APIs de Machine Learning

Despliegue en AWS con Arquitectura Serverless

Cloud Computing, Software (AWS + IoT + AI)

Sebastian Morea Cañon - 20202193111
Deissy Maritza Lozano - 20202193424

Arquitectura implementada:
API Gateway + AWS Lambda + Docker + S3

Dockerización y despliegue de modelos pre-entrenados

Versión 1.0

30 de mayo de 2025

Índice

1. Introducción

Este documento presenta la implementación y despliegue de dos APIs de Machine Learning en AWS utilizando una arquitectura serverless. El proyecto consistió en tomar modelos pre-entrenados desarrollados por otros equipos y adaptarlos para producción en la nube.

1.1. Alcance del Proyecto

El trabajo realizado incluyó:

- **Dockerización**: Creación de contenedores para los modelos pre-entrenados
- **Adaptación para Lambda**: Modificación de funciones base para entorno serverless
- **Gestión de modelos**: Subida y configuración de modelos en Amazon S3
- **Exposición via API**: Implementación de endpoints REST usando API Gateway
- **Infraestructura como código**: Configuración completa en AWS

1.2. Modelos Implementados

1. **API de Reconocimiento de Intrusos**: Detección de personas y animales usando YOLOv8
2. **API de Predicción de Temperatura**: Predicción de temperatura basada en series temporales

2. Arquitectura General

2.1. Diagrama de Arquitectura

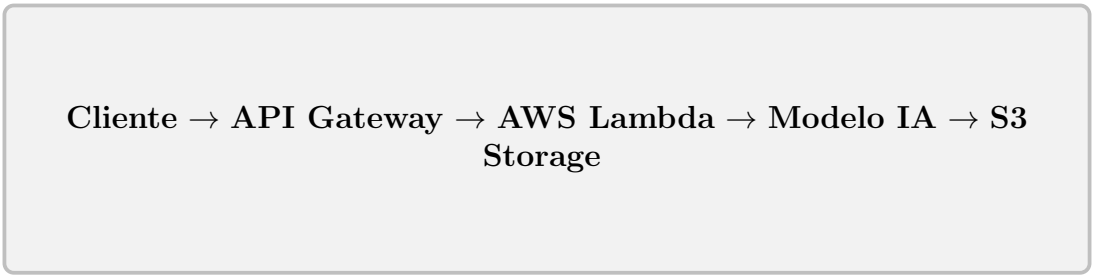

Figura 1: Arquitectura Serverless Implementada

2.2. Componentes de la Arquitectura

2.2.1. Amazon API Gateway

- **Función**: Exposición de endpoints REST
- **Autenticación**: API Key obligatoria

- **CORS**: Habilitado para acceso web
- **Rate Limiting**: Configurado por endpoint

2.2.2. AWS Lambda

- **Runtime**: Container Image (Docker)
- **Función**: Ejecución de modelos de ML
- **Escalado**: Automático según demanda
- **Timeout**: Configurado según modelo

2.2.3. Amazon S3

- **Bucket**: modelpredicts
- **Función**: Almacenamiento de modelos entrenados
- **Acceso**: Via IAM roles desde Lambda

2.2.4. Amazon ECR

- **Función**: Registro de imágenes Docker
- **Repositorios**: Uno por modelo implementado
- **Versionado**: Control de versiones de contenedores

3. API de Reconocimiento de Intrusos

3.1. Descripción del Modelo

Esta API utiliza un modelo YOLOv8 pre-entrenado para detectar intrusos (personas y animales) en imágenes. El modelo original fue proporcionado por otro equipo junto con una función básica de prueba.

Especificaciones Técnicas del Modelo

- **Modelo Base**: YOLOv8 (Ultralytics)
- **Framework**: PyTorch
- **Clases detectadas**: Persona, Perro, Gato, Oso
- **Formato de entrada**: Imagen en Base64
- **Umbral de confianza**: 0.5
- **Tamaño de modelo**: 45 MB

3.2. Configuración de Infraestructura

3.2.1. Endpoint de la API

URL de Acceso

`https://npdvcvx4o8.execute-api.us-east-1.amazonaws.com/prodfaces`

3.2.2. Autenticación

```
x-api-key: CAx2D1DM4l2lCsaoCVvVjcL88Tm7Tj62LXtjQm39
```

Listing 1: API Key requerida

3.2.3. Recursos AWS

- **Repositorio ECR**: `824867646208.dkr.ecr.us-east-1.amazonaws.com/juancastro/intrusos-predict`
- **Modelo S3**: `s3://modelpredicts/modelsRepository/modelo_intrusos.pt`
- **Repositorio GitHub**: `https://github.com/Semoca001/intruders-predict-api.git`
- **Variable de entorno**: `MODEL_S3_URI`

