Del dato a producción

Si sientes que tienes las piezas regadas, este es el dibujo que las junta: el C01 (sugerir la categoría de una foto con una ResNet-50 servida por Triton en EKS) de punta a punta, desde la foto etiquetada hasta el request que responde en milisegundos y el monitoreo que pide el siguiente modelo. Para cada herramienta verás qué es y cómo se trabaja con ella: si se programa, se configura, se opera o se renta como servicio administrado.

1. Empieza por el mapa: 12 estaciones en 4 tramos y, entre cada par, el artefacto que viaja (snapshot, checkpoint, engine, imagen, request, logs).
2. Luego, las 4 velocidades: el mismo mapa partido en líneas que corren cada mes, en días, en milisegundos y sin parar. Buena parte de la confusión viene de mezclarlas.
3. Entra a cada estación cuando la necesites: proceso paso a paso, archivos reales del C01, comandos, qué falla ahí y cuándo está lista.
4. Cierra con el quiz: un síntoma real, y tú dices en qué estación falló y qué miras primero.

LEYENDA

semanas Línea del modelo · **días** Línea del release · **milisegundos** Línea del request · **continuo** Línea de operación

transbordo: ahí se tocan dos líneas · snapshot el artefacto que viaja a la siguiente estación

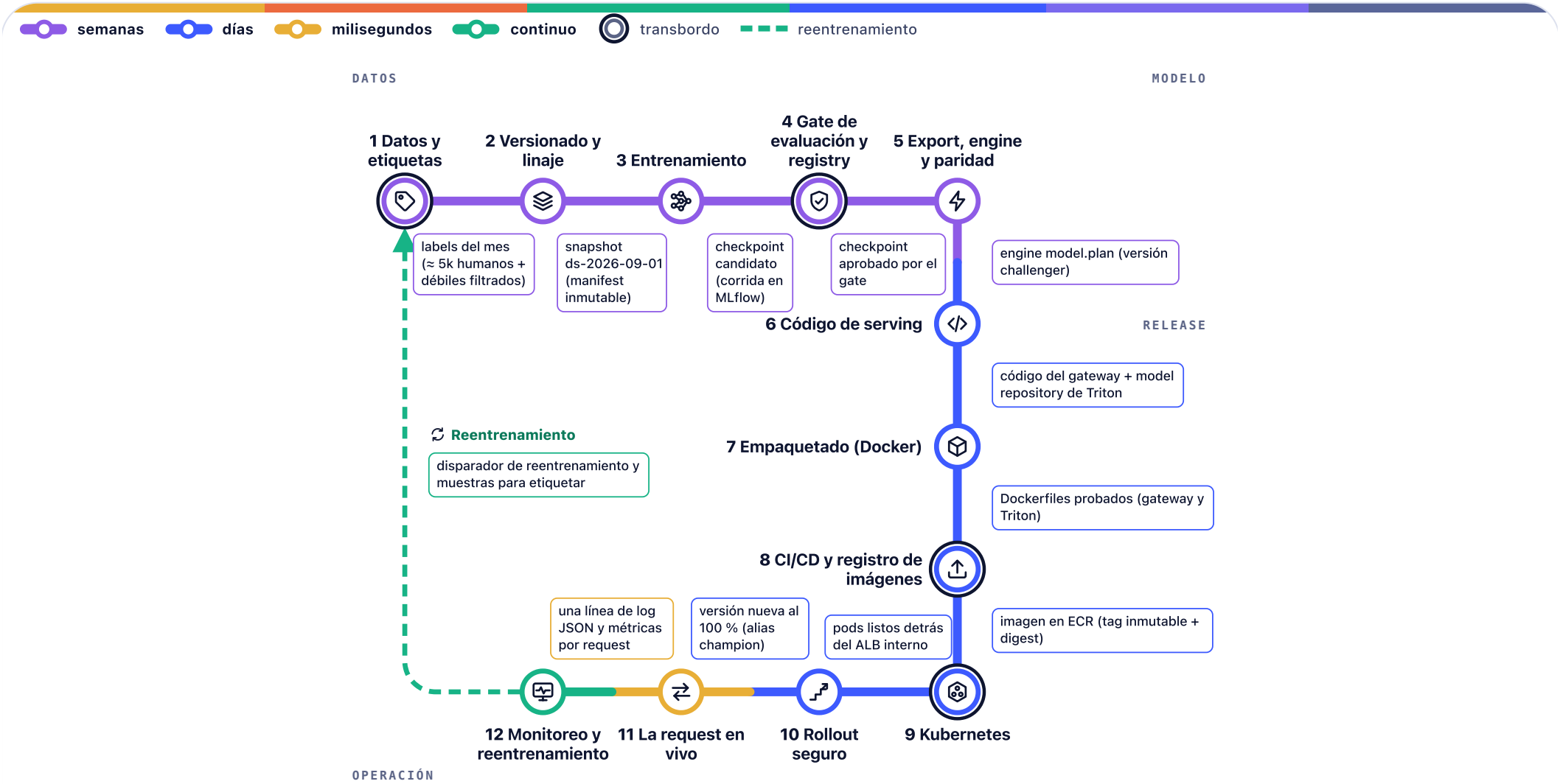
✓ ARCHIVO REAL · DESCRITO EN EL CURSO · *variable* la cambias tú

EL RECORRIDO EN UNA TABLA

#	ESTACIÓN	TRAMO	LÍNEA	LO QUE SALE Y A DÓNDE VIAJA
1	**Datos y etiquetas**	Datos	semanas	labels del mes (≈ 5k humanos + débiles filtrados) → 2 · Versionado
2	**Versionado y linaje**	Datos	semanas	snapshot ds-2026-09-01 (manifest inmutable) → 3 · Entrenamiento
3	**Entrenamiento**	Modelo	semanas	checkpoint candidato (corrida en MLflow) → 4 · Gate
4	**Gate de evaluación y registry**	Modelo	semanas	checkpoint aprobado por el gate → 5 · Export
5	**Export, engine y paridad**	Modelo	semanas	engine model.plan (versión challenger) → 6 · Serving
6	**Código de serving**	Release	días	código del gateway + model repository de Triton → 7 · Empaquetado
7	**Empaquetado (Docker)**	Release	días	Dockerfiles probados (gateway y Triton) → 8 · CI/CD
8	**CI/CD y registro de imágenes**	Release	días	imagen en ECR (tag inmutable + digest) → 9 · Kubernetes
9	**Kubernetes**	Operación	días	pods listos detrás del ALB interno → 10 · Rollout
10	**Rollout seguro**	Operación	días	versión nueva al 100 % (alias champion) → 11 · Request
11	**La request en vivo**	Operación	milisegundos	una línea de log JSON y métricas por request → 12 · Monitoreo
12	**Monitoreo y reentrenamiento**	Operación	continuo	disparador de reentrenamiento y muestras para etiquetar → 1 · Datos (reentrenamiento)

El mapa del C01

Doce estaciones en cuatro tramos, el artefacto que pasa de una a la siguiente y el reentrenamiento que cierra el ciclo.



Doce estaciones en cuatro tramos (Datos → Modelo → Release → Operación). Entre estación y estación viaja un artefacto concreto: si no puedes nombrarlo, esa flecha todavía es una caja negra.

CÓMO SE CIERRA EL CICLO

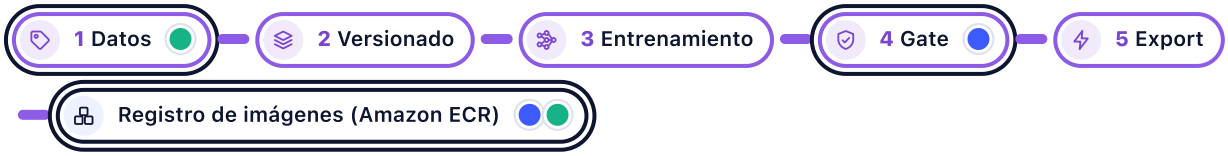
Cada request deja una línea JSON (request_id, model_version, top-5, latencias) que Fluent Bit lleva a S3; ahí se une con la categoría final de cada publicación y las correcciones de moderación. Sobre eso, el job diario calcula PSI y aceptación, y de esos logs salen las ≈ 5k imágenes al mes que se etiquetan. El calendario mensual, o un ticket por drift, arranca el siguiente reentrenamiento, que vuelve a pasar por gate, shadow y canary.

Las 4 velocidades

El mismo mapa partido en cuatro líneas que corren a ritmos distintos y solo se tocan en los transbordos.

El mapa junta cosas que corren a ritmos muy distintos: un modelo nuevo sale una vez al mes, una release tarda días y cada request se resuelve en milisegundos. Si lo separas en cuatro líneas, cada pregunta cae en la suya («¿por qué tarda el deploy?» es de la línea de días; «¿por qué subió el p99?», de la de milisegundos) y ves que solo se tocan en cuatro estaciones de transbordo. Es la distinción online/offline de M01 hecha mapa: los dos lados se conectan por logs → datos y por registry → deploy.

SEMANAS — Línea del modelo



RITMO

Mensual por calendario, y antes si un disparador lo adelanta. Cada ciclo junta ≈ 5k imágenes etiquetadas y el fine-tune dura ≈ 3–4 h en una GPU.

QUIÉN LA MUEVE

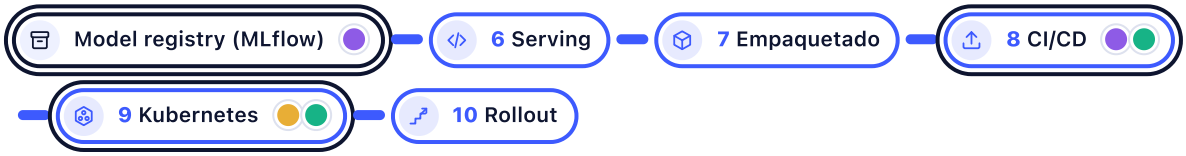
El orquestador corre solo el DAG (snapshot → validación → fine-tune → evaluación → gate → engine e imagen de Triton → paridad y latencia → registro); las personas son el equipo de etiquetado y el equipo de ML, que revisa el reporte del gate.

QUÉ LA DISPARA

El calendario mensual, o un ticket que lo adelanta: PSI > 0.2 sostenido 3 días, aceptación −3 pp semana contra semana o una categoría nueva en el árbol.

Termina en el registry con alias challenger: esta línea **nunca despliega**. El engine y la imagen de Triton que lo hornea se construyen en una T4 como pasos de este pipeline (después del gate), no en el release de código.

DÍAS — Línea del release



RITMO

Un commit pasa CI en minutos y el gateway sale con un rolling update; un modelo nuevo tarda días: shadow (1–2 días), canary 5 → 25 → 100 % con horas por etapa y cubriendo un pico diurno, y la release anterior viva 24–48 h para volver. Un A/B de 1–2 semanas solo si el cambio es grande.

QUIÉN LA MUEVE

CI/CD de forma automática (el curso propone GitHub Actions o GitLab CI); el análisis del canary revierte solo y una persona aprueba el paso final a 100 %.

QUÉ LA DISPARA

Un merge a main del gateway, o una versión challenger nueva en el registry con su imagen de Triton ya construida y probada en la T4: el release cambia los tags de imagen en los manifests.

Una release es el par **gateway vN + Triton vN** con su propio Service: nunca dos modelos detrás del mismo nombre. Al terminar el canary, el alias champion pasa a la versión nueva.

MILISEGUNDOS — Línea del request



RITMO

Siempre encendida: ≈ 116 req/s en promedio y ≈ 350 en el pico (500 de diseño). Cada request tarda ≈ 20–45 ms de mediana dentro del servidor; el SLO es p99 ≤ 200 ms.

QUIÉN LA MUEVE

Nadie en vivo: ALB, gateway y Triton responden solos. Las personas fijaron antes los timeouts (150 ms hacia Triton), las cuotas por caller y las réplicas mínimas.

QUÉ LA DISPARA

Cada foto que se sube al publicar: el backend de publicación llama POST /v1/classify server-to-server, con timeout de 300 ms.

Si algo falla, degrada en vez de caerse: `503` o `504` rápidos y el formulario muestra el selector manual de categorías.

CONTINUO — Línea de operación



RITMO

Nunca para: Prometheus scrapea gateway y Triton cada 15 s, las alertas miran ventanas de 5–30 min, el job de drift corre una vez al día y los labels llegan tarde (aceptación en minutos a horas, moderación en días).

QUIÉN LA MUEVE

Prometheus, Alertmanager y el job diario, solos; el on-call atiende los pages y el equipo de ML (con producto) los tickets en horario laboral.

QUÉ LA DISPARA

Umbrales escritos de antemano: p99 > 200 ms por 10 min o 5xx > 1 % por 5 min (page); cola de Triton > 20 ms (warning); PSI > 0.2 sostenido 3 días o aceptación −3 pp (ticket).

