PRJ-SAP-015: Comprehensive Deployment Guide (Monolith to Microservices)

Certification: AWS Certified Solutions Architect – Professional
Domain: Application Architecture & Design
Author: Mo Suleiman

---

1. Business Context & Project Overview

Legacy monolithic applications often become bottlenecks for business agility. They are difficult to scale independently, risky to update, and slow down release cycles. However, a “big bang” rewrite is highly risky and often fails.

This project demonstrates the **Strangler Fig Pattern**, a low-risk, incremental approach to modernization. By placing an Amazon API Gateway facade in front of the legacy monolith, we can intercept traffic and gradually route specific endpoints to newly built microservices. This allows the business to deliver new features quickly using modern serverless (AWS Lambda) and containerized (AWS Fargate) architectures, while slowly deprecating the legacy system without downtime.

Key Objectives

- Implement the Strangler Fig Pattern using Amazon API Gateway as a facade.
- Decompose a specific function (Users) into a serverless microservice using AWS Lambda and DynamoDB.
- Decompose another function (Orders) into a containerized microservice using AWS Fargate and Amazon Aurora.
- Introduce an event-driven, asynchronous communication pattern using Amazon SNS and SQS to prevent tight coupling.

---

2. GRC Mapping & Compliance

Framework / Standard	Requirement	How This Architecture Addresses It
NIST CSF	PR.AC-5: Network integrity is protected	API Gateway acts as a single entry point, enforcing rate limiting, WAF rules, and authentication before traffic reaches backend services.
PCI DSS	7.1.1: Restrict access to systems	Decomposing the monolith allows for granular IAM roles per microservice (least privilege) rather than one overly permissive role for the entire monolith.
AWS Well-Architected	Reliability Pillar	Strangler Fig reduces deployment risk. Event-driven architectures (SNS/SQS) provide buffer and retry mechanisms, preventing cascading failures.

---

3. Architecture Evolution

The architecture evolves through three distinct phases:

1. **Initial State:** A monolithic application on EC2 connected to a single database.
2. **Phase 1 (The Facade):** Amazon API Gateway is introduced, proxying all traffic (`/{proxy+}`) to the EC2 monolith.
3. **Phase 2 (Serverless Extraction):** The `/users` endpoint is intercepted by API Gateway and routed to a new AWS Lambda function backed by DynamoDB.
4. **Phase 3 (Containerized Extraction & Events):** The `/orders` endpoint is routed via an HTTP API with a VPC Link V2 to an AWS Fargate container. The Fargate service publishes `OrderCreated` events to an SNS topic, decoupling downstream processes.

---

4. Prerequisites & Permissions

To deploy this project, you need an AWS Account and an IAM User/Role with the following permissions:

- `AmazonEC2FullAccess`
- `AmazonAPIGatewayAdministrator`
- `AWSLambda_FullAccess`
- `AmazonDynamoDBFullAccess`
- `AmazonECS_FullAccess`
- `AmazonSNSFullAccess`
- `AmazonSQSFullAccess`
- `AWSCloudFormationFullAccess` (if using IaC)

---

5. Foundational Infrastructure as Code (IaC)

To accelerate deployment, use either Terraform or CloudFormation to provision the foundational resources (VPC, EC2 Monolith, DynamoDB, SNS/SQS).

Option A: Terraform Template (`main.tf`)

```
 provider "aws" {
  region = "us-east-1"
}

# 1. DynamoDB Table for Users Microservice
resource "aws_dynamodb_table" "users_table" {
  name           = "Users"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "UserID"

  attribute {
    name = "UserID"
    type = "S"
  }
}

# 2. SNS Topic for Order Events
resource "aws_sns_topic" "order_created" {
  name = "OrderCreated"
}

# 3. SQS Queue for Notifications Service
resource "aws_sqs_queue" "notifications_queue" {
  name = "NotificationsQueue"
}

# 4. SNS to SQS Subscription
resource "aws_sns_topic_subscription" "order_to_notifications" {
  topic_arn = aws_sns_topic.order_created.arn
  protocol  = "sqs"
  endpoint  = aws_sqs_queue.notifications_queue.arn
}

# 5. Security Group for Monolith
resource "aws_security_group" "monolith_sg" {
  name        = "monolith-sg"
  description = "Allow HTTP inbound traffic"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# 6. EC2 Instance (Simulated Monolith)
resource "aws_instance" "monolith" {
  ami           = "ami-04b70fa74e45c3917" # Ubuntu 24.04 LTS (us-east-1)
  instance_type = "t2.micro"
  vpc_security_group_ids = [aws_security_group.monolith_sg.id]

  user_data = <<-EOF
              #!/bin/bash
              export DEBIAN_FRONTEND=noninteractive
              apt-get update -y
              curl -fsSL https://deb.nodesource.com/setup_20.x | bash -
              apt-get install -y nodejs
              npm install -g pm2
              mkdir -p /home/ubuntu/monolith-app
              cd /home/ubuntu/monolith-app
              npm init -y
              npm install express
              cat << 'APP' > app.js
              const express = require("express");
              const app = express();
              app.use(express.json());
              app.get("/products", (req, res) => res.json({ source: "monolith",
message: "Here are the products!" }));
              app.get("/inventory", (req, res) => res.json({ source:
"monolith", message: "Here is the inventory!" }));
              app.get("/health", (req, res) => res.json({ status: "healthy"
}));
              app.listen(80, () => console.log("Monolith listening on port
80"));
              APP
              pm2 start app.js --name monolith
              pm2 save
              EOF

  tags = {
    Name = "Legacy-Monolith"
  }
```

```
}

output "monolith_public_ip" {
  value = aws_instance.monolith.public_ip
}
```

