Clase 201 — Serverless ML: AWS Lambda, GCP Cloud Functions

Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs Lambda container images + Cloud Functions 2nd gen. Duración estimada: 75 min.

Clase 201 — Serverless ML: AWS Lambda, GCP Cloud Functions

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs Lambda container images + Cloud Functions 2nd gen. Duración estimada: 75 min.*

Objetivo

Desplegar un modelo como función serverless que escala de 0 a N sin gestionar servidores. Decidir cuándo serverless gana (tráfico bursty, batch chico, no podés mantener infra) y cuándo pierde (cold start crítico, modelo >1 GB, latencia <50 ms requerida).

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Empaquetar un modelo sklearn/XGBoost como Lambda Container Image (hasta 10 GB) o Cloud Function 2nd gen (build automático con Buildpacks).
- Mitigar cold starts con: provisioned concurrency (Lambda), min-instances=1 (Cloud Functions), o snapshot-based init (SnapStart).
- Configurar API Gateway o HTTP trigger para exponer la función como REST.
- Calcular costo: $/M invocations × duration × memory, vs un pod K8s 24/7.
- Reconocer límites: timeout 15 min (Lambda), payload 6 MB sync / 256 KB async, disco efímero 512 MB /tmp (10 GB con ephemeral-storage).

Temas

#	Tema	Por qué importa
1	Modelo de ejecución: scale-to-zero, reques	Diferente a K8s/EC2 24/7.
2	Cold start vs warm	El bug que arruina la UX.
3	Package formats: ZIP (≤250 MB) vs Containe	Para modelos ML: container.
4	Cost model: invocations + GB-seconds	Cruce con K8s al ~1M req/día sostenido.
5	Provisioned concurrency / min-instances	Cambia la economía y la latencia.
6	Cuándo NO usar serverless	Modelos grandes, latencia <50 ms, stateful

Definiciones y características

- Cold start: tiempo entre llegada del request y arranque del runtime + carga del modelo en una instancia nueva. Lambda Python: ~300 ms (runtime) + tu init code. Con modelo grande: 5-15 s.
- Warm: instancia ya inicializada que recibe request → latencia = solo predict. Lambda mantiene warm ~5-15 min sin invocations (no garantizado).
- Provisioned Concurrency (Lambda): pagás por mantener N instancias warm permanentemente. Elimina cold starts a cambio de costo fijo.
- SnapStart (Lambda, Java/Python 3.12+): toma snapshot del runtime después de init y lo restaura —

cold start <500 ms aún con modelos grandes.
- Cloud Functions 2nd gen (basado en Cloud Run): timeouts hasta 60 min, hasta 32 GB RAM, mejor cold start que 1st gen, y la opción --min-instances=1 para mantener warm.
- API Gateway: pone HTTP delante de Lambda. Maneja auth, throttling, CORS. Cobra extra por request — para tráfico volumétrico, Lambda Function URL (gratis) es alternativa.
- Provisioned vs On-Demand: provisioned escala a 0 manteniendo N warm. On-demand escala a 0 puro. Trade-off costo vs latency tail.

Dataset / recursos

- Modelo: LogisticRegression sobre Iris (chico — buen fit serverless).
- Tools: AWS CLI + SAM (Serverless Application Model), o gcloud functions deploy.
- Imagen base Lambda: public.ecr.aws/lambda/python:3.12.

Ejercicios

1. Lambda con container: buildeá un Dockerfile basado en public.ecr.aws/lambda/python:3.12, copiá app.py con lambda_handler(event, context), y model.pkl. Push a ECR. aws lambda create-function con --package-type Image.
2. API Gateway: creá una API HTTP y conectala a la Lambda. curl <api-url>/predict -d '{"features":[5.1,3.5,1.4,0.2]}'. Medí latencia primera vs segunda llamada (cold vs warm).
3. Provisioned Concurrency: configurá provisioned-concurrency=1 en la Lambda. Vuelve a medir — debería eliminar el cold start.
4. Cloud Functions equivalent: gcloud functions deploy iris --gen2 --runtime=python312 --trigger-http --source=. --entry-point=predict --memory=512Mi --min-instances=1. Compará cold start con Lambda.
5. Cost calc: asumí 100 req/s sostenido, modelo en RAM 512 MB, 80 ms p99. Calculá $/mes en Lambda vs un pod K8s con 4 vCPU 24/7. Encontrá el punto de cruce.