Cierra el ciclo: un ticket de drift prioriza qué etiquetar y adelanta la línea del modelo, que vuelve a empezar en **Datos**.

LOS 4 TRANSBORDOS: DONDE SE TOCAN LAS LÍNEAS

Model registry (MLflow)

UNE ● semanas y ● días

QUÉ CRUZA Una versión challenger que ya pasó gate, paridad y latencia: ONNX y engine, labels.json con su hash, versión del preprocesamiento, métricas y punteros a snapshot, commit y digest; su imagen de Triton ya está en ECR con la versión como tag. El release solo cambia tags en los manifests; al terminar el canary, champion apunta a ella.

DÓNDE VIVE MLflow Model Registry, con aliases champion y challenger.

En el mapa: ● estación 4 · Gate



Registro de imágenes (Amazon ECR)

UNE ● semanas, ● días y ● continuo

QUÉ CRUZA Imágenes inmutables: classifier-gateway con el tag del commit (la sube CI) y classifier-triton con la versión del modelo y el engine horneado (la sube el pipeline del modelo, ya probada en la T4). El release las despliega cambiando el tag en los manifests, y el cluster las vuelve a bajar en cada scale-up y en cada rollback.

DÓNDE VIVE Amazon ECR, con tags inmutables.

En el mapa: ● estación 8 · CI/CD



El cluster (Amazon EKS)

UNE ● días, ● milisegundos y ● continuo

QUÉ CRUZA Entran los manifests del release (Deployments, Services, HPA, ScaledObject, Ingress); adentro corren los pods que atienden cada request; salen las métricas (/metrics del gateway, :8002 de Triton) y los logs que lee la operación.

DÓNDE VIVE Amazon EKS en dos zonas, namespace classifier: node group de g4dn.xlarge (T4) para Triton y nodos c6i para el gateway.

En el mapa: ● estación 9 · Kubernetes



Datos: de logs a etiquetas

UNE ● continuo y ● semanas

QUÉ CRUZA Los logs de predicción unidos con la categoría final de cada publicación y las correcciones de moderación, las ≈ 5k imágenes al mes elegidas para etiquetar (baja confianza, desacuerdos y una muestra aleatoria) y los tickets de drift que adelantan el reentrenamiento.

DÓNDE VIVE S3 (data lake en Parquet, alimentado por Fluent Bit), la cola de etiquetado (Label Studio o un proveedor) y el ticket del job diario.

En el mapa: ● estación 1 · Datos

Estación 1 de 12 · tramo Datos · línea de semanas

Datos y etiquetas

¿De dónde salen los labels con los que el modelo aprende y se evalúa, y cuánto vale cada uno?

Transbordo · Datos: de logs a etiquetas. Aquí se tocan las líneas continuo y semanas. Qué cruza

En C01 el modelo ya existe, pero el catálogo cambia y un modelo congelado se degrada: esta estación convierte lo que pasa en producción en labels nuevos cada mes. Hay cuatro fuentes de calidad muy distinta: la categoría elegida al publicar (≈ 2.5 M/día, pero contaminada por tu propia sugerencia), las correcciones de moderación, una cola humana de **≈ 5k imágenes/mes elegidas a propósito** y un **golden set** de ≈ 50k, congelado, que solo sirve para evaluar.

ENTRA

- logs de predicción en S3 (Parquet, una fila por request)
- categoría final de cada publicación y correcciones de moderación
- señales de monitoreo: baja confianza, drift, categorías nuevas

SALE

- labels del mes (≈ 5k humanos + débiles filtrados) **VIAJA A 2 · VERSIONADO**
- fotos etiquetadas, sin EXIF, en un bucket restringido
- golden set congelado (≈ 50k, solo evaluación)

EL PROCESO, PASO A PASO

PASO 1
Juntar señales

Une los logs de predicción con la categoría final de cada publicación y las correcciones de moderación, por request_id.

- data engineer
- job de datos sobre los logs en S3 (Parquet)

PASO 2
Filtrar el label débil

La categoría elegida al publicar entra filtrada por señales de confianza: moderación no la corrigió en 30 días, o se buscó en vez de aceptar la sugerencia.

- MLE
- mismo job de datos

PASO 3
Elegir qué etiquetar

De la muestra del 1 %: baja confianza (top-1 < 0.3 o margen chico), desacuerdos y 20–30 % al azar, más categorías nuevas; ≈ 5k/mes.

- MLE
- consulta sobre los logs en S3

PASO 4
Copiar solo esas fotos

Del storage del producto a un bucket cifrado y restringido, sin EXIF; las copias de la cola vencen a los 90 días.

- data engineer
- Amazon S3 (bucket restringido)

PASO 5
Etiquetar

Con una guía de ejemplos límite; el 10 % lo etiquetan dos personas para medir el acuerdo.

- etiquetadores
- Label Studio o un proveedor gestionado

PASO 6
Medir acuerdo y adjudicar

Kappa por clase sobre la doble anotación; los desacuerdos van a una persona experta y a la guía.

- MLE
- export de anotaciones + scikit-learn

ARCHIVOS DEL C01

ARCHIVO REAL **reference/c01/gateway/app/main.py**

La materia prima: el gateway escribe una línea JSON por request con request_id, model_version, el top-k con sus scores y un prefijo del sha256 de la imagen. De ahí salen la muestra por baja confianza y el join con la categoría final.

PYTHON extracto

```python
    probs = _softmax(logits)
    top = np.argsort(probs)[::-1][:top_k]  # 500 clases: ordenar todo cuesta microsegundos
    labels = request.app.state.labels
    TOP1.labels(model_version=MODEL_VERSION).observe(float(probs[top[0]]))  # → alerta ClassifierLowConfidenceSpike
    record["top_k"] = [[int(i), round(float(probs[i]), 4)] for i in top]  # ids + scores para monitoreo/labeling
```

DESCRITO EN EL CURSO **guía de etiquetado (versionada)**

Definición de cada categoría con los casos límite decididos por escrito (¿un celular con funda va en celulares o en fundas?); cada label guarda la versión de la guía con que se hizo. El curso la describe; no está en reference/c01.

DESCRITO EN EL CURSO **golden set (≈ 50k imágenes)**

Etiquetado por humanos y revisado, estratificado por categoría (piso de ≈ 50 por categoría) y por origen (móvil o web), y congelado. Solo evaluación, jamás entrenamiento; si se refresca, es una versión nueva.

DESCRITO EN EL CURSO **holdout sin sugerencias (≈ 1 % de quienes venden)**

Una porción de cuentas que solo ve el selector manual: da labels que tu modelo no influyó y mide el valor incremental del sistema.

COMANDOS CLAVE

```
pip install label-studio && label-studio start
```

Label Studio Community en tu máquina: la UI queda en `localhost:8080`. C01 acepta también un proveedor gestionado.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
La métrica offline sube cada mes, pero la tasa de aceptación no se mueve.	Entrenas y evalúas con la categoría elegida al publicar, que casi siempre es tu primera sugerencia: el modelo aprende a imitarse.	Compara la métrica sobre esos labels contra el golden set y el holdout sin sugerencias: si solo sube la primera, es el feedback loop.
El acuerdo bruto entre anotadores se ve alto, pero en algunas categorías el modelo no aprende la frontera.	Guía ambigua en los casos límite; el porcentaje de acuerdo engaña cuando una clase domina.	Calcula Cohen's kappa por clase sobre el 10 % con doble anotación y lleva cada desacuerdo a la guía.
La accuracy medida en el set nuevo sale bastante más baja que la que ve el tráfico real.	Se recortó la muestra aleatoria: el set solo trae lo que eligió el active learning, que es más difícil a propósito.	Revisa la mezcla de la cola del mes: el 20–30 % tiene que venir de la muestra aleatoria.
Una categoría que negocio agregó al árbol nunca aparece en las sugerencias.	No hay ejemplos: el modelo no puede predecir una clase que no vio.	Cuenta los ejemplos etiquetados de esa categoría; siémbrala con búsquedas por texto del título hasta tener ≈ 200–500.

LISTA CUANDO…

- [x] La cola del mes tiene sus ≈ 5k imágenes con la mezcla acordada: incertidumbre, desacuerdos y 20–30 % al azar.
- [x] La doble anotación tiene su kappa por clase y los desacuerdos están adjudicados.
- [x] Solo se copiaron las fotos que se etiquetan, sin EXIF; el resto se referencia por request_id y sha256.
- [x] Los labels débiles entran filtrados y nunca como única fuente de evaluación.

CÓMO RECORDARLA

el label es el dato más caro. Etiquetar ≈ 5k imágenes cuesta ≈ US$250–500 al mes y entrenar ≈ US$10–50: el dinero (y la calidad) está en qué imágenes eliges, no en la GPU.

TRAMPA

tratar la categoría elegida al publicar como la verdad. Está sesgada por tu propia sugerencia: sirve para entrenar (filtrada) y como métrica online, nunca como única evaluación; para eso están el golden set y el holdout sin sugerencias.

En el curso: C01 · Pipeline de datos y entrenamiento · M9 · Etiquetado: guías, calidad, costo y active learning · M1 · El ciclo de vida de ML en producción

HERRAMIENTAS DE ESTA ESTACIÓN

Amazon S3

ALMACENAMIENTO DE OBJETOS

Un almacén de archivos en la nube, sin límite práctico de tamaño: guardas archivos («objetos») bajo un nombre con forma de ruta («key») dentro de un bucket, y pagas por lo que guardas y por cada lectura o escritura.

COMO *Como un Drive para máquinas: las «carpetas» son solo prefijos del nombre de cada archivo.*

EN EL C01 Guarda los logs de predicción (Parquet), las fotos etiquetadas sin EXIF en un bucket restringido, los manifests de cada snapshot (s3://…/datasets/ds-2026-09-01/), los checkpoints del fine-tune y los artefactos de MLflow.

¿CÓMO SE TRABAJA?

Administrado · se configura · se opera

No instalas nada: AWS lo opera. Tú **configuras** buckets y reglas (quién lee; una regla de lifecycle que borra las copias de la cola de etiquetado a los 90 días; una bucket policy que exige `If-None-Match: *` para no pisar keys) y tus jobs leen y escriben con el SDK o la CLI (`aws s3 ls`). S3 guarda los objetos de forma durable y cobra por GB y por request; no entiende de datasets: el manifest, la inmutabilidad y el sha256 de la tarjeta, que delata cambios, los pones tú.

EJEMPLO

TEXTO M9 · versionado: tarjeta de un snapshot de C01 (ejemplo del curso, no está en reference/c01)

```
{
  "snapshot_id": "ds-2026-09-01",
  "parent": "ds-2026-08-01",
  "created_at": "2026-09-01T03:12:44Z",
  "rows_path": "s3://ml-datos/datasets/ds-2026-09-01/rows.parquet",
  "rows_sha256": "9f2c…",
  "images": {"humano": "s3://ml-datos/labeled/ (restringido, sin EXIF)",
             "resto": "storage del producto (por key + sha256, no se copian)"},
```

DÓNDE CORRE

Servicio de AWS en la región del cluster; lo usan los jobs de datos, el entrenamiento, MLflow y Fluent Bit.

ALTERNATIVAS

Google Cloud Storage o Azure Blob Storage en otras nubes.

En el curso: M9 · Versionado de datos y snapshots inmutables · C01 · Pipeline de datos y entrenamiento · C01 · Estimaciones back-of-the-envelope

Label Studio

ETIQUETADO DE DATOS

Una aplicación web donde personas ven cada imagen y eligen su label; guarda las anotaciones y las exporta.

COMO *Un formulario de encuesta donde cada pregunta es una imagen y la respuesta, su categoría.*

EN EL C01 Es una de las dos opciones del curso para la cola mensual (≈ 5k imágenes, 10 % con doble anotación); la otra es un proveedor gestionado. En la edición Community el overlap, el ground truth y las métricas de acuerdo no vienen: exportas las anotaciones y calculas kappa tú.

¿CÓMO SE TRABAJA?

Se opera · se configura

Lo levantas (`label-studio start` o un contenedor) y **configuras** cada proyecto con una plantilla XML corta: qué se muestra (`Image`) y qué se responde (`Choices`). No copia las imágenes: las lee de S3 con URLs firmadas (links temporales) y escribe cada anotación como JSON en un bucket de destino. El día a día es **operar** la UI: el equipo de etiquetado anota, tú revisas y exportas.

EJEMPLO

TEXTO ilustrativo: el formato de plantilla de la documentación de Label Studio con dos categorías de gateway/app/labels.json

```
<View>
  <Image name="image" value="$image"/>
  <Choices name="category" toName="image">
    <Choice value="Celulares y telefonía > Smartphones"/>
    <Choice value="Celulares y telefonía > Fundas para celular"/>
  </Choices>
</View>
```

DÓNDE CORRE

Servidor web propio (pip o contenedor) al que se entra con el navegador; también hay ediciones de pago en la nube.