Option B: AWS CloudFormation Template (`template.yaml`)

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Foundational resources for PRJ-SAP-015 Monolith Decomposition

Resources:
  UsersTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: Users
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: UserID
          AttributeType: S
      KeySchema:
        - AttributeName: UserID
          KeyType: HASH

  OrderCreatedTopic:
    Type: AWS::SNS::Topic
    Properties:
      TopicName: OrderCreated

  NotificationsQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: NotificationsQueue

  SNSSQSSubscription:
    Type: AWS::SNS::Subscription
    Properties:
      Protocol: sqs
      Endpoint: !GetAtt NotificationsQueue.Arn
      TopicArn: !Ref OrderCreatedTopic

  MonolithSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Allow HTTP inbound traffic
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 80
          ToPort: 80
          CidrIp: 0.0.0.0/0
```

```
  MonolithInstance:
    Type: AWS::EC2::Instance
    Properties:
      InstanceType: t2.micro
      ImageId: ami-04b70fa74e45c3917 # Ubuntu 24.04 LTS (us-east-1)
      SecurityGroupIds:
        - !Ref MonolithSecurityGroup
      UserData:
        Fn::Base64: !Sub |
          #!/bin/bash
          export DEBIAN_FRONTEND=noninteractive
          apt-get update -y
          curl -fsSL https://deb.nodesource.com/setup_20.x | bash -
          apt-get install -y nodejs
          npm install -g pm2
          mkdir -p /home/ubuntu/monolith-app
          cd /home/ubuntu/monolith-app
          npm init -y
          npm install express
          cat << 'APP' > app.js
          const express = require("express");
          const app = express();
          app.use(express.json());
          app.get("/products", (req, res) => res.json({ source: "monolith",
message: "Here are the products!" }));
          app.get("/inventory", (req, res) => res.json({ source: "monolith",
message: "Here is the inventory!" }));
          app.get("/health", (req, res) => res.json({ status: "healthy" }));
          app.listen(80, () => console.log("Monolith listening on port
80"));
          APP
          pm2 start app.js --name monolith
          pm2 save
      Tags:
        - Key: Name
          Value: Legacy-Monolith