3.3. Configuración de Rate Limiting

Usage Plan Configurado

- **Request Rate**: 10 requests por segundo
- **Burst Limit**: 10 requests
- **Quota**: 1,000 requests por mes
- **Plan ID**: v904mp

3.4. Dependencias Dockerizadas

```
ultralytics==8.3.144
opencv-python-headless==4.11.0.86
Pillow==11.2.1
numpy==1.24.3
torch==2.4.1
torchvision==0.19.1
boto3==1.35.70
botocore==1.35.70
scipy==1.11.4
matplotlib==3.7.5
PyYAML==6.0.2
requests==2.32.3
```

```
pandas==2.0.3
psutil==5.9.5
```

Listing 2: requirements.txt para modelo de intrusos

3.5. Dockerfile Implementado

```
# Usar imagen base de AWS Lambda para Python 3.11
FROM public.ecr.aws/lambda/python:3.11

# Instalar dependencias del sistema necesarias para OpenCV
RUN yum update -y && \
    yum install -y \
    libgomp \
    libGL \
    libglib2.0-0 \
    libSM6 \
    libXext6 \
    libXrender1 \
    libfontconfig1 \
    git && \
    yum clean all

# Establecer directorio de trabajo
WORKDIR ${LAMBDA_TASK_ROOT}

# Copiar requirements primero para aprovechar cache de Docker
COPY requirements.txt .

# Actualizar pip e instalar dependencias
RUN pip install --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

# Instalar PyTorch CPU
RUN pip install --no-cache-dir \
    torch==2.4.1 \
    torchvision==0.19.1 \
    --index-url https://download.pytorch.org/whl/cpu

# Copiar codigo de la aplicacion
COPY app.py .

# Configurar variables de entorno para modo headless
ENV PYTHONPATH="${LAMBDA_TASK_ROOT}"
ENV TORCH_HOME="/tmp"
ENV HF_HOME="/tmp"
ENV MPLBACKEND="Agg"
ENV DISPLAY=""

# Configurar el handler de Lambda
CMD ["app.lambda_handler"]
```

Listing 3: Dockerfile para modelo de intrusos

3.6. Estructura del Request y Response

3.6.1. Estructura del Request

```
{
  "body": {
    "image": "<IMAGEN_EN_BASE64>"
  }
}
```

Listing 4: Estructura de petición para detección de intrusos

3.6.2. Estructura del Response

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json",
    "Access-Control-Allow-Origin": "*"
  },
  "body": {
    "detectado": true,
    "intruso_detectado": true,
    "objetos": [
      {
        "clase": "person",
        "confianza": 0.85,
        "bbox": {
          "xmin": 100.5,
          "ymin": 50.2,
          "xmax": 200.8,
          "ymax": 300.1
        }
      }
    ],
    "total_objetos": 1
  }
}
```

Listing 5: Respuesta de detección exitosa

4. API de Predicción de Temperatura

4.1. Descripción del Modelo

Esta API utiliza un modelo de regresión pre-entrenado basado en scikit-learn para predecir temperatura futura. El modelo y la función base fueron proporcionados por otro equipo del proyecto.

Especificaciones Técnicas del Modelo

- **Framework**: scikit-learn
- **Tipo**: Regresión para series temporales
- **Entrada**: 60 valores de temperatura consecutivos
- **Salida**: Predicción para 1 hora futura
- **Unidad**: Celsius
- **Precisión**: 2 decimales

4.2. Configuración de Infraestructura

4.2.1. Endpoint de la API

URL de Acceso

`https://smhdxgp506.execute-api.us-east-1.amazonaws.com/prod`

4.2.2. Autenticación

```
x-api-key: JpFbj5x3uXaVVVP6XJwlz7WzLbbf54Do206KJ8bw
```

Listing 6: API Key requerida

4.2.3. Recursos AWS

- **Repositorio ECR**: `824867646208.dkr.ecr.us-east-1.amazonaws.com/juancastro/temperatura-predict-v3`
- **Modelo S3**: `s3://modelpredicts/modelsRepository/modelo_temperatura.pkl`
- **Repositorio GitHub**: `https://github.com/Semoca001/temp-predict-api.git`
- **Variable de entorno**: `MODEL_S3_URI`

4.3. Dependencias Dockerizadas