ALTERNATIVAS

Un proveedor gestionado (SageMaker Ground Truth en AWS), CVAT para visión o una herramienta interna mínima con la cola en una tabla.

En el curso: M9 · Etiquetado: guías, calidad, costo y active learning · C01 · Pipeline de datos y entrenamiento · Herramientas por categoría · Datos, streaming, features, versionado y etiquetado

Estación 2 de 12 · tramo Datos · línea de semanas

Versionado y linaje

¿Cómo sabes, meses después, con qué datos exactos se entrenó un modelo?

Cada dataset es un **snapshot inmutable** con id (`ds-2026-09-01`) y un manifest en Parquet: por imagen, hash, ruta, label, fuente del label y fecha. No copia fotos, las referencia, así que un snapshot nuevo pesa ≈ 0.1 % de las imágenes. Cada modelo registrado guarda tres punteros (snapshot, commit y digest del contenedor de entrenamiento): con eso respondes «¿con qué datos?», «¿lo puedo reproducir?» y «¿en qué datasets está esta foto que pidieron borrar?».

ENTRA

- labels del mes + referencias a imágenes (key + sha256)
- snapshot anterior (parent)

→

SALE

- snapshot ds-2026-09-01 (manifest inmutable) **VIAJA A 3 · ENTRENAMIENTO**
- reporte de validación del snapshot

EL PROCESO, PASO A PASO

PASO 1

Armar el manifest

Una fila por imagen: sha256, key, label, fuente del label, seller_id y fecha (≈ 400 MB sin comprimir para ≈ 2 M imágenes).

automático (pipeline)

paso «snapshot» del pipeline mensual (job de datos)

PASO 2

Deduplicar y separar

Dedupe por hash (la misma foto subida en 3 publicaciones no cae en train y en test) y split por seller_id.

automático (pipeline)

mismo paso, sobre el manifest

PASO 3

Publicar inmutable

Escribe en s3://.../datasets/ds-2026-09-01/ con una bucket policy que exige If-None-Match: * (escritura condicional de S3) para que nadie pise una key; el sha256 del manifest en la tarjeta (rows_sha256) delata si alguien lo reescribió.

automático (pipeline)

Amazon S3 (un prefijo por snapshot)

PASO 4

Validar antes de gastar GPU

Labels dentro de las 500 categorías, cero duplicados con el golden set y conteos por fuente en rango.

automático (pipeline)

paso «validar datos» del pipeline

PASO 5

Apuntar desde el modelo

La corrida guarda el id del snapshot, el commit y el digest de la imagen de entrenamiento; la versión registrada los hereda.

automático (pipeline)

MLflow (parámetros y tags de la corrida)

ARCHIVOS DEL C01

DESCRITO EN EL CURSO **s3://.../datasets/ds-2026-09-01/ (manifest en Parquet)**

Una fila por imagen con hash, ruta, label, fuente del label y fecha. Las fotos no se copian: se referencian por key + sha256. Lo describen C01 y M9; no está en reference/c01.

DESCRITO EN EL CURSO **tarjeta del snapshot (JSON junto al manifest)**

Snapshot padre, conteos por split, mezcla de fuentes de labels, regla del split, commit del código y resultado de la validación. M9 muestra un ejemplo.

DESCRITO EN EL CURSO **punteros de la corrida en MLflow**

dataset = id del snapshot, commit de git y digest de la imagen de entrenamiento: el linaje versión → corrida → snapshot → hash de cada imagen.

COMANDOS CLAVE

```
aws s3 ls s3://bucket/datasets/snapshot_id/
```

Mira qué hay en el prefijo de un snapshot (solo lectura).

```
dvc add carpeta_datos && dvc push
```

Si el equipo ya usa DVC: hashea la carpeta, deja `carpeta_datos.dvc` para git y sube los datos al remoto en S3.

```
lakectl branch create lakefs://repo/rama -s lakefs://repo/main
```

Si usan lakeFS: una rama aislada sobre el bucket, sin copiar datos; después `lakectl commit` y `lakectl merge`.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
Nadie puede decir con qué datos se entrenó el modelo que está en producción.	El entrenamiento leyó una carpeta «latest» o un prefijo que alguien sobrescribió.	En MLflow, el parámetro dataset de la corrida tiene que ser un id de snapshot (ds-…), no una ruta.
La corrida de este mes dio exactamente las mismas métricas que la del mes pasado.	El paso de snapshot no recibió el id como parámetro y la caché del orquestador (activa por defecto en Kubeflow) reutilizó el snapshot anterior.	Revisa los parámetros de la corrida en el orquestador: dataset_id tiene que cambiar cada mes.
La accuracy offline sale sospechosamente alta.	La misma foto quedó en train y en el golden set (subida en varias publicaciones), o el split no fue por seller_id.	Cruza los sha256 del manifest de train con los del golden set: tiene que dar cero.
Alguien pidió borrar sus fotos y no sabes en qué datasets ni modelos están.	Falta el manifest con hash por imagen o el puntero de cada modelo a su snapshot.	Busca el sha256 en los manifests y sigue versión → corrida → snapshot en MLflow.

✓ LISTA CUANDO…

- [x] El snapshot tiene id, manifest y tarjeta, y su prefijo no se puede sobrescribir.
- [x] La validación pasó: 500 categorías, cero duplicados con el golden set, conteos por fuente en rango.
- [x] El pipeline recibe el id del snapshot como parámetro, nunca una carpeta «latest».
- [x] Desde un sha256 llegas a los snapshots y a los modelos que lo usan.

CÓMO RECORDARLA

snapshot = manifest, no copia. Un id inmutable más tres punteros desde el modelo (datos, commit, digest) responden «¿con qué se entrenó?» en un minuto.

TRAMPA

inmutable «por convención». Sin una política que prohíba sobrescribir, alguien reescribe el prefijo y el modelo de marzo deja de ser reproducible. Y object lock protege versiones, no keys: un PUT con la misma key crea otra versión; por eso el manifest guarda el sha256.

En el curso: C01 · Pipeline de datos y entrenamiento | M9 · Versionado de datos y snapshots inmutables | M10 · Reproducibilidad: los cinco ingredientes

HERRAMIENTAS DE ESTA ESTACIÓN

DVC / lakeFS

VERSIONADO DE DATOS

Herramientas que le dan a tus datos lo que git le da a tu código: versiones con id, historial y volver atrás, sin meter los archivos en git.

COMO *DVC: git guarda la etiqueta del equipaje (el hash) y el equipaje viaja en S3. lakeFS: ramas de git sobre el bucket entero.*

EN EL C01 El curso los usa solo si ya existen en la empresa. Si no, C01 versiona con prefijos inmutables en S3 y un manifest en Parquet: lo importante es la inmutabilidad y el puntero desde el modelo, no la marca.

¿CÓMO SE TRABAJA?

Se opera · se configura

Con DVC **operas** comandos junto a git: `dvc add` calcula el hash y deja un archivo `.dvc` chico que sí va a git, y `dvc push` sube los datos al remote, el almacenamiento de DVC (aquí, un bucket de S3). lakeFS es un servidor delante del bucket: **configuras** el repositorio y creas ramas y commits con `lakectl`, su UI o su API compatible con S3; una rama nueva no copia datos.

EJEMPLO

BASH comandos de la documentación de DVC y lakeFS (C01 no los usa por defecto)

```bash
dvc add {carpeta_datos}            # crea {carpeta_datos}.dvc con el hash y agrega la carpeta a .gitignore
git add {carpeta_datos}.dvc .gitignore && git commit -m "{mensaje}"
dvc remote add -d {remote} s3://{bucket}/{prefijo}
dvc push                           # sube los datos al remote en S3
lakectl branch create lakefs://{repo}/{rama} -s lakefs://{repo}/main
lakectl commit lakefs://{repo}/{rama} -m "{mensaje}"
lakectl merge lakefs://{repo}/{rama} lakefs://{repo}/main
```

DÓNDE CORRE

DVC: una CLI en tu máquina o en el job, con los datos en S3. lakeFS: un servidor delante del bucket, propio o administrado (licencia BSL 1.1 desde la v1.87.0: el uso interno sin modificar está permitido).

ALTERNATIVAS

Prefijos inmutables en S3 + manifest en Parquet (la elección por defecto de C01); Delta Lake o Iceberg con time travel para tablas, cuidando su retención.

En el curso: M9 · Versionado de datos y snapshots inmutables · Herramientas por categoría · Datos, streaming, features, versionado y etiquetado

Amazon S3

almacenamiento de objetos · ficha completa en la estación 1 · Datos →

administrado · se configura · se opera

Estación 3 de 12 · tramo Modelo · ● línea de semanas

3 Entrenamiento

¿Cómo se vuelve tu notebook de fine-tuning un job que corre solo cada mes y se puede reproducir?

Un pipeline orquestado lanza el fine-tune de ResNet-50 en **una GPU on-demand por unas horas**: warm start desde el champion, AMP y el mismo `preprocess.py` que usa el gateway. Corre cada mes, o antes si un disparador de monitoreo lo adelanta. Todo queda en MLflow: parámetros, métricas por época, artefactos, id del snapshot, commit y digest de la imagen.

ENTRA

- id del snapshot (como parámetro)
- checkpoint del champion (warm start)
- imagen de entrenamiento por digest

SALE

- checkpoint candidato (corrida en MLflow) — VIAJA A 4 · GATE
- curva de val_top1 y métricas finales
- checkpoints intermedios en S3

EL PROCESO, PASO A PASO

PASO 1
Probar el código en CI

Cada commit al código de training sobreajusta un batch de 32 imágenes: la loss tiene que caer casi a cero.

automático (CI)

job de CI (CPU o una GPU chica)

PASO 2
Disparar el pipeline

Calendario mensual, o un disparador de monitoreo que abre un ticket y adelanta la corrida; @champion se resuelve una vez.

automático (pipeline)

orquestador (Argo Workflows, Airflow o Kubeflow)

PASO 3
Pedir una GPU

Un pod de Kubernetes con toleration de GPU o un training job gestionado; nadie deja GPUs de entrenamiento prendidas.

automático (pipeline)

nodo GPU on-demand (EKS) o servicio gestionado

PASO 4
Fine-tune

Warm start desde el champion, AMP, seeds fijadas y el mismo preprocess.py del gateway; ≈ 3–4 h.

automático (pipeline)

contenedor de entrenamiento (por digest), 1 GPU

PASO 5
Checkpoint cada 15 min

Pesos + optimizador (≈ 300 MB con Adam) a S3; si se pierde el nodo, el orquestador reintenta el paso y reanuda.

automático (pipeline)

Amazon S3 (prefijo de checkpoints)

PASO 6
Registrar la corrida

Parámetros, val_top1 por época, artefactos, id del snapshot, commit y digest de la imagen.

automático (pipeline)

MLflow (tracking)

ARCHIVOS DEL C01

ARCHIVO REAL **`reference/c01/gateway/app/preprocess.py`**

El preprocesamiento único: el DataLoader de entrenamiento y evaluación lo importa igual que el gateway (EXIF → RGB → resize 256 → crop 224 → normalización de ImageNet). Así no hay training-serving skew de lógica por construcción.

PYTHON extracto

```python
def preprocess(data: bytes, max_pixels: int | None = None) -> np.ndarray:
    """Punto de entrada único: bytes → float32 [3,224,224]. Lo usan gateway, eval y training."""
    return to_chw_normalized(resize_center_crop(decode(data, max_pixels=max_pixels)))
```

DESCRITO EN EL CURSO **código de entrenamiento (script versionado)**

Dataset sobre el manifest, loop con AMP, seeds, checkpoints y logging a MLflow. El curso lo describe y M10 muestra fragmentos; no está en reference/c01.

DESCRITO EN EL CURSO **definición del pipeline (DAG)**

Snapshot → validar → fine-tune → evaluar → gate → engine e imagen de Triton → paridad y latencia → registrar, cada paso en un contenedor con su imagen por digest. M10 la muestra en Kubeflow Pipelines.

DESCRITO EN EL CURSO **imagen de entrenamiento (por digest)**

Python, torch, CUDA, cuDNN y la misma versión de Pillow que el gateway, referenciada por @sha256 y no por tag.

COMANDOS CLAVE

```
mlflow server --backend-store-uri postgresql://usuario:clave@host_db/db --artifacts-destination s3://bucket
--host 0.0.0.0
```

Lo levanta plataforma una vez, no en cada corrida: UI y API en el puerto 5000, metadatos en Postgres y artefactos en S3. Sin `--host 0.0.0.0` solo responde en esa máquina; si los jobs lo llaman por nombre, suma `--allowed-hosts` *nombre* (MLflow ≥ 3.5).

