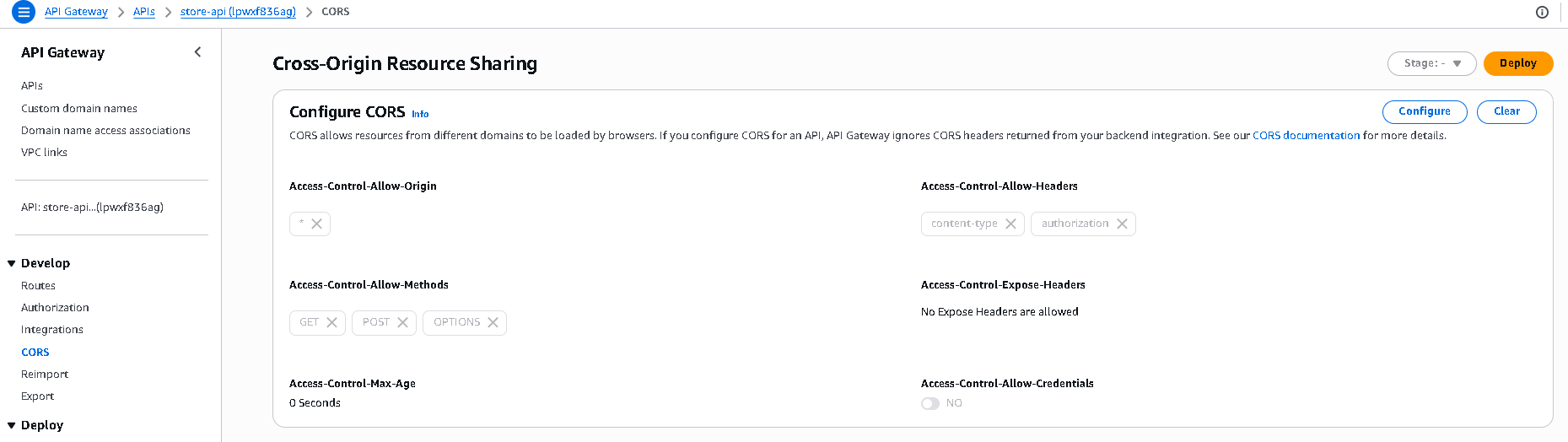
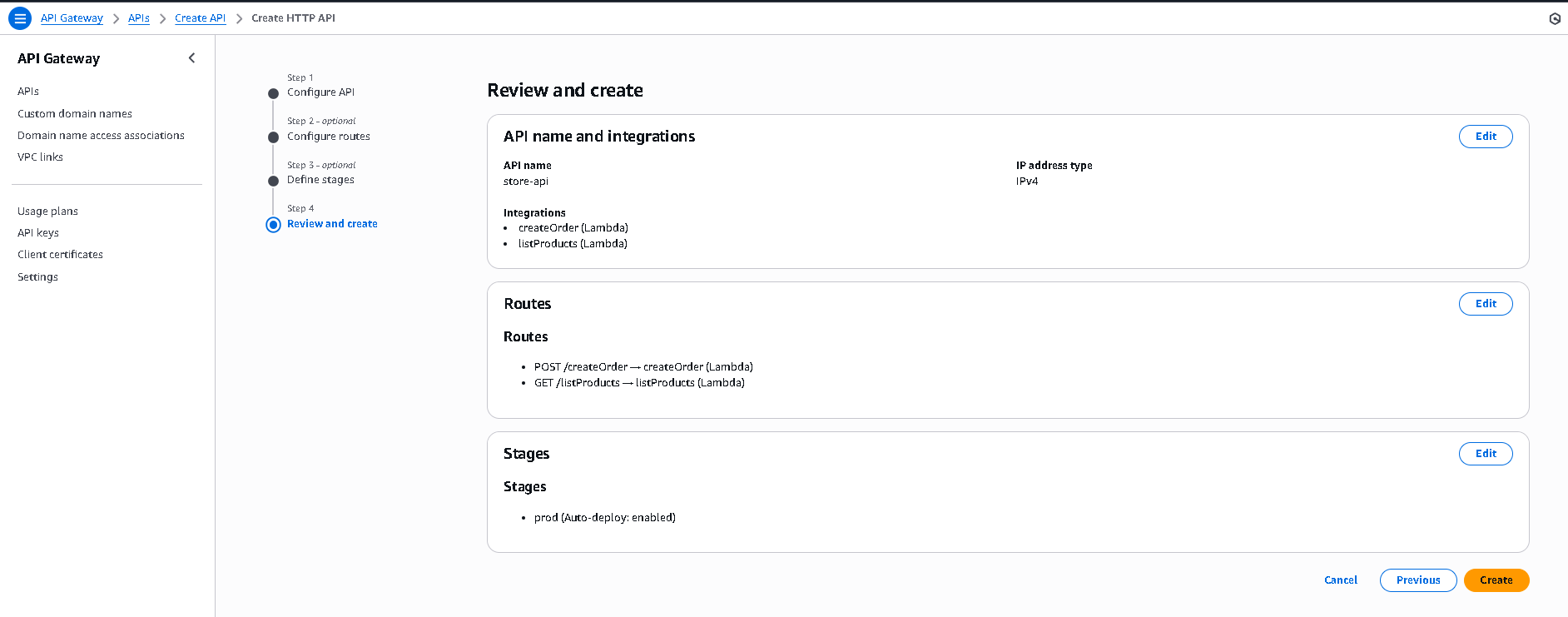
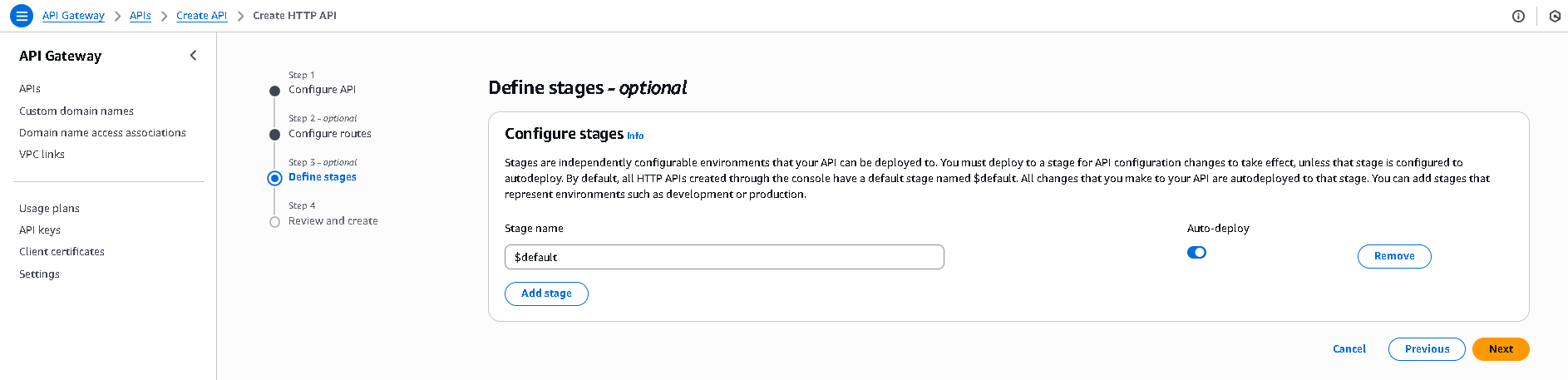
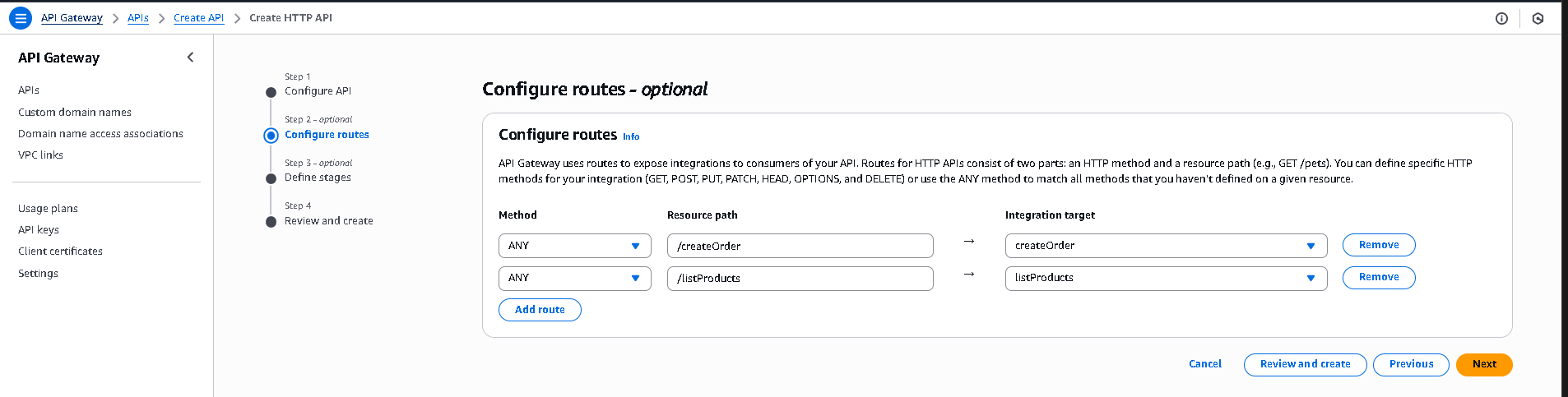
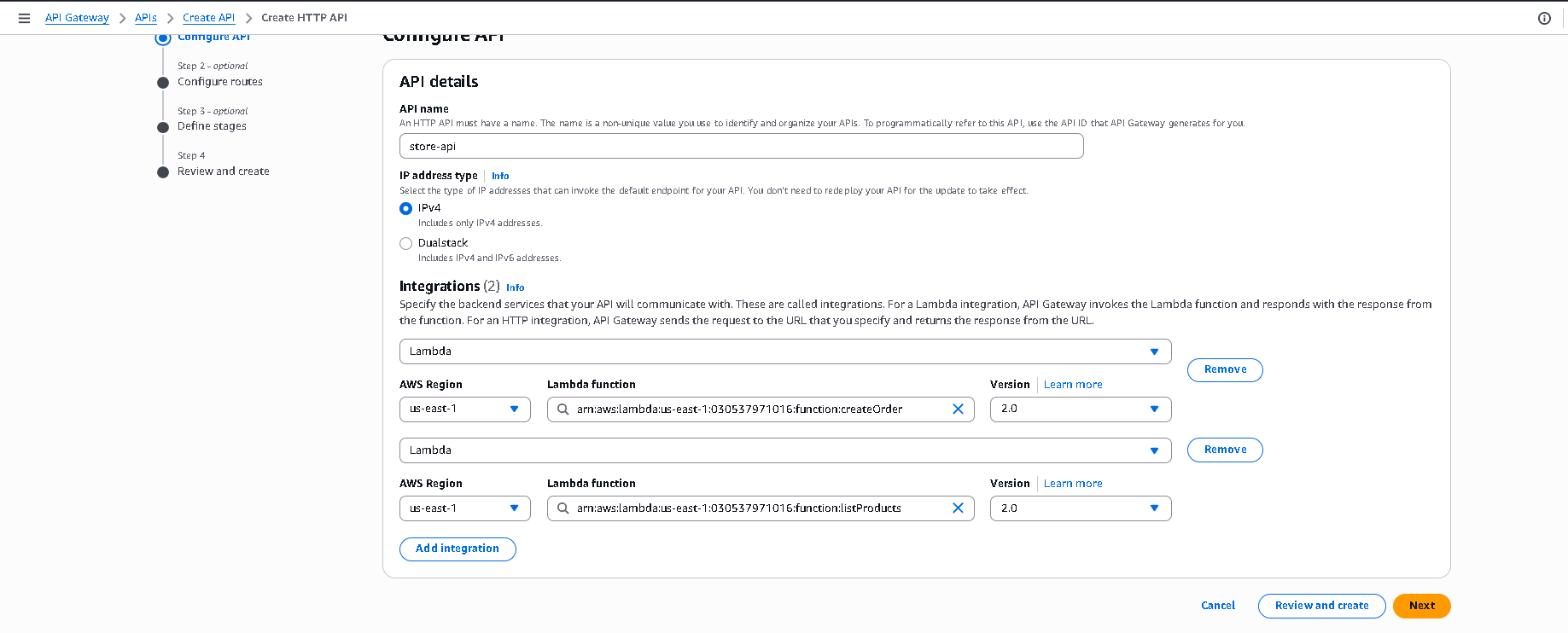
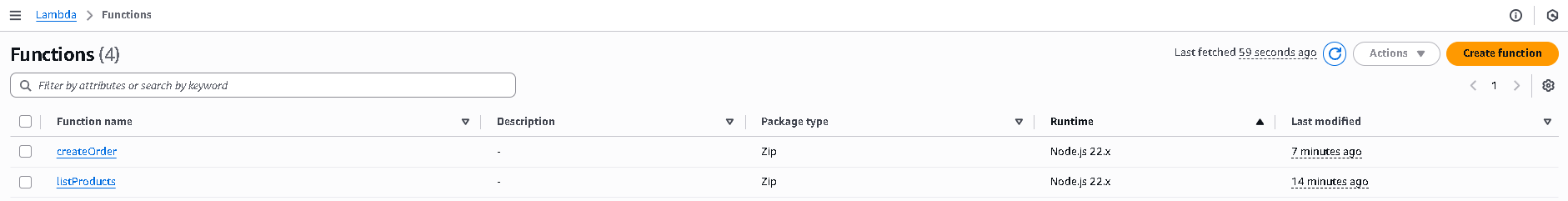
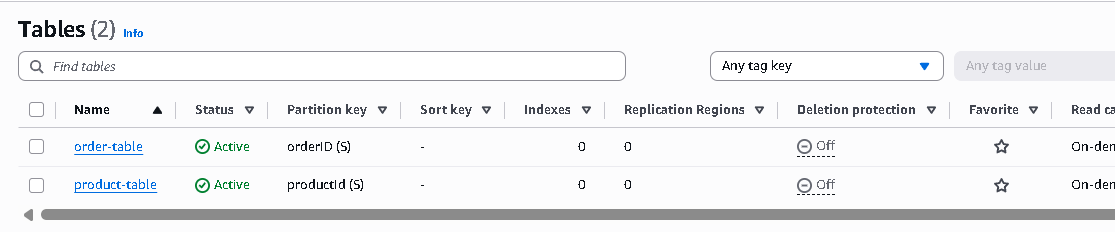
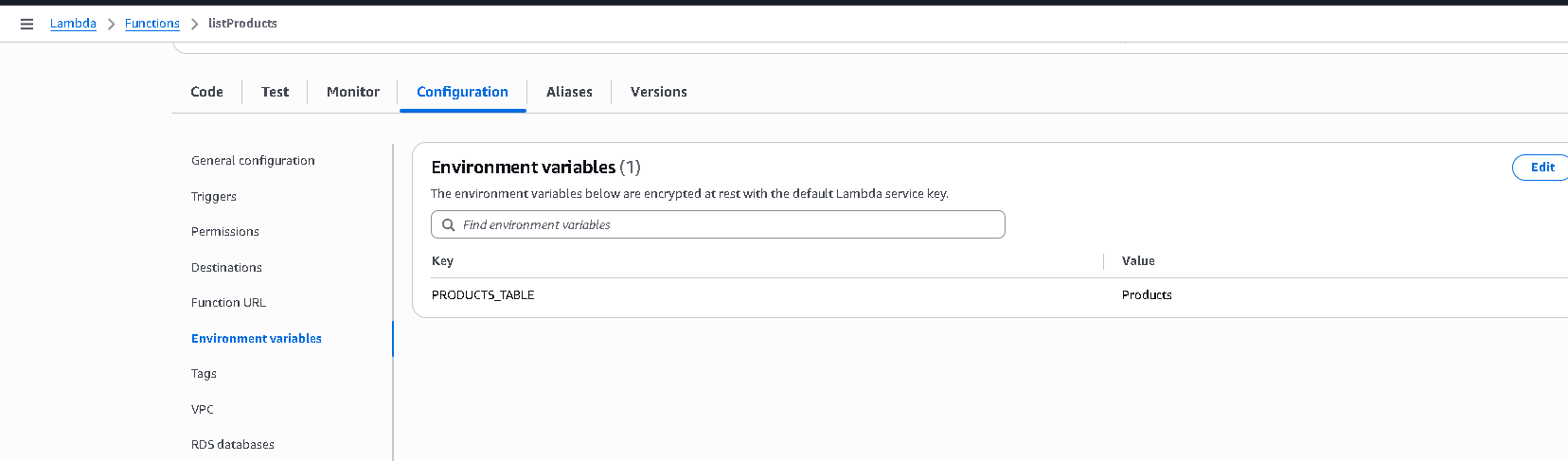
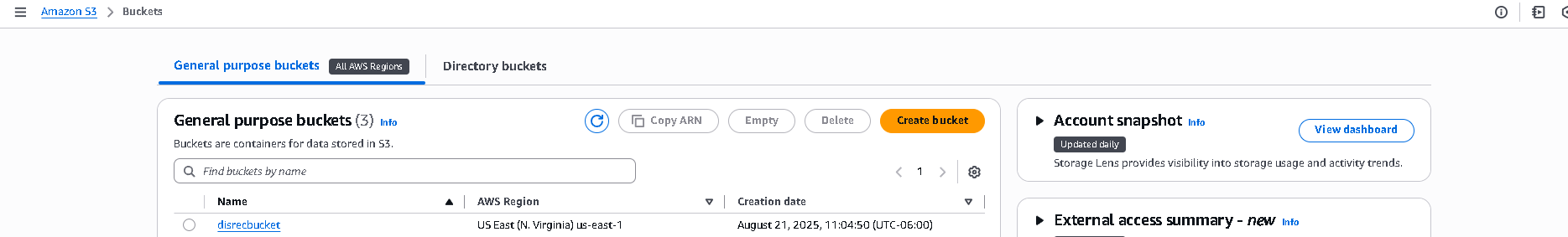
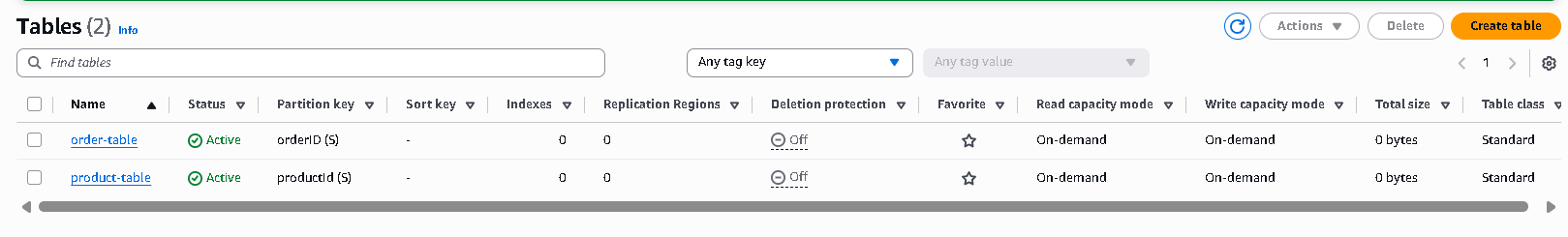