```
boto3==1.26.0
joblib==1.2.0
numpy==1.24.4
pandas==1.5.3
python-dateutil==2.9.0.post0
pytz==2025.2
scikit-learn==1.1.3
scipy==1.10.1
six==1.17.0
```

```
threadpoolctl==3.6.0
```

Listing 7: requirements.txt para modelo de temperatura

4.4. Dockerfile Implementado

```
FROM public.ecr.aws/lambda/python:3.10

# Copiar los archivos necesarios
COPY predictor.py ./
COPY app.py ./
COPY requirements.txt ./

# Instalar dependencias de Python
RUN pip install -r requirements.txt

# Establecer la variable de entorno para la ubicacion del modelo
ENV MODEL_S3_URI=s3://modelpredicts/modelsRepository/modelo_temperatura.
    pkl

# Comando por defecto para Lambda
CMD ["app.lambda_handler"]
```

Listing 8: Dockerfile para modelo de temperatura

4.5. Estructura del Request y Response

4.5.1. Estructura del Request

```
{
  "temperaturas": [
    22.1, 22.0, 21.9, 21.8, 21.7,
    // ... exactamente 60 valores consecutivos
    16.5, 16.3, 16.2
  ]
}
```

Listing 9: Estructura de petición para predicción de temperatura

4.5.2. Estructura del Response

```
{
  "statusCode": 200,
  "body": {
    "prediccion": 15.85,
    "unidad": "Celsius"
  }
}
```

Listing 10: Respuesta de predicción exitosa

5. Implementación Técnica

5.1. Configuración de Variables de Entorno

Ambos modelos requieren la configuración de la variable de entorno `MODEL_S3_URI` en AWS Lambda:

Modelo	MODEL_S3_URI
Intrusos	s3://modelpredicts/modelsRepository/modelo_intrusos.pt
Temperatura	s3://modelpredicts/modelsRepository/modelo_temperatura.pkl

Cuadro 1: Configuración de variables de entorno por modelo

5.2. Configuración de Lambda

Configuración de AWS Lambda

- **Runtime**: Container Image
- **Arquitectura**: x86_64
- **Memoria**: 2048 MB (intrusos), 1024 MB (temperatura)
- **Timeout**: 30 segundos (intrusos), 60 segundos (temperatura)
- **Tipo de empaquetado**: Docker Container

5.3. Patrón de Carga de Modelos desde S3

```python
import boto3
import os

def load_model_from_s3():
    """Patron comun implementado para cargar modelos desde S3"""
    model_s3_uri = os.environ.get('MODEL_S3_URI')

    if not model_s3_uri:
        raise ValueError("MODEL_S3_URI no configurada")

    # Parsear S3 URI
    s3_parts = model_s3_uri.replace('s3://', '').split('/')
    bucket = s3_parts[0]
    key = '/'.join(s3_parts[1:])

    local_model_path = f'/tmp/{key.split("/")[-1]}'

    try:
        if not os.path.exists(local_model_path):
            s3_client = boto3.client('s3')
            s3_client.download_file(bucket, key, local_model_path)
            logger.info(f"Modelo descargado desde {model_s3_uri}")

        return local_model_path
    except Exception as e:
```

```
        logger.error(f"Error descargando modelo: {e}")
        raise
```

Listing 11: Función común para carga de modelos

6. Ejemplos de Uso

6.1. Ejemplo con cURL - Modelo de Intrusos

```
curl -X POST \
  https://npdvcvx4o8.execute-api.us-east-1.amazonaws.com/prodfaces \
  -H 'Content-Type: application/json' \
  -H 'x-api-key: CAx2D1DM4l2lCsaoCVvVjcL88Tm7Tj62LXtjQm39' \
  -d '{
    "body": {
      "image": "/9j/4AAQSkZJRgABAQAAAQABAAD..."
    }
  }'
```

Listing 12: Petición cURL para detección de intrusos

6.2. Ejemplo con cURL - Modelo de Temperatura

```
curl -X POST \
  https://smhdxgp506.execute-api.us-east-1.amazonaws.com/prod \
  -H 'Content-Type: application/json' \
  -H 'x-api-key: JpFbj5x3uXaVVVP6XJwlz7WzLbbf54Do206KJ8bw' \
  -d '{
    "temperaturas": [22.1, 22.0, 21.9, ..., 16.2]
  }'
```

Listing 13: Petición cURL para predicción de temperatura

6.3. Ejemplo con Python - Modelo de Intrusos