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
La accuracy online es varios puntos menor que la offline y no hay ningún error en los logs.	Training-serving skew: el entrenamiento preprocesa distinto que el gateway (PIL vs OpenCV, bilinear vs bicubic, draft mode u otra versión de Pillow).	Verifica que training importe gateway/app/preprocess.py con el mismo Pillow y corre la paridad end-to-end por el gateway real.
Dos corridas con la misma config dan 85.3 % y 85.1 % de top-1.	Ruido de seed y no determinismo de cuDNN: no es una regresión de la receta.	Compárala con la dispersión medida entre seeds (≈ 0.1 pp en el ejemplo del curso) antes de concluir.
El mejor modelo salió de un notebook y nadie lo puede reproducir.	Corrió fuera del pipeline: celdas fuera de orden, datos locales o cambios sin commit (el tag de commit de MLflow no avisa de cambios sin commit).	Los candidatos salen solo del pipeline, desde un commit limpio; revisa en MLflow de dónde vino la corrida.
El fine-tune murió a la hora 2 y alguien relanzó todo a la mañana siguiente.	Se perdió el nodo GPU (spot o falla) y no había checkpoints ni reintento por paso.	Checkpoint cada 15 min en S3 y reintentos en el paso de fine-tune del orquestador.
La loss no baja ni siquiera con unas pocas imágenes.	Bug en el código de entrenamiento (labels desalineados, learning rate roto).	Corre el smoke test: sobreajustar un batch de 32 imágenes tiene que llevar la loss casi a cero.

✓ LISTA CUANDO…

- [x] La corrida salió del pipeline desde un commit limpio, no de una laptop.
- [x] MLflow tiene parámetros, curva de val_top1, artefactos, id del snapshot, commit y digest.
- [x] El entrenamiento usó el mismo preprocess.py y la misma versión de Pillow que el gateway.
- [x] La GPU se apagó al terminar y los checkpoints quedaron en S3.

CÓMO RECORDARLA

el notebook explora, el pipeline produce. Lo que va a producción sale de un script versionado que corre el orquestador en una GPU efímera, con todo anotado en MLflow.

TRAMPA

«fijé la seed, así que es reproducible». cuDNN y las sumas atómicas en GPU cambian los resultados; en C01 reproducible significa «mismo resultado dentro del ruido», con los cinco ingredientes fijados: código, datos, config, entorno y seeds.

En el curso: C01 · Pipeline de datos y entrenamiento · M10 · Pipelines de entrenamiento orquestados · M10 · Reproducibilidad: los cinco ingredientes · M9 · Training-serving skew: causas, detección y prevención

HERRAMIENTAS DE ESTA ESTACIÓN

Orquestador de pipelines (Argo Workflows / Airflow / Kubeflow Pipelines)

ORQUESTACIÓN DE PIPELINES (DAGS)

Un sistema que corre una receta de pasos en orden (un DAG), cada uno en su contenedor, y se encarga de reintentos, dependencias, historial y disparos por calendario o evento.

COMO *Un supervisor de línea de producción: si una estación falla, repite solo esa y lo anota todo en la bitácora.*

EN EL C01 Corre el pipeline mensual: snapshot → validar → fine-tune (GPU) → evaluar → gate → engine e imagen de Triton en una T4 → paridad y latencia → registrar. C01 no fija marca: nombra Argo Workflows, Airflow o Kubeflow Pipelines, y como alternativa más simple un CronJob de Kubernetes con un script por etapas.

¿CÓMO SE TRABAJA?

{ } Se programa · se configura · se opera

Tú **escribes** la definición del DAG: en Python con Airflow o con el SDK de Kubeflow (que compila a YAML), o directo en YAML con Argo Workflows. El orquestador la ejecuta solo: agenda, lanza cada paso en su contenedor, reintenta y guarda logs y artefactos. Lo **operas** desde su UI o su CLI: ver corridas, relanzar un paso, entender por qué falló.

EJEMPLO

PYTHON M10 · pipelines de entrenamiento: el pipeline de C01 en Kubeflow Pipelines (ejemplo del curso)

```python
@dsl.pipeline(name="c01-monthly")
def c01_monthly(dataset_id: str, champion_version: str,   # el scheduler resuelve @champion una vez
                seed: int = 17, golden_set: str = "golden-v3"):
    snap = snapshot(dataset_id=dataset_id)   # el id es una entrada: la caché no puede confundir meses
    ft = finetune(manifest=snap.outputs["manifest"], seed=seed,
                  ckpt_uri=f"s3://ml-ckpt/c01/{dataset_id}/")
    ft.set_accelerator_type("nvidia.com/gpu").set_accelerator_limit(1)
    ft.set_retry(num_retries=3, backoff_duration="120s")   # un nodo perdido no tira la corrida
```

DÓNDE CORRE

Argo Workflows y Kubeflow, dentro del cluster de Kubernetes; Airflow, donde lo despliegues o administrado (MWAA).

ALTERNATIVAS

Un CronJob de Kubernetes con un script por etapas (lo más simple para un modelo al mes), SageMaker Pipelines o Dagster.

En el curso: M10 · Pipelines de entrenamiento orquestados

Herramientas por categoría · Orquestación: contenedores, escalado, workflows y despliegue · C01 · Pipeline de datos y entrenamiento

PyTorch

FRAMEWORK DE DEEP LEARNING

La librería de Python con la que defines, entrenas y corres redes neuronales: tu terreno conocido.

EN EL C01 Fine-tunea cada mes el ResNet-50 de 500 clases (warm start, AMP) y queda como referencia FP32: export_onnx.py y parity_check.py cargan el checkpoint con PyTorch para comparar todo lo demás contra él. En producción no corre: se sirve el engine de TensorRT.

¿CÓMO SE TRABAJA?

{ } Se programa

Lo **programas**: el script de entrenamiento (dataset sobre el manifest, loop con AMP, seeds, checkpoints, logging a MLflow) es código tuyo y versionado. Lo que cambia en producción es quién lo corre: el orquestador lo lanza en un contenedor fijado por digest (el hash de la imagen) sobre una GPU efímera, nunca desde un notebook.

EJEMPLO

PYTHON reference/c01/model/export_onnx.py (extracto de build_model)

```python
    # weights=None: nada de descargar pesos de ImageNet; num_classes=500 crea la fc correcta
    model = torchvision.models.resnet50(weights=None, num_classes=NUM_CLASSES)
    if checkpoint:
        # weights_only=True: carga solo tensores, no ejecuta pickle arbitrario (un .pt es código)
        state = torch.load(checkpoint, map_location="cpu", weights_only=True)
        # strict (por defecto): si una llave no coincide, falla AQUÍ y no en producción con pesos al azar
        model.load_state_dict(state)
```

DÓNDE CORRE

En el contenedor de entrenamiento, sobre un nodo GPU on-demand; en CI (CPU), como referencia del export y de la paridad.

En el curso: M10 · Reproducibilidad: los cinco ingredientes · C01 · Pipeline de datos y entrenamiento

M9 · Training-serving skew: causas, detección y prevención

MLflow (tracking + model registry)

TRACKING DE EXPERIMENTOS Y REGISTRY DE MODELOS

Un servidor con UI donde cada entrenamiento deja sus parámetros, métricas y archivos, más un catálogo de versiones de modelo con nombres que se mueven (aliases).

COMO *Un cuaderno de laboratorio compartido más un estante con etiquetas móviles: «champion» y «challenger».*

EN EL C01 Registra cada corrida (snapshot, commit, digest, val_top1 por época) y guarda el modelo classifier por versiones con los aliases champion (lo que sirve producción) y challenger (el candidato). Cada versión incluye ONNX, engine, labels.json con su hash y las métricas del gate.

¿CÓMO SE TRABAJA?

{ } Se programa | se opera

Lo **programas** desde tu código: `mlflow.log_params` y `mlflow.log_metric` en el entrenamiento, y `MlflowClient` para crear versiones y mover aliases. El servidor guarda metadatos en Postgres y archivos en S3, y sirve una UI para comparar corridas; lo **operas** como cualquier servicio. Promover es mover el alias (los stages Staging/Production de muchos tutoriales están deprecados desde MLflow 2.9), y lo hace el pipeline de CD, no una persona desde la consola.

EJEMPLO

PYTHON M10 · model registry: al registrar, la imagen de Triton ya existe (se construyó y pasó la paridad en la T4)

```python
# Último paso del pipeline del modelo: engine e imagen de Triton ya pasaron paridad y latencia
mv = client.create_model_version("classifier", source=f"{run.info.artifact_uri}/model",
                                 run_id=run.info.run_id)     # la versión apunta a su corrida: linaje
client.set_model_version_tag("classifier", mv.version, "gate", "passed")
client.set_registered_model_alias("classifier", "challenger", mv.version)

# Al terminar el canary (lo hace CD, no una persona con la consola):
client.set_registered_model_alias("classifier", "champion", mv.version)
```

DÓNDE CORRE

Un servidor (UI + API REST) con Postgres para metadatos y un bucket S3 para artefactos; lo opera el equipo.

ALTERNATIVAS

MLflow administrado (SageMaker AI, Databricks, Azure ML), Weights & Biases, o una carpeta por versión en S3 + una tabla.

En el curso: M10 · Model registry: versiones, aliases y linaje | M10 · Experiment tracking: qué registrar y con qué | Herramientas por categoría · Experimentos y registro de modelos

Estación 4 de 12 · tramo Modelo · ● línea de semanas

Gate de evaluación y registry

¿Quién decide, y con qué números, que el modelo nuevo puede reemplazar al de producción?

Transbordo · Model registry (MLflow). Aquí se tocan las líneas ● semanas y ● días. Qué cruza

Un gate **automático y escrito antes de entrenar** compara challenger contra champion sobre el mismo golden set: no inferioridad pareada (≥ −0.3 pp en top-1 y top-5), recall de las 100 categorías con más tráfico, ECE ≤ 0.05 y slices. Si pasa, el checkpoint sigue a la estación 5 (engine e imagen de Triton en una T4) y **recién después de paridad y latencia** se registra en MLflow con alias `challenger`; `champion` se mueve cuando termina el canary.

ENTRA

- checkpoint candidato (corrida en MLflow)
- champion actual (versión fija, como parámetro)
- golden set congelado (≈ 50k)

SALE

- checkpoint aprobado por el gate — VIAJA A 5 · EXPORT
- reporte del gate (veredicto, slices, recall por categoría)
- versión challenger en MLflow (después de la estación 5)

EL PROCESO, PASO A PASO

PASO 1 — Evaluar a los dos

Challenger y champion sobre las mismas ≈ 50k imágenes; se guardan las predicciones por imagen para el test pareado.

automático (pipeline) · paso «evaluar» del pipeline, en GPU

PASO 2 — Aplicar los umbrales

Diferencia pareada ≥ −0.3 pp (también contra el champion de hace 3 meses), recall top-100 con McNemar + Holm, ECE ≤ 0.05 y slices.

automático (pipeline) · paso «gate» del pipeline

PASO 3 — Si falla, frena

La corrida termina en rojo con el motivo en el tracking y avisa; un override pide dos personas y deja un tag.

automático (pipeline) · orquestador + MLflow

PASO 4 — Pasar a export

El checkpoint aprobado se exporta, se compila y se hornea en la imagen de Triton; contra ese Triton corren la paridad y la latencia (estación 5).

automático (CI) · runner de CI con T4

PASO 5 — Registrar la versión

Solo con paridad y latencia en verde: ONNX, engine, labels.json con su hash, versión del preprocess, métricas y punteros; tag gate=passed y alias challenger.

automático (pipeline) · MLflow Model Registry

PASO 6 — Promover al final

Después del shadow y del canary, CD mueve champion a esta versión; la anterior queda lista para rollback.

automático (CD) · pipeline de CD (estación 10)

ARCHIVOS DEL C01

ARCHIVO REAL **`reference/c01/gateway/app/labels.json`**

Los 500 labels en el orden de las clases del entrenamiento. El orden es parte del modelo: se registra con la versión y con su hash. En la referencia viaja en la imagen del gateway (lo más simple); endurecido, se publica junto al engine.

TEXTO extracto

```
[
  "Celulares y telefonía > Smartphones",
  "Celulares y telefonía > Teléfonos básicos",
  "Celulares y telefonía > Fundas para celular",
```

DESCRITO EN EL CURSO **umbrales del gate (archivo versionado)**

Los criterios escritos antes de la corrida: margen de −0.3 pp, recall por categoría con test, ECE ≤ 0.05, slices, paridad ≥ 99.5 % y p99 ≤ champion + 10 %. Cambian solo en un PR propio.

DESCRITO EN EL CURSO **reporte del gate**

Predicciones por imagen de los dos modelos, métricas por slice y por categoría, y el veredicto con su motivo; queda en la corrida y, si pasa, como tags de la versión.

