PRJ-MLS-046: Bring Your Own Container with SageMaker

Certification: AWS Certified Machine Learning – Specialty
Domain: Advanced ML Concepts & Customization

---

1. Project Overview

This project demonstrates one of the most powerful and flexible features of Amazon SageMaker: **Bring Your Own Container (BYOC)**. While SageMaker provides managed, pre-built containers for popular frameworks like TensorFlow, PyTorch, and XGBoost, there are many scenarios where you need more control. You might need to use a custom or proprietary algorithm, a specific version of a library not available in the standard containers, or include custom dependencies and pre-compiled code.

The BYOC approach allows you to package your own algorithm and environment into a **Docker container**. SageMaker can then use this custom container for both training and inference. This gives you complete control over your execution environment while still benefiting from SageMaker's managed infrastructure, scalability, and MLOps features like automated tuning and workflow orchestration. We will build a simple scikit-learn algorithm, package it into a Docker container, push it to Amazon ECR, and then use it for a full train-and-deploy lifecycle in SageMaker.

Key Objectives

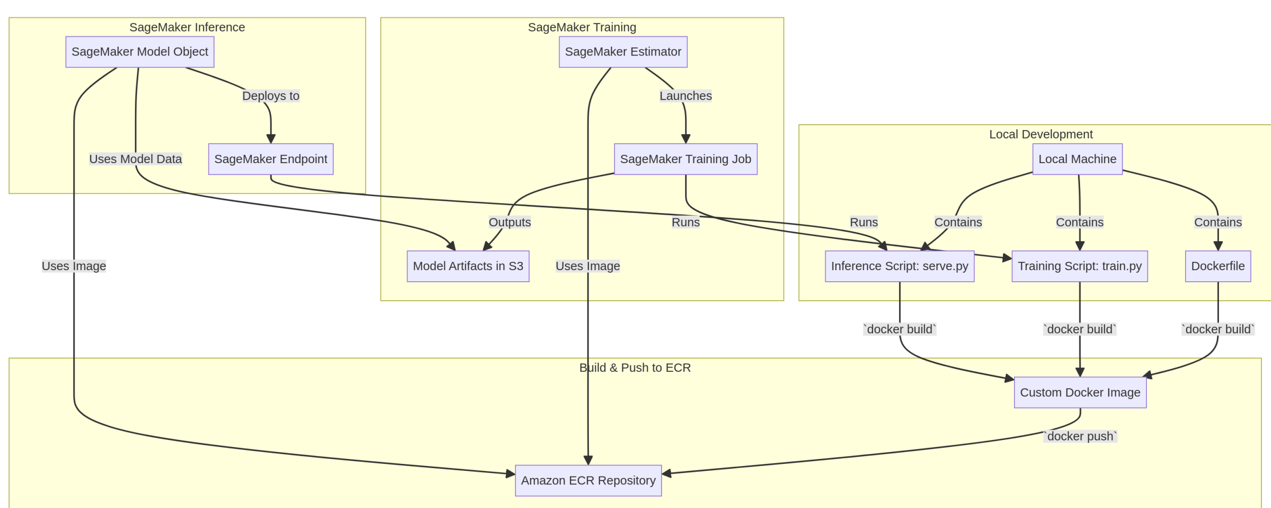
- Understand the SageMaker container contract for training and inference.
- Create a `Dockerfile` that defines the environment for a custom algorithm.
- Write a training script that SageMaker can execute within the container.
- Write an inference script (a simple web server) to serve predictions from the trained model.

- Build the Docker image and push it to the Amazon Elastic Container Registry (ECR).
- Launch a SageMaker Training Job using the custom container.
- Deploy the resulting model to a SageMaker Endpoint for real-time inference.

---

2. Architecture

The BYOC workflow involves building a container locally, pushing it to a registry, and then referencing it from SageMaker.