Homework verificable

1. Lambda function (container image) que sirve predict con cold start <3 s y warm <80 ms.
2. API Gateway HTTP exponiendo la Lambda.
3. CloudWatch alarm que dispara si Errors > 5 en 5 min, o Duration P99 > 1000 ms.
4. Script python cost_analysis.py que dado req_per_second_mean, req_size, mem_mb, duration_ms calcula $/mes en (a) Lambda on-demand, (b) Lambda con PC=2, (c) ECS Fargate equivalente, (d) K8s con 3 pods t3.medium.
5. Recomendación escrita: para tu workload, qué opción elegir y por qué.

Criterio de aceptación: curl <api>/predict responde en <100 ms (warm) y <3 s (cold), el cost analysis identifica el punto de cruce correcto (típicamente serverless gana <~500k requests/día sostenido).

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Cold start de 30 s	Carga del modelo dentro del handler. Fix:
Task timed out after 3.00 seconds	Default Lambda timeout es 3 s. Fix: aws la
Imagen ZIP excede 250 MB descomprimido	Probablemente trayendo numpy+scipy+pandas.

Pérdida de estado entre invocations	Esperado: cada instancia es efímera. Fix:
Costos explotaron	(1) Provisioned concurrency olvidada. (2)
errorMessage: Unable to import module	Falta una dep o estructura incorrecta. Fix

Preguntas frecuentes

¿Cuándo serverless conviene para ML?

(1) Tráfico bursty con valles largos (1000 req/min picos, 0 req/min noche). (2) Modelo chico (<1 GB) y latencia tolerante (>200 ms p99 OK). (3) Equipo sin DevOps. (4) Predicciones batch ad-hoc (S3 trigger). Suma ~70% de los casos típicos en startups.

¿Cuándo NO?

(1) Latencia estricta <50 ms — cold start es matador. (2) Modelo grande (>3 GB) — init carísimo. (3) Throughput sostenido >1000 req/s 24/7 — sale más caro que K8s. (4) Stateful: caches calientes, sessions, websockets persistentes.

¿Lambda o SageMaker Serverless Inference?

SageMaker Serverless: built-in para ML (deploy con un click desde model registry, batch transform). Misma idea de scale-to-zero. Latencia similar. Pros: integración con SageMaker; cons: vendor-locked y más caro por request. Para ML puro en AWS: SageMaker. Para mezcla ML + lógica de negocio: Lambda.

¿GPU en serverless?

Lambda no tiene GPU. Cloud Run/Functions 2nd gen tampoco (al momento de escribir). Para inferencia GPU serverless: Modal, Banana, Replicate (3rd party SaaS). Sino: K8s con GPU nodes.

¿Cómo monitoreo cold start vs warm?

CloudWatch: métrica InitDuration (solo en cold), Duration (siempre). Lambda Insights agrega CPU/mem/network. Para correlacionar con latencia user-facing: X-Ray tracing.

¿Container Image vs Layer?

Lambda Layers: hasta 5 layers de 250 MB cada uno (compartidos entre funciones). Container Image: hasta 10 GB, todo en uno. Para ML moderno: Container Image siempre — más simple, sin límite efectivo.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 11.
- AWS Lambda container images for Python.
- Lambda SnapStart for Python.
- Cloud Functions 2nd gen.
- Modal — serverless con GPU para ML.

Siguiente clase

Clase 202 — Monitoreo: data drift, model drift, alertas

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
lambda_dockerfile = '''\
FROM public.ecr.aws/lambda/python:3.12

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py model.pkl ./

CMD ["app.lambda_handler"]
'''
print('# Dockerfile (Lambda)')
print(lambda_dockerfile)
```

Archivos complementarios

- notebook.ipynb