DESCRITO EN EL CURSO **versión del registry (classifier vN)**

Alias challenger y, tras el canary, champion. Incluye ONNX, engine (con la GPU y la release de NGC como tags), labels.json + hash, versión del preprocess, métricas del gate y punteros a snapshot, commit y digest.

COMANDOS CLAVE

```
python -c "from mlflow import MlflowClient; print(MlflowClient().get_model_version_by_alias('classifier', 'champion').version)"
```

¿Qué versión es `champion` ahora? Con `MLFLOW_TRACKING_URI` apuntando al servidor.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
El gate bloquea casi todos los reentrenamientos mensuales.	Exige «≥ champion» exacto (con ≈ 50k imágenes el ruido es ≈ ±0.3 pp: decide al azar) o una mejora significativa, que un refresco mensual casi nunca muestra.	Revisa el criterio: no inferioridad con margen (diferencia pareada ≥ −0.3 pp, con McNemar o bootstrap).
El top-1 global sube, pero una categoría grande se rompió y nadie lo notó.	El gate solo miraba el promedio.	Recall por categoría de las 100 con más tráfico (test pareado + Holm) y slices: móvil, web, poca luz, imágenes chicas.
Cada mes el gate pasa, pero el modelo de hoy es claramente peor que el de hace un año.	Erosión del margen: −0.2 pp por mes, doce meses seguidos.	Compara también contra el champion de hace 3 meses.
La corrida terminó en verde, pero no apareció ningún challenger.	El paso del gate devolvió false en vez de fallar: los pasos siguientes se saltaron y nadie se enteró.	Haz que el gate lance una excepción: corrida en rojo y alerta de pipeline fallido.
Todo responde 200 con scores normales, pero los nombres de categoría no corresponden; solo cae la aceptación.	labels.json desalineado con el modelo (otro orden de clases).	Compara el hash de labels.json registrado con la versión contra el que sirve el gateway; 500 labels = dimensión de logits.

✓ LISTA CUANDO…

- [x] El candidato es no inferior al champion (≥ −0.3 pp pareado) y al de hace 3 meses.
- [x] Ninguna de las 100 categorías grandes cae > 5 pp con caída significativa (McNemar + Holm), ningún slice cae más que su intervalo y ECE ≤ 0.05.
- [x] Paridad y latencia (estación 5) pasaron y la versión quedó como challenger con todos sus artefactos.
- [x] Sabes qué versión es champion y a cuál vuelves si algo falla.

CÓMO RECORDARLA

el gate decide si vale la pena probar; el canary, si se queda. Promover y volver atrás es mover el alias `champion`, no reentrenar.

TRAMPA

mover los umbrales después de ver el resultado («el slice cayó 2 pp, subo el límite a 2.5»). El gate deja de medir y empieza a justificar: los umbrales cambian en un PR propio, antes de la corrida.

En el curso: C01 · Pipeline de datos y entrenamiento · C01 · Evaluación offline y online · M10 · El validation gate: cuándo un modelo puede reemplazar al actual · M10 · Model registry: versiones, aliases y linaje

HERRAMIENTAS DE ESTA ESTACIÓN

MLflow (tracking + model registry)
tracking de experimentos y registry de modelos · ficha completa en la estación 3 · Entrenamiento →

{} se programa se opera

5

Estación 5 de 12 · tramo Modelo · ● línea de semanas

Export, engine y paridad

¿Cómo pasa el checkpoint de PyTorch a un engine que Triton sirve rápido, sin cambiar sus predicciones?

El checkpoint aprobado se exporta a **ONNX opset 17** (batch dinámico, nombres `input` / `logits`) y pasa una primera paridad en CPU con ONNX Runtime. Después, **en una T4 y con la misma release de NGC (24.08) que Triton**, porque un engine no es portable, `trtexec --fp16` compila el engine, `docker build triton/` lo hornea en la imagen de Triton y, con ese Triton arriba, corren los dos gates: paridad contra PyTorch FP32 (top-1 agreement ≥ 99.5 %) y latencia con perf_analyzer (p99 ≤ champion + 10 %). Todo es parte del pipeline del modelo, en un runner de CI con T4, y solo si pasa la versión se registra como challenger (estación 4).

ENTRA

- checkpoint aprobado por el gate (.pt)
- muestra de imágenes del set de evaluación

SALE

- engine model.plan (versión challenger) — **VIAJA A 6 · SERVING**
- imagen de Triton con el engine horneado (tag = versión del modelo)
- ONNX, reportes de paridad y latencia, y huellas (sha256, GPU, NGC) para el registry

EL PROCESO, PASO A PASO

PASO 1 → Exportar a ONNX

export_onnx.py: modelo en eval(), exportador TorchScript, opset 17 y batch dinámico; verifica el opset real y corre ORT con batch 1 y 32.

automático (CI) · job de CI (CPU)

PASO 2 → Paridad en CPU

parity_check.py con ONNX Runtime contra PyTorch FP32: valida el export sin tocar una GPU.

automático (CI) · job de CI (CPU)

PASO 3 → Compilar el engine

build_engine.sh: aborta si la GPU no es T4; trtexec --fp16 con perfil 1/16/32 dentro de tensorrt:24.08.

automático (CI) · runner de CI con T4 (g4dn.xlarge)

PASO 4 → Levantar Triton con el engine

docker build triton/ hornea el engine en la imagen (tag = versión del modelo) y ese Triton arranca en la misma T4: lo que se prueba es lo que se despliega.

automático (CI) · runner de CI con T4

PASO 5 → Paridad FP16

parity_check.py --backend triton: top-1 ≥ 99.5 %, top-5 ≥ 99 % y |Δlogit| ≤ 0.5; exit 1 frena todo.

automático (CI) · runner de CI con T4

PASO 6 → Gate de latencia

perf_analyzer a 250 y 500 req/s contra ese Triton: p99 ≤ champion + 10 %.

automático (CI) · runner de CI con T4

PASO 7 → Registrar huellas

sha256 del ONNX y del plan, GPU y NGC_RELEASE van a la versión challenger del registry.

automático (CI) · MLflow (estación 4)

ARCHIVOS DEL C01

✓ ARCHIVO REAL **reference/c01/model/export_onnx.py**

Checkpoint → ONNX (exportador TorchScript, opset 17, batch dinámico, nombres input/logits). Su verify() falla si el opset del archivo no es el pedido o si ONNX Runtime no coincide con PyTorch en batch 1 y 32; CI lee el exit code.

PYTHON extracto

```python
    proto = onnx.load(path)  # carga el grafo con sus pesos
    onnx.checker.check_model(proto, full_check=True)  # estructura válida + shape inference completa
    real_opset = next(o.version for o in proto.opset_import if o.domain in ("", "ai.onnx"))
    dims = [d.dim_param or d.dim_value for d in proto.graph.input[0].type.tensor_type.shape.dim]
    print(f"[export] opset={real_opset} input={proto.graph.input[0].name}{dims} output={proto.graph.output[0].name}")
    ok = real_opset == opset  # el opset pedido debe ser el real (ver la trampa del exportador dynamo)
```

✓ ARCHIVO REAL **reference/c01/model/parity_check.py**

Gate de paridad con métricas de producto: acuerdo top-1, solapamiento del top-5 y |Δlogit| máximo, contra ONNX Runtime (CPU) o contra el engine servido por Triton. Exit 1 = no se promueve.

PYTHON extracto

```python
    p.add_argument("--min-top1", type=float, default=0.995, help="umbral del gate: acuerdo top-1 ≥ 99.5 %%")
    p.add_argument("--min-top5", type=float, default=0.99, help="solapamiento medio del top-5")
    # FP32 vs FP32 da ~1e-4; TensorRT FP16 da ~1e-2: el umbral depende del backend (ajústalo con datos)
    p.add_argument("--max-abs-diff", type=float, default=0.5, help="tope de |Δlogit| (atrapa exports rotos)")
```

✓ ARCHIVO REAL **reference/c01/model/build_engine.sh**

Compila classifier.onnx a model.plan con trtexec --fp16 dentro de nvcr.io/nvidia/tensorrt:24.08-py3. Aborta si la GPU no es una T4 y al final imprime GPU, release y sha256 para el registry.

BASH extracto

```bash
# Guardia: compilar en la GPU equivocada produce un engine que falla (o rinde mal) al servir
gpu="$(nvidia-smi --query-gpu=name --format=csv,noheader | head -n1)"
if [[ "$gpu" != *"$EXPECTED_GPU"* ]]; then
  echo "GPU detectada: '$gpu'. Este engine debe construirse en una $EXPECTED_GPU (la de producción)." >&2
  exit 1
fi
```

✓ ARCHIVO REAL **reference/c01/model/requirements.txt**

Entorno del job de export y paridad, todo fijado: la versión de torch cambia el comportamiento del exportador y Pillow tiene que ser la misma que en el gateway.

TEXTO extracto

```
torch==2.14.0  # la versión FIJA importa: el exportador TorchScript está deprecado y cambia entre releases
torchvision==0.29.0  # arquitectura resnet50 (y la referencia de preprocesamiento en los tests de skew)
onnx==1.23.0  # onnx.checker y lectura del opset/shapes
onnxscript==0.7.2  # lo necesita el exportador dynamo (--dynamo)
onnxruntime==1.30.0  # ejecuta el .onnx en CPU para el gate de paridad
numpy==2.5.3
pillow==12.3.0  # MISMA versión que el gateway: el preprocesamiento compartido depende de ella
```

COMANDOS CLAVE

```
python model/export_onnx.py --checkpoint checkpoint.pt --out model/classifier.onnx
```

Exporta y verifica. Sin `--checkpoint` usa pesos aleatorios: solo prueba el pipeline.

```
python model/parity_check.py --checkpoint checkpoint.pt --onnx model/classifier.onnx --images carpeta_muestra_eval --n 2000
```

Paridad en CPU con ONNX Runtime; exit 1 = no pasa. Sin `--checkpoint`, la referencia de PyTorch tendría pesos aleatorios.

```
bash model/build_engine.sh
```

En la T4, con Docker y NVIDIA Container Toolkit: deja `triton/model_repository/classifier/1/model.plan`.

```
python model/parity_check.py --backend triton --triton-url host:8000 --checkpoint checkpoint.pt --images carpeta_muestra_eval --n 2000
```

El engine FP16 servido por Triton contra PyTorch FP32 del mismo checkpoint; exit 1 = no se registra.

```
perf_analyzer -m classifier -u host:8000 --request-rate-range=250:500:250 --percentile=99
```

Carga a tasa fija (250 y 500 req/s) para el gate de latencia: compara el p99 con el del champion.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
Los pods de Triton quedan en CrashLoopBackOff con el engine nuevo.	El engine se compiló en otra GPU u otra versión de TensorRT: por defecto no es portable.	Compara las huellas que imprimió build_engine.sh (GPU, NGC_RELEASE) con la T4 y con el tag tritonserver:24.08 de triton/Dockerfile.

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
export_onnx.py imprime opset=18 y «FALLÓ la verificación».	Usaste --dynamo pidiendo opset 17: la conversión hacia abajo falla en ReduceMean y el archivo queda en 18.	Exporta con el exportador TorchScript (el default del script) o con --dynamo --opset 18.
parity_check.py --backend triton da 97 % de acuerdo top-1.	No es el redondeo de FP16 (mueve los logits ≈ 1e-2): es un bug del export o de una capa del engine. El preprocesamiento queda descartado: los dos modelos reciben el mismo tensor.	Corre primero --backend ort: si FP32 ya da 97 %, el problema está en el export; si pasa, busca la capa que diverge en el engine.
Con batch 1 funciona y con batch 7 falla o devuelve basura.	El eje de batch quedó constante: no se declaró dinámico en el export.	Batch dinámico en el export (verify() prueba 1 y 32) y perfil 1/16/32 en trtexec.
El engine nuevo es más preciso, pero perf_analyzer muestra un p99 2× más alto que el del champion.	Modelo más pesado, o un perfil mal elegido: opt 16 cuando online llegan batches de ≈ 1–3.	perf_analyzer a 250 y 500 req/s; compara con un engine construido con --optShapes=input:4x3x224x224.

✓ LISTA CUANDO…

- [x] export_onnx.py imprime opset=17 y «[export] OK».
- [x] El engine se construyó en una T4 con tensorrt:24.08, la misma release que tritonserver:24.08.
- [x] parity_check.py --backend triton da top-1 agreement ≥ 99.5 % y exit 0.
- [x] perf_analyzer a 250 y 500 req/s: p99 ≤ champion + 10 %.
- [x] Los sha256 del ONNX y del plan, la GPU y la release están en la versión del registry.

CÓMO RECORDARLA

ONNX es el formato de entrega; el engine sirve para un tipo de GPU y una versión de TensorRT. Se compila donde va a correr (T4, NGC 24.08) y se compara contra PyTorch antes de que llegue a producción.