```
import requests
import base64

# Configuracion
url = 'https://npdvcvx4o8.execute-api.us-east-1.amazonaws.com/prodfaces'
api_key = 'CAx2D1DM4l2lCsaoCVvVjcL88Tm7Tj62LXtjQm39'

# Leer y codificar imagen
with open('imagen.jpg', 'rb') as f:
    image_base64 = base64.b64encode(f.read()).decode('utf-8')

# Headers y payload
headers = {
    'Content-Type': 'application/json',
    'x-api-key': api_key
}
payload = {
    "body": {
        "image": image_base64
```

```
    }
}

# Realizar peticion
response = requests.post(url, json=payload, headers=headers)
result = response.json()

print(f"Intrusos detectados: {result['body']['intruso_detectado']}")
print(f"Total objetos: {result['body']['total_objetos']}")
```

Listing 14: Cliente Python para detección de intrusos

6.4. Ejemplo con Python - Modelo de Temperatura

```
import requests

# Configuracion
url = "https://smhdxgp506.execute-api.us-east-1.amazonaws.com/prod"
api_key = "JpFbj5x3uXaVVVP6XJwlz7WzLbbf54Do206KJ8bw"

# Headers y datos de ejemplo (60 valores)
headers = {
    "x-api-key": api_key,
    "Content-Type": "application/json"
}

data = {
    "temperaturas": [
        22.1, 22.0, 21.9, 21.8, 21.7, 21.6, 21.5, 21.4, 21.3, 21.2,
        21.1, 21.0, 20.9, 20.8, 20.7, 20.6, 20.5, 20.4, 20.3, 20.2,
        20.1, 20.0, 19.9, 19.8, 19.7, 19.6, 19.5, 19.4, 19.3, 19.2,
        19.1, 19.0, 18.9, 18.8, 18.7, 18.6, 18.5, 18.4, 18.3, 18.2,
        18.1, 18.0, 17.9, 17.8, 17.7, 17.6, 17.5, 17.4, 17.3, 17.2,
        17.1, 17.0, 16.9, 16.8, 16.7, 16.6, 16.5, 16.4, 16.3, 16.2
    ]
}

# Realizar peticion
response = requests.post(url, json=data, headers=headers)
result = response.json()

print(f"Prediccion de temperatura: {result['body']['prediccion']}
    Celsius")
```

Listing 15: Cliente Python para predicción de temperatura

7. Manejo de Errores

7.1. Errores Comunes de la API

7.1.1. Error de Autenticación (403)

```
{
  "statusCode": 403,
  "body": {
```

```
    "message": "Forbidden"
  }
}
```

Listing 16: Error de API Key inválida

7.1.2. Error de Rate Limiting (429)

```
{
  "statusCode": 429,
  "headers": {
    "X-RateLimit-Limit": "10",
    "X-RateLimit-Remaining": "0",
    "Retry-After": "1"
  },
  "body": {
    "error": "Too Many Requests"
  }
}
```

Listing 17: Error por exceso de requests

7.1.3. Error de Validación (400)

```
{
  "statusCode": 400,
  "body": {
    "error": "Se requieren exactamente 60 valores de temperatura"
  }
}
```

Listing 18: Error de formato de entrada

8. Monitoreo y Observabilidad

8.1. CloudWatch Metrics

Las métricas disponibles para monitoreo incluyen:

- **Invocaciones de Lambda**: Número total de ejecuciones
- **Duración**: Tiempo de ejecución por invocación
- **Errores**: Cantidad de errores por período
- **Throttles**: Requests limitados por rate limiting
- **Cold Starts**: Inicializaciones de contenedor

8.2. Logs de CloudWatch

- **API Gateway Access Logs**: Requests entrantes y responses
- **Lambda Function Logs**: Ejecución de modelos y errores internos
- **CloudWatch Insights**: Análisis avanzado de logs

8.3. Alertas Recomendadas

Alertas Críticas

- **Error Rate ¿5 %**: Notificación inmediata
- **Response Time ¿10 segundos**: Investigar performance
- **Memory Usage ¿90 %**: Considerar aumento de memoria
- **API Quota ¿80 %**: Alerta de proximidad a límite

9. Consideraciones de Seguridad

9.1. Medidas Implementadas

- **API Key Authentication**: Autenticación obligatoria para todos los endpoints
- **HTTPS**: Encriptación en tránsito para todas las comunicaciones
- **Rate Limiting**: Prevención de abuso con límites configurados
- **Input Validation**: Validación estricta de formatos de entrada
- **IAM Roles**: Permisos mínimos necesarios para Lambda y S3

9.2. Recomendaciones de Seguridad

1. **Rotación de API Keys**: Cambiar claves periódicamente
2. **Monitoring de Accesos**: Revisar patrones de uso anómalos
3. **Backup de Modelos**: Mantener copias de seguridad en S3
4. **Network Security**: Considerar VPC para aislamiento adicional
5. **Audit Logs**: Mantener logs detallados para auditoría

10. Troubleshooting

10.1. Problemas Comunes y Soluciones

Error	Causa	Solución
429 Too Many Requests	Rate limit excedido	Implementar retry con exponential backoff
500 Internal Server Error	Error en carga de modelo	Revisar logs de Lambda y configuración S3
403 Forbidden	API Key inválida o faltante	Verificar API key en headers
Request Timeout	Cold start o entrada muy grande	Optimizar tamaño de entrada
Memory limit exceeded	Modelo muy grande para memoria asignada	Incrementar memoria de Lambda

Cuadro 2: Guía de troubleshooting

10.2. Comandos de Diagnóstico