Outputs:
  MonolithPublicIP:
    Description: Public IP of the Monolith Instance
    Value: !GetAtt MonolithInstance.PublicIp
```

6. Post-Deployment Configuration

After deploying the foundational infrastructure via IaC, proceed with the manual steps below to complete the Strangler Fig pattern.

Phase 1: Deploying the API Gateway Facade

The first phase is to place an API Gateway in front of the legacy monolith. Initially, this gateway acts as a transparent proxy, routing 100% of the traffic directly to the monolith.

Step 1.1: Retrieve the Monolith Public IP

1. Navigate to the **CloudFormation Console** (or Terraform outputs).
2. Select the stack you deployed.
3. Click the **Outputs** tab.
4. Copy the value for `MonolithPublicIP`.

Step 1.2: Create the REST API

1. Navigate to the **API Gateway Console**.
2. Click **Create API**.
3. Under **REST API** (not Private, not HTTP API), click **Build**.
4. Choose **New API**.
5. **API name:** `Strangler-Fig-Facade`
6. **Endpoint Type:** Regional
7. Click **Create API**.

Step 1.3: Configure the Proxy Resource

1. In the Resources pane, select the root resource (`/`).
2. Click the **Actions** dropdown and select **Create Resource**.
3. Check the box for **Configure as proxy resource**. The Resource Path will automatically populate with `{proxy+}`.
4. Enable **API Gateway CORS** (optional but recommended for web clients).

5. Click **Create Resource**.

Step 1.4: Set Up the HTTP Proxy Integration

1. On the setup screen for the `ANY` method under `/{proxy+}`, configure the following:
   - **Integration type:** HTTP Proxy
   - **Endpoint URL:** `http://<MonolithPublicIP>/{proxy}` (Replace `<MonolithPublicIP>` with the IP you copied in Step 1.1).
2. Click **Save**.

Step 1.5: Deploy and Test the Facade

1. Click the **Actions** dropdown and select **Deploy API**.
2. **Deployment stage:** [New Stage]
3. **Stage name:** `v1`
4. Click **Deploy**.
5. At the top of the Stage Editor, copy the **Invoke URL**.
6. **Validation:** Open a browser and navigate to `https://<Invoke-URL>/products`. You should see the response: `"Response from Monolith: Here are the products!"`. Navigate to `https://<Invoke-URL>/inventory` to verify other endpoints are also routing correctly.

Phase 2: Serverless Extraction of the Users Service

With the facade in place, we intercept requests to the `/users` endpoint and route them to a newly built AWS Lambda function backed by Amazon DynamoDB.

Step 2.1: Create the Users Lambda Function

1. Navigate to the **AWS Lambda Console**.
2. Click **Create function**.
3. Choose **Author from scratch**.
4. **Function name:** `UsersMicroservice`

5. **Runtime:** Node.js 20.x
6. Under **Permissions**, expand **Change default execution role**.
7. Choose **Create a new role from AWS policy templates**.
8. **Role name:** `UsersMicroserviceRole`
9. **Policy templates:** Select **Simple microservice permissions** (this grants access to DynamoDB).
10. Click **Create function**.

Step 2.2: Implement the Microservice Logic

1. In the Lambda code editor, replace the default code with the following snippet. This code demonstrates interacting with the DynamoDB table created by your CloudFormation stack.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, PutCommand, ScanCommand } from "@aws-
sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const dynamo = DynamoDBDocumentClient.from(client);
const TABLE_NAME = "Users";

export const handler = async (event) => {
    const httpMethod = event.httpMethod;

    try {
        if (httpMethod === 'GET') {
            const response = await dynamo.send(new ScanCommand({ TableName:
TABLE_NAME }));
            return {
                statusCode: 200,
                body: JSON.stringify({
                    message: "Successfully retrieved users from DynamoDB
Microservice!",
                    users: response.Items
                })
            };
        } else if (httpMethod === 'POST') {
            const body = JSON.parse(event.body);
            const userId = body.userId || Date.now().toString();

            await dynamo.send(new PutCommand({
                TableName: TABLE_NAME,
                Item: {
                    UserID: userId,
                    Name: body.name || "Unknown",
                    Email: body.email || "unknown@example.com"
                }
            }));

            return {
                statusCode: 201,
                body: JSON.stringify({ message: `User ${userId} created
successfully in DynamoDB.` })
            };
        }

        return { statusCode: 405, body: "Method Not Allowed" };
```

```
    } catch (err) {
        console.error(err);
        return { statusCode: 500, body: JSON.stringify({ error: err.message
}) };
    }
};
```

1. Click **Deploy** to save the code.

Step 2.3: Route API Gateway to the Lambda Function

1. Return to the **API Gateway Console** and select the `Strangler-Fig-Facade` API.
2. Select the root resource (`/`).
3. Click **Actions** -> **Create Resource**.
4. **Resource Name:** `users` (Do **NOT** check “Configure as proxy resource”).
5. Click **Create Resource**.
6. Select the newly created `/users` resource.
7. Click **Actions** -> **Create Method**.
8. Select **ANY** from the dropdown and click the checkmark.
9. Configure the integration:
    - **Integration type:** Lambda Function
    - **Use Lambda Proxy integration:** Check this box (Required for the event object in the code to populate correctly).
    - **Lambda Region:** Select your region (e.g., `us-east-1`).
    - **Lambda Function:** Type `UsersMicroservice` and select it.
10. Click **Save** and grant API Gateway permission to invoke the Lambda function.

Step 2.4: Deploy and Validate Phase 2

1. Click **Actions** -> **Deploy API**.
2. Select the `v1` stage and click **Deploy**.
3. **Validation 1 (New Microservice):** Navigate to `https://<Invoke-URL>/users` . You should see the response from the new Lambda function, confirming the extraction was successful.