TRAMPA

«construí el engine en mi laptop y lo subí». Por defecto un engine solo sirve en el tipo de GPU y la versión de TensorRT con que se construyó. Hay modos de compatibilidad opcionales (y algo más lentos), pero el de hardware de TensorRT 10.3, `ampere+`, exige compute capability ≥ 8.0 y la T4 es 7.5: se compila en CI, en una T4.

En el curso: C01 · Archivos de referencia | C01 · Evaluación offline y online | M8 · ONNX y ONNX Runtime | M8 · TensorRT

HERRAMIENTAS DE ESTA ESTACIÓN

ONNX

FORMATO DE MODELO

Un formato de archivo estándar que describe la red (operadores, pesos y firma de entradas y salidas) sin depender de PyTorch.

COMO *El PDF de los modelos: lo genera una herramienta y lo leen muchas otras.*

EN EL C01 classifier.onnx (≈ 98 MB, opset 17, eje de batch dinámico que el engine acota a 1–32, tensores input y logits) es el punto de entrega entre entrenamiento y serving: de él compila TensorRT y con él corre la paridad en CPU (ONNX Runtime), sin GPU.

¿CÓMO SE TRABAJA?

{ } Se programa

No es un servicio, es un archivo. Lo produces con una llamada en tu script (`torch.onnx.export`) fijando el opset (la versión de los operadores), los nombres y los ejes dinámicos, y lo verificas con `onnx.checker` y ONNX Runtime. Desde torch 2.9 el exportador por defecto es dynamo; C01 pide el de TorchScript (`dynamo=False`) porque así sale opset 17 limpio.

EJEMPLO

PYTHON reference/c01/model/export_onnx.py (extracto)

```python
        torch.onnx.export(
            model, (example,), out, dynamo=False, opset_version=opset,
            input_names=[INPUT_NAME], output_names=[OUTPUT_NAME],
            dynamic_axes={INPUT_NAME: {0: "batch"}, OUTPUT_NAME: {0: "batch"}},  # eje 0 = batch variable
            do_constant_folding=True,  # precalcula lo constante: grafo más simple para TensorRT
        )
```

DÓNDE CORRE

Es un archivo: lo genera el job de export (CPU) y lo leen trtexec en la T4 y ONNX Runtime en el gate de paridad.

ALTERNATIVAS

Torch-TensorRT, que compila desde PyTorch sin pasar por ONNX. Si el benchmark INT8 en CPU de C01 sale bien, ONNX se queda como formato y cambia el runtime: ONNX Runtime u OpenVINO en vez de TensorRT.

En el curso: M8 · ONNX y ONNX Runtime | C01 · Archivos de referencia

NVIDIA TensorRT

COMPILADOR DE INFERENCIA (GPU NVIDIA)

Un compilador que convierte el ONNX en un «engine» optimizado para un tipo de GPU concreto: fusiona capas, usa FP16 y elige los kernels más rápidos midiendo en esa GPU.

COMO *Como compilar C optimizado para un procesador concreto: ahí vuela; en otro, puede no correr.*

EN EL C01 trtexec --fp16 compila classifier.onnx a model.plan con perfil de batch 1/16/32, en una T4 (sm_75) y dentro del contenedor tensorrt:24.08 (TensorRT 10.3), la misma release que Triton. Triton carga el plan con el backend tensorrt_plan.

¿CÓMO SE TRABAJA?

Se opera

En C01 no se programa: se **opera** corriendo `trtexec` con flags (ONNX de entrada, `--fp16`, shapes mínima, óptima y máxima) dentro del contenedor oficial, desde `build_engine.sh`. TensorRT hace solo el resto: fusiona capas, elige la precisión por capa y cronometra kernels en esa GPU. Ojo: en TensorRT 11 ya no existe `--fp16`; la precisión se fija antes, en el ONNX (ModelOpt AutoCast).

EJEMPLO

BASH reference/c01/model/build_engine.sh (extracto del docker run)

```bash
  "nvcr.io/nvidia/tensorrt:${NGC_RELEASE}-py3" \
  trtexec \
    --onnx="$ONNX" \
    --saveEngine="$PLAN_DIR/model.plan" \
    --fp16 \
    --minShapes=input:1x3x224x224 \
    --optShapes=input:16x3x224x224 \
    --maxShapes=input:32x3x224x224 \
    --shapes=input:16x3x224x224
```

DÓNDE CORRE

Build: contenedor nvcr.io/nvidia/tensorrt:24.08-py3 en un runner de CI con T4. Runtime: dentro de Triton, en la misma T4.

ALTERNATIVAS

ONNX Runtime (CPU o CUDA), OpenVINO en CPU Intel, o Torch-TensorRT desde PyTorch.

En el curso: M8 · TensorRT | C01 · Archivos de referencia | M8 · Cómo validar una optimización

NVIDIA Triton Inference Server

MODEL SERVER

Un servidor de NVIDIA (hoy lo llama Dynamo-Triton) que carga modelos desde una carpeta, junta requests en batches y los corre en la GPU, con API HTTP/gRPC estándar, health checks y métricas.

COMO *La cocina de un restaurante: junta pedidos de muchas mesas y cocina por tandas; no sabe quién pidió qué.*

EN EL C01 Sirve el engine classifier (TensorRT FP16) en cada T4: dynamic batching de 2 ms hasta 32, 2 instancias por GPU, una cola que rechaza a los 100 ms y warmup antes de READY. El modelo va horneado en la imagen tritonserver:24.08, así que el tag de la imagen es la versión del modelo.

¿CÓMO SE TRABAJA?

Se configura | se opera

Casi no escribes código: **configuras**. Armas el model repository (una carpeta por modelo con su `config.pbtxt` y una subcarpeta numerada con `model.plan`) y eliges los flags de arranque (`--model-control-mode=none`, `--strict-readiness=true`). Triton hace solo la cola, el batching, las instancias y las métricas del puerto 8002; lo **operas** con `curl` a `/v2/health/ready`, perf_analyzer y sus logs.

EJEMPLO

TEXTO estructura de reference/c01/triton/ (model.plan lo genera model/build_engine.sh)

```
triton/model_repository/
└── classifier/
    ├── config.pbtxt     # lo escribes tú: contrato + batching + cola
    └── 1/
        └── model.plan   # lo genera model/build_engine.sh (no va en git)
```

DÓNDE CORRE

Contenedor tritonserver:24.08 con el modelo horneado, en nodos GPU T4 de EKS (2 a 4 pods); en local, con docker compose.

ALTERNATIVAS

FastAPI + PyTorch (el MVP de C01), BentoML o Ray Serve; KServe puede ponerse encima; TorchServe está archivado.

En el curso: M7 · NVIDIA Triton Inference Server | C01 · Serving | C01 · Archivos de referencia

Estación 6 de 12 · tramo Release · ● línea de días

Código de serving

¿Qué código y qué configuración ponen al modelo detrás de una API que cumple p99 ≤ 200 ms?

Dos piezas con trabajos distintos: un **gateway FastAPI en CPU** que valida, decodifica en un threadpool, preprocesa con el mismo `preprocess.py`, llama a Triton y arma el top-k, y **Triton en la T4**, configurado en `config.pbtxt` (dynamic batching de 2 ms, 2 instancias, rechazo en cola a los 100 ms). Los 24 tests de pytest prueban el contrato y cada error sin GPU. Es código que cambia en días; el modelo llega ya hecho de la estación 5.

ENTRA

- model.plan (engine) de la versión challenger
- labels.json (500 categorías)
- preprocess.py compartido con el entrenamiento

SALE

- código del gateway + model repository de Triton — VIAJA A 7 · EMPAQUETADO
- 24 tests en verde
- contrato POST /v1/classify

EL PROCESO, PASO A PASO

PASO 1

Fijar el contrato

POST /v1/classify?top_k=5 con los bytes de la imagen; respuesta con model_version; errores 400, 413, 415, 422, 503 y 504.

MLE · gateway/app/main.py

PASO 2

Validar barato primero

Content-Type, 10 MB (por Content-Length y contando el stream), 40 MP y relación 1:8 leídos del header antes de decodificar.

MLE · main.py + preprocess.py

PASO 3

Decode fuera del event loop

run_in_threadpool con un semáforo de 2 decodes por proceso; si no hay lugar en 50 ms, 503 con Retry-After.

MLE · main.py (CPU del pod del gateway)

PASO 4

Llamar a Triton con presupuesto

Cliente async de Triton con tensores binarios; 150 ms de timeout → 504, Triton caído → 503; deadline total de 250 ms.

MLE · main.py → Triton :8000

PASO 5

Configurar Triton

config.pbtxt: tensorrt_plan, max_batch_size 32, ventana de 2 ms, REJECT a los 100 ms, 2 instancias y warmup.

MLE · triton/model_repository/classifier/

PASO 6

Probar sin GPU

pytest con un Triton falso: 24 tests de 200, 4xx, 503, 504, health checks y métricas.

automático (CI) · tu máquina y el job de CI (sin GPU)

ARCHIVOS DEL C01

ARCHIVO REAL **reference/c01/gateway/app/main.py**

El gateway: validación en capas, decode en un threadpool acotado, llamada a Triton con deadline y traducción de cada falla a un código útil (504 si está lento; 503 con Retry-After si está caído o rechaza), métricas y una línea de log por request.

PYTHON extracto

```python
        result = await asyncio.wait_for(
            client.infer(MODEL_NAME, [inp], outputs=[out], request_id=request_id),
            timeout=timeout,  # ≤ INFER_TIMEOUT_S: lo que quede del presupuesto del request
        )
    except TimeoutError:  # asyncio.TimeoutError es alias de TimeoutError desde Python 3.11
        # 504: Triton está vivo pero lento (cola llena, GPU saturada); el cliente cae a su fallback
        raise HTTPException(504, "timeout de inferencia") from None
    except aiohttp.ClientError:  # conexión rechazada/reseteada, DNS: Triton caído o reiniciando
        raise HTTPException(503, "Triton no disponible", headers={"Retry-After": "1"}) from None
    except InferenceServerException as exc:  # Triton respondió error: modelo no cargado, request rechazado
        detail = f"Triton rechazó la inferencia: {exc.message()}"
        raise HTTPException(503, detail, headers={"Retry-After": "1"}) from None
```

ARCHIVO REAL **reference/c01/triton/model_repository/classifier/config.pbtxt**

Contrato del modelo en Triton (input FP32 [3,224,224] → logits [500]) y su comportamiento: dynamic batching de 2 ms, rechazo a los 100 ms en cola, 2 instancias en la T4 y warmup antes de READY. Extracto: el bloque sigue con su llave de cierre.

PBTXT extracto

```
dynamic_batching {
  max_queue_delay_microseconds: 2000
  # Sin preferred_batch_size a propósito: con un solo perfil de optimización (1–32) Triton arma el
  # batch más grande posible dentro de la espera; fijar tamaños solo tiene sentido con varios
  # perfiles/engines por tamaño, y mal elegidos agregan espera.
  default_queue_policy {
    # Protección ante sobrecarga: un request que lleva 100 ms en cola ya no llega a tiempo al
    # timeout del gateway (150 ms). Mejor rechazarlo y liberar GPU para requests que siguen vivos
    # que calcular una respuesta que nadie espera.
    timeout_action: REJECT
    default_timeout_microseconds: 100000
  }
```

ARCHIVO REAL **reference/c01/gateway/tests/test_api.py**

Los 24 tests corren sin GPU: la fixture cambia el cliente de Triton por un fake que respeta su contrato y puede simular a Triton caído, lento o rechazando.