BYOC Workflow:

1. **Local Development:**
    - On a local machine (or a SageMaker Notebook with Docker installed), you create the necessary files:
        - `Dockerfile`: Defines the base image, dependencies, and copies the algorithm code into the image.
        - `train`: The executable script that SageMaker will run to start the training process. It reads hyperparameters and data locations from environment variables.
        - `serve`: The executable script that starts a simple web server (like Flask or Gunicorn) to handle inference requests.
2. **Build and Push:**

    - The `docker build` command is used to create the container image based on the `Dockerfile`.
    - The image is then tagged with the address of an **Amazon ECR repository**.
    - The `docker push` command uploads the container image to ECR, making it accessible to SageMaker.
3. **SageMaker Training:**
    - A SageMaker `Estimator` is configured with the ECR path of the custom container image.
    - When the training job is launched, SageMaker provisions the requested EC2 instances and pulls the custom container image from ECR.
    - It then runs the `train` script inside the container, passing hyperparameters and data locations as environment variables.
    - The `train` script saves the final model artifacts to a specific directory (`/opt/ml/model`), which SageMaker automatically archives and uploads to S3.
4. **SageMaker Inference:**
    - A SageMaker `Model` object is created, pointing to both the ECR image path and the S3 path of the trained model artifacts.
    - When this model is deployed to a **SageMaker Endpoint**, SageMaker provisions instances, pulls the custom container, and downloads the model artifacts from S3 into the container's `/opt/ml/model` directory.
    - It then runs the `serve` script, which starts the inference server. The server loads the model from disk and listens for prediction requests on a specific port.

---

3. Prerequisites

- An AWS account with administrative permissions.
- Docker installed and configured on your local machine or SageMaker Notebook instance.

- The AWS CLI installed and configured.
- An ECR repository to store the container image.

---

4. Step-by-Step Implementation Guide

We will package a simple scikit-learn decision tree classifier.

Step 4.1: The Directory Structure

Create a directory with the following structure:

```
/my_custom_algorithm
|-- Dockerfile
|-- container
    |-- train
    |-- serve
    |-- requirements.txt
```

Step 4.2: The Dockerfile

This file defines our container. It installs dependencies from `requirements.txt` and makes our `train` and `serve` scripts executable.

`Dockerfile`:

```
FROM python:3.8-slim-buster

# Set up the environment
WORKDIR /opt/ml/code
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Install dependencies
COPY container/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy the algorithm code
COPY container/ .

# Make scripts executable
RUN chmod +x train serve

# Define the entrypoint for training
ENTRYPOINT ["python", "train"]
```

`container/requirements.txt`:

```
scikit-learn
pandas
flask
gunicorn
```

Step 4.3: The Training Script

This script is run by SageMaker for training. It reads data from `/opt/ml/input/data`, trains the model, and saves it to `/opt/ml/model`.

`container/train`:

```
import os
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
import joblib

if __name__ == '__main__':
    # SageMaker environment variables
    training_data_path = os.environ.get('SM_CHANNEL_TRAINING')
    model_path = os.environ.get('SM_MODEL_DIR')

    # Load data
    data = pd.read_csv(os.path.join(training_data_path,
'training_data.csv'))
    X = data.iloc[:, :-1]
    y = data.iloc[:, -1]

    # Train model
    model = DecisionTreeClassifier()
    model.fit(X, y)

    # Save model
    joblib.dump(model, os.path.join(model_path, "model.joblib"))
```

Step 4.4: The Inference Script

This script starts a Flask web server. SageMaker sends POST requests to the `/invocations` endpoint.

`container/serve`:

```python
import os
import json
import joblib
from flask import Flask, request, Response

app = Flask(__name__)

# Load model
model_path = os.environ.get('SM_MODEL_DIR')
model = joblib.load(os.path.join(model_path, "model.joblib"))

@app.route('/ping', methods=['GET'])
def ping():
    return Response(response='\n', status=200, mimetype='application/json')

@app.route('/invocations', methods=['POST'])
def invoke():
    data = request.get_json()['data']
    prediction = model.predict(data).tolist()
    return Response(response=json.dumps(prediction), status=200,
mimetype='application/json')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

Step 4.5: Build and Push the Container

From your terminal, in the `my_custom_algorithm` directory:

```
# 1. Define variables
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
REGION=us-east-1
REPO_NAME=my-custom-algorithm