listProducts Lambda:

const AWS = require('aws-sdk');  
const ddb = new AWS.DynamoDB.DocumentClient();  
const PRODUCTS\_TABLE = process.env.PRODUCTS\_TABLE;  
  
exports.handler = async () => {  
 const data = await ddb.scan({ TableName: PRODUCTS\_TABLE }).promise();  
 return { statusCode: 200, headers: { 'Content-Type': 'application/json' }, body: JSON.stringify(data.Items || []) };  
};

createOrder Lambda:

const { v4: uuidv4 } = require('uuid');  
const AWS = require('aws-sdk');  
const ddb = new AWS.DynamoDB.DocumentClient();  
const ORDERS\_TABLE = process.env.ORDERS\_TABLE;  
  
exports.handler = async (event) => {  
 const body = JSON.parse(event.body || "{}");  
 const orderId = uuidv4();  
 const now = Date.now();  
 const item = { orderId, userId: body.userId, items: body.items, total: body.total, orderDate: now, status: 'PLACED' };  
 await ddb.put({ TableName: ORDERS\_TABLE, Item: item }).promise();  
 return { statusCode: 201, headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({ orderId }) };  
};

# 🖼️ Screenshots



# 📚 References

* AWS WAF Managed Rules: https://docs.aws.amazon.com/waf/latest/developerguide/aws-managed-rule-groups-list.html
* CloudFront OAC: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/private-content-restricting-access-to-s3.html
* API Gateway HTTP APIs: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api.html
* AWS Lambda Guide: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
* DynamoDB Guide: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
* Amazon S3 Guide: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html