PYTHON extracto

```
@pytest.fixture
def api(monkeypatch):
    """App con el fake inyectado. `with TestClient` ejecuta el lifespan (crea y cierra el cliente)."""
    FakeTriton.mode = "ok"
    monkeypatch.setattr(main.triton_http, "InferenceServerClient", FakeTriton)
    with TestClient(main.app) as client:
        yield client
```

COMANDOS CLAVE

```
cd reference/c01/gateway && pip install -r requirements-dev.txt && pytest
```

Los 24 tests con un Triton falso, en un venv con Python 3.13 (la versión con que se probó); no necesitas GPU.

```
uvicorn app.main:app --port 8080
```

Gateway suelto desde `gateway/`, sin Triton: `/healthz` da 200, `/readyz` 503 y `/v1/classify` 503 con Retry-After.

```
curl -fsS http://localhost:8000/v2/health/ready
```

¿Triton está listo con su modelo cargado? (arranca con `--strict-readiness`).

```
curl -H "Content-Type: image/jpeg" --data-binary @foto.jpg "localhost:8080/v1/classify?top_k=5"
```

Un request de punta a punta contra el gateway, con Triton arriba.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
El p99 crece con la concurrencia sin que la CPU esté llena; hasta /healthz tarda.	Decode de JPEG (o model(x)) dentro de un async def: bloquea el event loop.	Busca trabajo de CPU fuera de run_in_threadpool; compara classifier_preprocess_seconds con la latencia total.
Triton se cae y los pods del gateway empiezan a reiniciarse en cascada.	La dependencia de Triton quedó en liveness en vez de readiness.	Revisa que /healthz no consulte a Triton y /readyz sí (test_healthz_does_not_depend_on_triton).
pytest falla en test_classify_ok_respects_top_k_and_contract después de un cambio en el gateway.	El nombre, la shape o el dtype del tensor ya no coinciden con config.pbtxt (input FP32 [3,224,224], logits [500]).	Lee el assert que falló: el test revisa lo que el gateway le pasó al fake (input, shape [1, 3, 224, 224], FP32 y salida logits), lo mismo que pide config.pbtxt.
Pods del gateway en OOMKilled durante ráfagas de fotos web grandes.	Demasiados decodes a la vez: el threadpool trae 40 hilos y un PNG de 40 MP ocupa ≈ 320 MB al decodificarse.	Revisa PREPROCESS_CONCURRENCY (2 por proceso) frente al límite de memoria del pod (1 GiB).

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
Los primeros requests después de levantar Triton tardan mucho más que el resto.	Falta model_warmup: la primera ejecución paga la inicialización y la readiness pasa con el modelo frío.	Revisa model_warmup en config.pbtxt (batch 32, 3 veces, zero_data).

✓ LISTA CUANDO…

- [x] pytest pasa los 24 tests sin GPU: 200, 400, 413, 415, 422, 503, 504, health checks y métricas.
- [x] /healthz no depende de Triton; /readyz da 200 solo con el modelo listo.
- [x] Los nombres, shapes y dtypes del gateway coinciden con config.pbtxt.
- [x] Ningún trabajo de CPU corre en el event loop; 150 ms de timeout a Triton y 250 ms en total.
- [x] config.pbtxt tiene ventana de 2 ms, REJECT a los 100 ms, 2 instancias y warmup.

CÓMO RECORDARLA

el gateway sabe del producto; Triton, solo de tensores. Gateway: cuotas, bytes, decode, top-k, errores y logs. Triton: cola, batching y GPU.

TRAMPA

un `async def` que decodifica el JPEG. Bloquea el event loop: ningún otro request avanza, ni los health checks, y el p99 sube con la concurrencia sin que la CPU esté llena. El I/O va con `await`; la CPU, a `run_in_threadpool`.

En el curso: C01 · Serving | C01 · Archivos de referencia | M7 · Qué hace un model server (y qué no) | M7 · NVIDIA Triton Inference Server

HERRAMIENTAS DE ESTA ESTACIÓN

FastAPI + Uvicorn

FRAMEWORK WEB (API EN PYTHON)

FastAPI es la librería con la que escribes endpoints HTTP en Python; Uvicorn es el servidor que la ejecuta y atiende las conexiones.

COMO *FastAPI escribe el menú y las reglas del mostrador; Uvicorn abre la puerta y atiende la fila.*

EN EL C01 Es el gateway en CPU: POST /v1/classify valida los bytes, decodifica en un threadpool, preprocesa, llama a Triton con 150 ms de timeout, arma el top-k con labels.json y responde con model_version; además expone /healthz, /readyz y /metrics. Un proceso uvicorn por contenedor.

¿CÓMO SE TRABAJA?

{ } Se programa | se configura

Lo **programas**: `main.py` declara rutas con decoradores y FastAPI valida los parámetros solo (top_k fuera de 1–20 → 422). Lo **configuras** con variables de entorno (`TRITON_URL`, `MODEL_VERSION`, `INFER_TIMEOUT_S`) y flags de uvicorn (`--workers 1`, `--timeout-keep-alive 75`). La regla que más importa: el I/O con `await`; el trabajo de CPU, en un hilo aparte con `run_in_threadpool`.

EJEMPLO

PYTHON reference/c01/gateway/app/main.py (extracto)

```python
@app.post("/v1/classify")
async def classify(request: Request, response: Response, top_k: int = Query(5, ge=1, le=MAX_TOP_K)):
    """Body = bytes crudos de la imagen (no multipart: menos parsing, sin archivos temporales)."""
    t0 = time.perf_counter()  # el reloj arranca con el request ya recibido (el SLO es server-side)
    # Si el cliente (o un proxy) ya trae un id lo respetamos: así se traza el request de punta a punta
    request_id = request.headers.get("x-request-id") or uuid.uuid4().hex
```

DÓNDE CORRE

Imagen python:3.13-slim con un proceso uvicorn, en nodos CPU de EKS (3 a 12 pods con HPA por CPU).

ALTERNATIVAS

Un ensemble de Triton con Python backend; C01 lo descarta porque el decode competiría por la CPU del nodo GPU.

En el curso: M7 · FastAPI + PyTorch: el punto de partida | C01 · Serving | C01 · Archivos de referencia

pytest

TESTS EN PYTHON

La herramienta que encuentra y corre tus funciones test_* y te dice cuáles fallan y por qué.

COMO *Un checklist que CI revisa en cada cambio.*

EN EL C01 Corre los 24 tests del gateway sin GPU ni Triton: un cliente falso reemplaza a tritonclient y se prueban todos los códigos (200, 400, 413, 415, 422, 503, 504), los health checks y las métricas, en tu máquina y en CI antes del build.

¿CÓMO SE TRABAJA?

Se programa · se opera

Lo **programas**: escribes funciones `test_*` con `assert` y fixtures, funciones que preparan lo que el test necesita (la de C01, `api`, usa `monkeypatch` para cambiar el cliente de Triton por uno falso solo durante el test); `pytest.ini` dice dónde buscar. Lo **operas** con un solo comando, `pytest`, en tu máquina y en el job de CI, que frena la release si un test falla.

EJEMPLO

PYTHON reference/c01/gateway/tests/test_api.py (extracto)

```python
def test_slow_triton_is_504(api):
    FakeTriton.mode = "slow"
    r = post(api, image_bytes())
    assert r.status_code == 504  # cortamos a los 0.15 s: el cliente cae a su fallback manual
    assert "X-Request-ID" in r.headers  # los errores también son trazables
```

DÓNDE CORRE

Tu máquina y el job de CI, en CPU; no entra a la imagen de producción.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · C01 · Archivos de referencia

NVIDIA Triton Inference Server

model server · ficha completa en la estación 5 · Export →

se configura · se opera

Estación 7 de 12 · tramo Release · ● línea de días

Empaquetado (Docker)

¿Cómo se convierten el código y el engine en algo que corre igual en tu máquina, en CI y en el cluster?

El C01 empaqueta cada tier en su propia imagen: el gateway (FastAPI, multi-stage sobre `python:3.13-slim`, ≈ 0.1 GB comprimida) y Triton con `config.pbtxt` y `model.plan` **horneados** (≈ 9.2 GB comprimida). Lo probado es, byte a byte, lo que corre, y desplegar o volver atrás es cambiar un tag. El CI construye la del gateway (estación 8), el pipeline del modelo la de Triton en la T4 (estación 5) y `docker compose` las prueba juntas en una máquina con GPU.

ENTRA

- código del gateway y requirements.txt fijado
- model.plan (estación 5) + config.pbtxt (estación 6)

SALE

- Dockerfiles probados (gateway y Triton) — VIAJA A 8 · CI/CD
- docker-compose.yml que levanta todo en una máquina con GPU

EL PROCESO, PASO A PASO

PASO 1

Fijar dependencias

requirements.txt con == en las 37 dependencias, transitivas incluidas, y la misma versión de Pillow que el entrenamiento.

MLE · repo, en tu máquina

PASO 2

Dockerfile del gateway

Multi-stage: el venv se arma en builder y solo él pasa a runtime; usuario 10001 y un solo proceso uvicorn.

MLE · repo, en tu máquina

PASO 3

Dockerfile de Triton

FROM tritonserver:24.08-py3 (mismo YY.MM que el engine) y COPY del model repository; el build falla si falta model.plan.

MLE · repo; la imagen la construye el pipeline del modelo en la T4 (estación 5)

PASO 4

docker build

Para linux/amd64, con tag = commit (gateway) o versión del modelo (Triton); el .dockerignore del gateway deja fuera tests, secretos y modelos.

MLE · tu máquina o una g4dn de desarrollo; las imágenes oficiales salen del CI (gateway) y del pipeline del modelo (Triton)

PASO 5

Probar todo con compose

docker compose up --build: Triton con GPU y healthcheck, el gateway espera service_healthy y Prometheus carga las 6 alertas.

MLE · máquina Linux con GPU NVIDIA y NVIDIA Container Toolkit

PASO 6

Revisar peso y capas

≈ 0.1 GB el gateway contra ≈ 9.2 GB Triton, comprimidas (lo que baja cada nodo): el pull de Triton es el que alarga el cold start.

MLE · tu máquina

ARCHIVOS DEL C01

ARCHIVO REAL **reference/c01/gateway/Dockerfile**

Etapa runtime del gateway: copia solo el venv de la etapa builder, corre como el UID 10001 y trae un HEALTHCHECK que solo usan docker run y compose.

DOCKERFILE extracto

```
# Solo el venv ya resuelto: pip, compiladores y caches de la etapa builder se quedan atrás
COPY --from=builder /opt/venv /opt/venv
# El código queda con dueño root y solo lectura para "app": el proceso no puede modificarse
COPY app/ ./app/
USER 10001
# Documenta el puerto (no lo publica): el Service de k8s y compose apuntan aquí
EXPOSE 8080
# HEALTHCHECK lo usan `docker run`/compose. Kubernetes lo IGNORA: allá mandan liveness/readiness
# probes. Sin curl en la imagen slim, se usa Python (stdlib) para pegarle a /healthz.
HEALTHCHECK --interval=10s --timeout=2s --start-period=10s --retries=3 \
    CMD ["python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8080/healthz', timeout=1)"]
```

ARCHIVO REAL **reference/c01/triton/Dockerfile**

Arranque de Triton: carga todo al iniciar y no cambia nada en caliente, muere si el modelo no carga y solo da ready con el modelo listo.

DOCKERFILE extracto

```
CMD ["tritonserver", \
     "--model-repository=/models", \
     "--model-control-mode=none", \
     "--disable-auto-complete-config", \
     "--exit-on-error=true", \
     "--strict-readiness=true", \
     "--http-port=8000", \
     "--grpc-port=8001", \
     "--metrics-port=8002", \
     "--exit-timeout-secs=30", \
     "--log-format=ISO8601"]
```

ARCHIVO REAL **reference/c01/docker-compose.yml**

El gateway en compose: la misma configuración por variables de entorno que en Kubernetes, y no arranca hasta que Triton está sano.

YAML extracto

```
  gateway:
    build: ./gateway  # imagen multi-stage de gateway/Dockerfile
    ports:
      - "8080:8080"  # la API pública: POST /v1/classify
    environment:  # misma configuración que en k8s, solo cambian los valores
      TRITON_URL: "triton:8000"  # nombre del servicio en la red de compose, sin http://
      MODEL_NAME: "classifier"
      MODEL_VERSION: "2026-09-01-resnet50-fp16"  # lo devuelve la API y queda en cada línea de log
      INFER_TIMEOUT_S: "0.15"
    depends_on:
      triton:
        condition: service_healthy  # no arranca hasta que el modelo esté listo (evita 503 al inicio)
```

ARCHIVO REAL **reference/c01/gateway/.dockerignore**

Lo que nunca entra al contexto de build: secretos locales, imágenes de prueba y modelos (el modelo vive en Triton).