4. **Validation 2 (Legacy Monolith):** Navigate to `https://<Invoke-URL>/products`. You should still see the response from the EC2 monolith. The Strangler Fig is actively splitting traffic.

Phase 3: Containerized Extraction & Event-Driven Architecture

For services that require long-running processes or complex dependencies, serverless Lambda functions may not be appropriate. In this phase, we extract the `/orders` endpoint into a containerized microservice running on AWS Fargate.

Step 3.1: Create an Internal Application Load Balancer (ALB)

Because Fargate tasks are ephemeral, API Gateway cannot route to them by IP address. We must place them behind an internal ALB.

1. Navigate to the **EC2 Console** -> **Load Balancers**.
2. Click **Create Load Balancer** -> **Application Load Balancer**.
3. **Name:** `Orders-Microservice-ALB`
4. **Scheme:** Internal (Do not expose the microservice directly to the internet).
5. **VPC:** Select the default VPC (or the VPC where your Fargate cluster will reside).
6. **Mappings:** Select **ALL available Availability Zones** in your VPC. *Crucial Step: If you do not select all AZs, and Fargate launches a task in an unselected AZ, the ALB target will remain "Unused" and health checks will fail.*
7. **Security Groups:** Create a new security group allowing inbound HTTP (port 80) from the VPC CIDR block.
8. **Listeners and Routing:** Create a new Target Group named `orders-tg`.
    - Target type: **IP addresses** (Required for Fargate `awsvpc` network mode).
    - Protocol: HTTP / Port 80.
    - Health check path: `/orders` (Must match a valid route in your app, not just `/`).
9. Complete the creation of the ALB.

Step 3.2: Build and Push the Docker Image (Using AWS CloudShell)

Important Note: AWS Fargate requires images built for the `linux/amd64` architecture. If you build the image on an Apple Silicon Mac (ARM64), the Fargate task will fail to start (Exec format error). To avoid this, we will use AWS CloudShell, which natively builds for AMD64.

1. Navigate to the **Amazon ECR Console** and create a new private repository named `orders-microservice`.
2. Open **AWS CloudShell** from the top navigation bar.
3. In CloudShell, create the application files:

```
mkdir orders-app && cd orders-app
npm init -y
npm install express @aws-sdk/client-sns
```

4. Create `app.js`:

```
cat << 'EOF' > app.js
const express = require('express');
const { SNSClient, PublishCommand } = require('@aws-sdk/client-sns');
const app = express();
app.use(express.json());
const sns = new SNSClient({ region: 'us-east-1' });

app.post('/orders', async (req, res) => {
   try {
       const orderId = Date.now().toString();
       // In a real app, you'd save to a database here
       await sns.send(new PublishCommand({
           TopicArn: process.env.SNS_TOPIC_ARN,
           Message: JSON.stringify({ orderId: orderId, status:
'CREATED', amount: req.body.amount || 0 })
       }));
       res.status(201).json({ message: `Order ${orderId} created and
event published.` });
   } catch (err) {
       res.status(500).json({ error: err.message });
   }
});

app.get('/orders', (req, res) => {
   res.status(200).json({ message: "Orders service is running!" });
});

app.listen(80, () => console.log('Orders service listening on port
80'));
EOF
```

5. Create `Dockerfile`:

```
cat << 'EOF' > Dockerfile
FROM node:20-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 80
CMD ["node", "app.js"]
EOF
```

6. Authenticate Docker with ECR (Replace `<ACCOUNT_ID>` with your AWS Account ID):

```
aws ecr get-login-password --region us-east-1 | docker login --
username AWS --password-stdin <ACCOUNT_ID>.dkr.ecr.us-east-
1.amazonaws.com
```

7. Build and push the image:

```
docker build -t orders-microservice .
docker tag orders-microservice:latest <ACCOUNT_ID>.dkr.ecr.us-east-
1.amazonaws.com/orders-microservice:latest
docker push <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/orders-
microservice:latest
```

Step 3.3: Deploy the Fargate Service

1. Navigate to the **IAM Console** and create a new role for the ECS Task.
   - Trusted entity: **Elastic Container Service Task**
   - Attach policies: `AmazonSNSFullAccess` (so the container can publish events).
   - Role name: `OrdersTaskRole`
2. Navigate to the **Amazon ECS Console**.

3. Create a new **Task Definition**:
    - Launch type: AWS Fargate.
    - Task execution role: `ecsTaskExecutionRole` (creates automatically if missing, needed to pull ECR image).
    - Task role: `OrdersTaskRole` (the one you just created).
    - Container definition: Point to your ECR image URI (`<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/orders-microservice:latest`), map port 80.
    - Environment variables: Add `SNS_TOPIC_ARN` and set it to the ARN of your `OrderCreated` SNS topic.