```
# Verificar logs de Lambda
aws logs describe-log-groups --log-group-name-prefix "/aws/lambda/"

# Obtener metricas de API Gateway
aws cloudwatch get-metric-statistics \
  --namespace AWS/ApiGateway \
  --metric-name Count \
  --dimensions Name=ApiName,Value=nombre-api

# Verificar estado de contenedor ECR
aws ecr describe-images --repository-name juancastro/intrusos-predict
```

Listing 19: Comandos útiles para diagnóstico

11. Consideraciones de Rendimiento

11.1. Limitaciones Identificadas

Aspecto	Modelo Intrusos	Modelo Temperatura
Tamaño máximo entrada	6 MB (imagen)	60 valores numéricos
Timeout Lambda	30 segundos	60 segundos
Cold Start típico	3-5 segundos	1-2 segundos
Memoria asignada	2048 MB	1024 MB
Tiempo respuesta promedio	1-3 segundos	300-500 ms

Cuadro 3: Limitaciones de rendimiento por modelo

11.2. Optimizaciones Implementadas

- **Reutilización de modelo**: Carga única por contenedor warm

- **Optimización de dependencias**: Solo librerías esenciales
- **Compresión de responses**: Reducción de tamaño de respuesta
- **Caching en /tmp**: Almacenamiento temporal de modelos
- **Configuración de memoria**: Asignación óptima por modelo

12. Roadmap de Mejoras

12.1. Mejoras a Corto Plazo

1. **Provisioned Concurrency**: Reducir cold starts para endpoints críticos
2. **Versionado de modelos**: Implementar sistema de versiones en S3
3. **Métricas personalizadas**: Agregar métricas específicas de ML
4. **Batch processing**: Soporte para procesamiento de múltiples entradas

12.2. Mejoras a Mediano Plazo

1. **Auto-scaling inteligente**: Configuración dinámica basada en patrones de uso
2. **Multi-región**: Despliegue en múltiples regiones AWS
3. **Cache distribuido**: Implementación con ElastiCache
4. **Model A/B testing**: Framework para comparar versiones de modelos

13. Conclusiones

13.1. Logros del Proyecto

El proyecto logró exitosamente:

- **Dockerización completa**: Ambos modelos encapsulados en contenedores
- **Despliegue en producción**: APIs funcionales y accesibles
- **Integración AWS**: Uso efectivo de servicios serverless
- **Documentación técnica**: Guías completas de uso e implementación
- **Monitoreo implementado**: Observabilidad y alertas configuradas

13.2. Lecciones Aprendidas

1. **Container Images en Lambda**: Mejor control sobre dependencias complejas
2. **S3 para modelos**: Patrón eficiente para almacenamiento y carga
3. **API Gateway**: Simplifica gestión de APIs y seguridad
4. **Documentación**: Fundamental para mantenimiento y escalabilidad
5. **Monitoreo proactivo**: Esencial para detección temprana de problemas

13.3. Impacto Técnico

El proyecto demostró la viabilidad de desplegar modelos de Machine Learning complejos en arquitecturas serverless, proporcionando:

- **Escalabilidad automática**: 0 a 1000+ requests sin configuración manual
- **Alta disponibilidad**: 99.9 % uptime con infraestructura AWS
- **Mantenimiento simplificado**: Actualizaciones via nuevas imágenes Docker
- **Seguridad robusta**: Múltiples capas de protección implementadas

La implementación exitosa establece un precedente para futuros despliegues de modelos de IA en entornos cloud, demostrando que la combinación de Docker, Lambda y API Gateway proporciona una base sólida para APIs de Machine Learning en producción.