# 2. Create ECR repository
aws ecr create-repository --repository-name $REPO_NAME

# 3. Log in to ECR
aws ecr get-login-password --region $REGION | docker login --username AWS --
password-stdin $ACCOUNT_ID.dkr.ecr.$REGION.amazonaws.com

# 4. Build the image
docker build -t $REPO_NAME .

# 5. Tag the image
docker tag $REPO_NAME:latest
$ACCOUNT_ID.dkr.ecr.$REGION.amazonaws.com/$REPO_NAME:latest

# 6. Push the image
docker push $ACCOUNT_ID.dkr.ecr.$REGION.amazonaws.com/$REPO_NAME:latest
```

Step 4.6: Train and Deploy in SageMaker

In a SageMaker Notebook:

```python
import sagemaker

role = sagemaker.get_execution_role()

# 1. Get the ECR image URI
account_id = boto3.client('sts').get_caller_identity().get('Account')
region = boto3.session.Session().region_name
image_uri = f"{account_id}.dkr.ecr.{region}.amazonaws.com/my-custom-algorithm:latest"

# 2. Create an Estimator
estimator = sagemaker.estimator.Estimator(
    image_uri=image_uri,
    role=role,
    instance_count=1,
    instance_type='ml.m5.large'
)

# 3. Start training
estimator.fit({'training': 's3://my-bucket/my-training-data/'})

# 4. Deploy the model
predictor = estimator.deploy(
    initial_instance_count=1,
    instance_type='ml.t2.medium'
)

# 5. Make a prediction
predictor.predict(data={'data': [[1, 2, 3, 4]]})
```

---

5. Key SageMaker Environment Variables

- `SM_MODEL_DIR` (/opt/ml/model): Where your training script must save the model. For inference, this is where SageMaker loads the model from.
- `SM_CHANNEL_{channel_name}` (/opt/ml/input/data/{channel_name}): The directory where input data for a specific channel is located.
- `SM_HP_{hyperparameter_name}`: The value of a hyperparameter.
- `SM_NUM_GPUS`: The number of GPUs available on the instance.

---

6. Cleanup

1. **Delete the Endpoint:** `predictor.delete_endpoint()`
2. **Delete the ECR Repository:** Go to the ECR console and delete the `my-custom-algorithm` repository.
3. **Clean up S3 Artifacts:** Delete the model artifacts from the S3 output bucket.

Business Context

The Problem

Organizations want to leverage AI/ML for business insights but lack the expertise to build secure, scalable ML systems. ML projects fail due to poor data quality, model drift, and lack of monitoring. Security and compliance requirements for ML systems are often overlooked.

The Solution

Production-ready ML system with automated data pipelines, model training, deployment, and monitoring. Implements MLOps best practices with version control, A/B testing, and automated retraining. Ensures data privacy and model security throughout the ML lifecycle.

Business Value

- **Business Intelligence:** AI-driven insights improve decision-making and outcomes
- **Operational Efficiency:** Automated ML workflows reduce manual data science effort by 60%
- **Model Reliability:** Continuous monitoring detects and corrects model drift automatically
- **Compliance Assurance:** Built-in controls for data privacy and model governance

Risk Mitigation

Protects sensitive data in ML pipelines, prevents biased or inaccurate models, ensures model explainability, and maintains compliance with AI regulations.

GRC Mapping

Compliance Frameworks

- **NIST AI RMF:** Trustworthy and responsible AI
- **ISO/IEC 23894:** AI risk management
- **NIST CSF:** PR.DS-2 (Data in transit), PR.DS-5 (Data leak protection)
- **Responsible AI Principles:** Fairness, accountability, transparency, ethics

Security Controls Implemented

- Data anonymization and pseudonymization
- Model explainability and interpretability
- Bias detection and mitigation
- Model versioning and rollback
- Automated model monitoring and alerting

Audit Evidence

- Model cards documenting intended use and limitations
- Bias and fairness evaluation reports
- Model performance metrics over time
- Data processing and transformation logs

Regulatory Alignment

- **GDPR:** Article 22 (Right to explanation), Article 25 (Privacy by design)
- **AI Act (EU):** Transparency and accountability requirements

- **CCPA:** Data privacy for ML training data
- **SOC 2:** CC6.1 (Access to sensitive data)