4. **Create a dedicated Security Group for Fargate tasks:**
    - Navigate to the **EC2 Console -> Security Groups**.
    - Create a new security group named `orders-fargate-sg`.
    - **Inbound Rules:** Allow HTTP (Port 80) from the `Orders-Microservice-ALB` security group.
    - **Outbound Rules:** Ensure there is a rule allowing **All traffic** (or at least **HTTPS Port 443**) to `0.0.0.0/0`. *Crucial Step: Without this outbound rule, the Fargate task will fail to pull the image from ECR because it cannot reach the ECR API over the internet.*

5. Create a new **ECS Cluster** named `Modernization-Cluster`.

6. Create a new **Service** within the cluster:
    - Launch type: Fargate.
    - Task definition: Select the one you just created.
    - Desired tasks: 2.
    - Networking: Select the same VPC and subnets as the ALB. Assign the `orders-fargate-sg` you just created.
    - **Crucial Step:** Ensure **Auto-assign public IP** is **ENABLED**. Fargate tasks need a public IP to pull images from ECR unless you have configured VPC Endpoints.
    - Load balancing: Select Application Load Balancer. Choose `Orders-Microservice-ALB` and the `orders-tg` target group.

7. Wait for the service to reach a steady state and the targets to become healthy in the ALB.

Step 3.4: Create a VPC Link V2

To avoid the complexities of legacy REST API VPC Links (which require an NLB in front of your ALB), we will use an HTTP API with a VPC Link V2, which connects directly to the ALB.

1. Navigate to the **API Gateway Console** -> **VPC Links** (in the left navigation pane).
2. Click **Create**.
3. Choose **VPC link for HTTP APIs**.
4. **Name:** `Orders-VPCLink-V2`
5. **VPC:** Select your VPC.
6. **Subnets:** Select all subnets (the same ones assigned to your ALB).
7. **Security groups:** Select the `orders-fargate-sg` or default SG.
8. Click **Create**.

Step 3.5: Create an HTTP API and Route to Fargate

1. Go to **API Gateway -> Create API -> HTTP API -> Build**.
2. **Name:** `Strangler-Fig-HTTP`. Click **Next** and create it without initial integrations.
3. In the new API, go to **Integrations -> Create**.
4. **Integration type:** Private resource.
5. **Attach to VPC Link:** Select `Orders-VPCLink-V2`.
6. **Target service:** ALB/NLB -> Select `Orders-Microservice-ALB`.
7. **Listener:** HTTP 80. Click **Create**.
8. Go to **Routes -> Create route**.
9. **Method:** GET, **Path:** `/orders`.
10. Attach the integration you just created to this route.

Step 3.6: Deploy and Validate Phase 3

1. HTTP APIs deploy automatically to the `$default` stage by default.
2. Go to **Stages** and select the `$default` stage to find the Invoke URL.
3. **Validation:** Navigate to `https://<Invoke-URL>/orders` (using the HTTP API URL, not the REST API URL). You should receive a response from the Fargate container. *(Note: If you deploy to a named stage like `v1`, you must include it in the path: `/v1/orders`. Using `$default` is recommended for simplicity).*
4. **Event Validation:** If your container code successfully publishes to the `OrderCreated` SNS topic, navigate to the **Amazon SQS Console**. Check the `NotificationsQueue`. You should see messages available in the queue, proving that the event-driven architecture is functioning correctly and decoupling the orders service from downstream notification systems.

---

7. Conclusion & Cleanup

You have successfully implemented the Strangler Fig Pattern. The API Gateway facade now intelligently routes traffic:

- `/users` -> Serverless AWS Lambda (Modernized)
- `/orders` -> Containerized AWS Fargate (Modernized)
- `/*` (Everything else) -> EC2 Monolith (Legacy)

Over time, you will continue this process, creating new routes and microservices until the `/{proxy+}` route receives zero traffic, at which point the legacy EC2 instance can be safely decommissioned.

Cleanup Instructions: To avoid ongoing AWS charges, ensure you delete the resources in this order:

1. Delete the ECS Service and Cluster.
2. Delete the ALB and Target Groups.
3. Delete the API Gateway VPC Link.
4. Delete the API Gateway REST and HTTP APIs.

5. Delete the AWS Lambda function.
6. Finally, delete the CloudFormation stack to remove the foundational resources (VPC, EC2, DynamoDB, SNS, SQS).