TEXTO extracto

```
# Secretos locales: jamás dentro de una imagen (se inyectan como env/Secret en runtime)
.env
*.pem
# Imágenes de prueba y modelos en cualquier carpeta (**/): el gateway no los necesita (el modelo vive en Triton)
**/*.jpg
**/*.jpeg
**/*.png
**/*.onnx
**/*.plan
```

COMANDOS CLAVE

```
docker build --platform linux/amd64 -t classifier-gateway:tag gateway/
```

En un Mac con Apple Silicon `--platform` es obligatorio: los nodos c6i y g4dn son x86. El tag es el commit.

```
docker build -t classifier-triton:version_modelo triton/
```

Falla a propósito si falta `model.plan`: primero `build_engine.sh` en la T4.

```
docker compose up --build
```

Desde `reference/c01`: gateway, Triton con GPU y Prometheus en una máquina.

```
curl -s -H "Content-Type: image/jpeg" --data-binary @foto "localhost:8080/v1/classify?top_k=5"
```

La prueba de punta a punta: top-5 con `model_version`.

```
docker history classifier-gateway:tag
```

Una fila por capa: encuentra la que infló la imagen.

```
docker buildx imagetools inspect --format '{{json .Image}}' registry/classifier-gateway:tag
```

Antes de desplegar: el config tiene que decir `"architecture": "amd64"`. Sin `--format`, una imagen de una sola plataforma no muestra la suya.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
En EKS el pod muere con exec format error (CrashLoopBackOff) o el pull falla con no match for platform in manifest.	La imagen se construyó en un Mac con Apple Silicon: salió arm64 y los nodos son x86.	`docker buildx imagetools inspect --format '{{json .Image}}' registry/classifier-gateway:tag` : ¿dice "architecture": "amd64"?
docker build triton/ se detiene en RUN test -f /models/classifier/1/model.plan.	Falta el engine: model.plan no se versiona en git y lo genera build_engine.sh en la T4.	`ls triton/model_repository/classifier/1/ y` , si está vacío, bash model/build_engine.sh en la g4dn.
Editas solo main.py y el build reinstala todas las dependencias.	COPY . . antes de pip install: la caché de capas se rompe hacia abajo.	`docker history` : requirements.txt tiene que copiarse e instalarse antes de COPY app/.
Pod en CreateContainerConfigError: container has runAsNonRoot and image will run as root.	El pod exige runAsNonRoot y la imagen corre como root (la de NGC lo hace por defecto) o no declara un UID numérico.	`kubectl describe pod pod` : USER 10001 en el Dockerfile del gateway y runAsUser: 1000 para Triton.
En compose el gateway da 503 al arrancar, o Triton avisa WARNING: The NVIDIA Driver was not detected.	depends_on sin condition: service_healthy, o el servicio de Triton sin la reserva de GPU (deploy.resources.reservations.devices).	`docker compose ps` (Triton en health: starting) y docker compose logs -f triton.

✓ LISTA CUANDO…

- [x] Las dos imágenes se construyen para linux/amd64, y la del gateway no lleva tests, secretos ni modelos.
- [x] El gateway corre como UID 10001 con un solo proceso uvicorn y /healthz responde 200.
- [x] La imagen de Triton trae config.pbtxt y model.plan, y su tag es la versión del modelo.
- [x] docker compose up --build devuelve un top-5 en /v1/classify y localhost:9090/alerts muestra las 6 alertas.
- [x] Sabes cuánto pesa cada imagen comprimida: ≈ 0.1 GB el gateway y ≈ 9.2 GB Triton.

CÓMO RECORDARLA

receta → imagen → contenedor. El `Dockerfile` es la receta, la imagen es el resultado inmutable (su digest es su identidad) y el contenedor es el proceso que corre a partir de ella; lo que cambia poco va arriba en la receta y lo que cambia en cada commit, abajo.

TRAMPA

creer que el `HEALTHCHECK` del Dockerfile cuida al pod. Lo usan `docker run` y compose; Kubernetes lo ignora y manda con las probes del manifest (estación 9).

En el curso: M5 · Un Dockerfile de ML, decisión por decisión · M5 · docker compose para desarrollo local · M5 · Reducir tamaño (y por qué importa para el cold start) · C01 · Archivos de referencia

HERRAMIENTAS DE ESTA ESTACIÓN

Docker (+ docker compose)

EMPAQUETADO DE CONTENEDORES

Una herramienta que empaqueta tu código con todo lo que necesita (Python, librerías, CUDA) en una imagen inmutable y la corre como un contenedor aislado.

COMO *La imagen es como un checkpoint del entorno completo, congelado en un archivo que nadie modifica; el contenedor es el proceso que corre a partir de ella, y de una imagen salen muchas réplicas idénticas.*

EN EL C01 Dos imágenes: classifier-gateway (FastAPI, ≈ 0.1 GB comprimida) y classifier-triton con el engine horneado (≈ 9.2 GB comprimida). docker compose levanta gateway, Triton y Prometheus en una máquina con GPU para desarrollo y pruebas de integración.

¿CÓMO SE TRABAJA?

Se configura · se opera

Escribes un `Dockerfile`, una receta de instrucciones de build (no un programa): de qué base partir, qué copiar, qué instalar, con qué usuario y con qué comando arrancar; para varios servicios, un `docker-compose.yml`. Docker arma las capas, las cachea y corre el proceso aislado. Lo operas con comandos: `docker build`, `run`, `logs`, `push` y `compose up`.

EJEMPLO

DOCKERFILE reference/c01/triton/Dockerfile (extracto)

```
FROM nvcr.io/nvidia/tritonserver:24.08-py3

# El model repository completo: classifier/config.pbtxt + classifier/1/model.plan
COPY model_repository /models

# Falla el BUILD si falta el engine (no se versiona en git): mejor aquí que un pod en CrashLoop
RUN test -f /models/classifier/1/model.plan

# 8000 HTTP/REST (KServe v2, lo usa el gateway) · 8001 gRPC · 8002 métricas Prometheus (nv_*)
EXPOSE 8000 8001 8002
```

DÓNDE CORRE

En tu máquina, en la máquina de desarrollo con GPU y en el runner de CI. En el cluster, las imágenes las corre el runtime de cada nodo (containerd), no Docker; la GPU necesita un host Linux con driver NVIDIA y NVIDIA Container Toolkit.

ALTERNATIVAS

Podman, que construye y corre las mismas imágenes OCI; para producción con varias réplicas y zonas, Kubernetes en lugar de compose.

En el curso: M5 · Un Dockerfile de ML, decisión por decisión · M5 · docker compose para desarrollo local · Docker y kubectl · Docker: construir, correr, depurar

Estación 8 de 12 · tramo Release · ● línea de días

CI/CD y registro de imágenes

¿Cómo llega un commit a una imagen probada, inmutable y lista para desplegar?

Transbordo · Registro de imágenes (Amazon ECR). Aquí se tocan las líneas ● semanas, ● días y ● continuo. Qué cruza

Cada commit pasa por CI: los 24 tests del gateway corren sin GPU y su imagen se construye **una sola vez** para `linux/amd64` y sube a ECR, donde los tags no se mueven. La de Triton la arma el pipeline del modelo en la T4 antes de registrar la versión (estación 5), con la versión del modelo como tag. Desde el push todo habla del **digest**: se escanea, se firma y staging y producción despliegan el mismo.

ENTRA

- commit con código, Dockerfiles y manifests
- imagen de Triton del pipeline del modelo (estación 5)

→

SALE

- imagen en ECR (tag inmutable + digest) **VIAJA A 9 · KUBERNETES**
- digest firmado listo para desplegar

EL PROCESO, PASO A PASO

PASO 1

Commit y pull request

El hash corto del commit (3f9c2ab) será el tag del gateway: desde un pod en producción llegas al código exacto.

MLE

tu máquina → GitHub

PASO 2

Tests sin GPU

pytest corre los 24 tests del gateway contra un Triton falso; el curso suma un test de tensores de referencia del preprocesamiento.

automático (CI)

runner de CI en CPU

PASO 3

Build del gateway y push

docker buildx build --platform linux/amd64 --push: la imagen sube una sola vez a un repo de ECR con tags inmutables.

automático (CI)

runner de CI → Amazon ECR

PASO 4

Llega la imagen de Triton

Solo con un modelo nuevo: el pipeline la arma en la T4 (build_engine.sh → docker build triton/ → paridad --backend triton) y la sube con la versión como tag.

automático (pipeline)

runner con T4 (g4dn.xlarge) → Amazon ECR

PASO 5

Digest, escaneo y firma

Cada push devuelve un digest sha256: el escáner busca CVEs y cosign firma ese digest.

automático (CI)

Amazon ECR y runner de CI

PASO 6

Apuntar los manifests

El manifest del gateway pasa al tag nuevo (mejor aún, al digest) con un PR al repo de manifests (GitOps) o kubectl set image; un modelo nuevo sigue la escalera de la estación 10.

automático (CD)

repo de manifests → EKS

ARCHIVOS DEL C01

DESCRITO EN EL CURSO — **.github/workflows/ (workflow de CI)**

El curso propone GitHub Actions (o el CI que ya use la empresa) con credenciales OIDC de corta vida hacia AWS: tests en cada commit, build único y push. El repo de referencia no trae el workflow.

ARCHIVO REAL **reference/c01/gateway/tests/test_api.py**

Los tests que corre el CI: un Triton falso con el contrato real del cliente, así detectan si cambia el nombre, el shape o el dtype del tensor.

PYTHON extracto

```python
"""Tests del gateway SIN GPU ni Triton: un cliente falso reemplaza a tritonclient (caso c01).

Por qué un fake y no un mock genérico: el fake respeta el contrato real del cliente aio (mismos
métodos async, mismas excepciones) y el gateway sigue construyendo InferInput/InferRequestedOutput
REALES, así que el test detecta si alguien cambia el nombre, shape o dtype del tensor.
Correr: cd gateway && pytest
"""
```

ARCHIVO REAL **reference/c01/README.md**

El orden real del C01: en la máquina con T4 se construyen el engine y la imagen de Triton, se levanta Triton con esa imagen y solo con la paridad aprobada la versión pasa al registry y al canary.

TEXTO extracto

```
checkpoint .pt ──export_onnx.py──▶ classifier.onnx ──parity_check.py (ORT, CPU)──▶ ¿pasa?
      └─ en la máquina con T4: build_engine.sh ──▶ model.plan ──docker build triton/──▶ imagen
         └─ Triton arriba ──parity_check.py --backend triton (FP16 vs FP32)──▶ ¿pasa? ──▶ registry / canary
```

DESCRITO EN EL CURSO **release.sh (M5 · registries)**

Script de ejemplo del curso: buildx con tag = commit y --push a ECR, digest con crane, firma keyless con cosign y kubectl set image por digest. No está en reference/c01.

ARCHIVO REAL **reference/c01/k8s/gateway.yaml**

Lo que consume el cluster: la imagen de ECR con el commit como tag inmutable, nunca :latest.

YAML extracto

```yaml
        - name: gateway
          # Tag = commit de git (inmutable); nunca :latest (no sabrías qué versión corre ni cómo volver)
          image: 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-gateway:3f9c2ab
```

COMANDOS CLAVE

```
cd reference/c01/gateway && pip install -r requirements-dev.txt && pytest
```

24 tests sin GPU ni Triton: lo primero que corre el CI.

```
aws ecr get-login-password --region region | docker login --username AWS --password-stdin registry
```

Login a ECR; el token dura 12 h. En CI, con un rol asumido por OIDC, sin llaves guardadas.

```
docker buildx build --platform linux/amd64 -t registry/classifier-gateway:sha --push gateway/
```

Build único, para la arquitectura de los nodos; *sha* son los primeros 7 caracteres del commit: `git rev-parse --short=7 HEAD` (en CI, `${GITHUB_SHA::7}`).

```
crane digest registry/classifier-gateway:sha
```

El digest es la identidad: es lo que se firma y se despliega.

```
cosign sign --yes registry/classifier-gateway@digest
```

Firma keyless con la identidad OIDC del CI.

```
kubectl -n classifier set image deployment/gateway gateway=registry/classifier-gateway@digest
```

Despliegue por digest; con GitOps ese valor va en un PR al repo de manifests.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
El push a ECR falla con ImageTagAlreadyExistsException.	El repo tiene tags inmutables y el CI intentó subir otra imagen con un tag que ya existe (un :latest o un commit reconstruido).	El tag sale de git rev-parse --short=7 HEAD de un commit nuevo: nunca se reusa ni se mueve.
Staging pasó todo, pero producción se comporta distinto.	Se reconstruyó la imagen para producción: otra base, otras transitivas, otro digest.	Compara el digest que corre en cada entorno (el imageID de sus pods) con el de crane digest; promover es apuntar al mismo.
Después del deploy, Triton queda en CrashLoopBackOff con Exit Code 1.	La imagen se armó fuera del pipeline del modelo (engine de otra GPU o de otro YY.MM de NGC) y nadie levantó Triton con ella antes de registrarla.	`kubectl logs pod -n classifier --previous` ; el arreglo va al pipeline que construye el engine y la imagen.
Los pods nuevos quedan en ImagePullBackOff: Failed to pull image .../classifier-gateway... not found.	El tag que pide el manifest no está en ECR: el build o el push de CI falló (o no corrió) y el manifest se actualizó igual. Menos seguido: nodos sin permiso de lectura o una imagen sin linux/amd64.	`aws ecr describe-images --repository-name classifier-gateway --image-ids imageTag=tag` ; si no existe, revisa el job de CI de ese commit.
Quieres hacer rollback y la imagen anterior ya no está en ECR.	Una lifecycle policy por antigüedad (más de 30 días) borró la release estable.	La lifecycle policy cuenta releases (conserva las últimas N) y nunca toca lo desplegado.

✓ LISTA CUANDO…

- [x] pytest pasa los 24 tests del gateway en cada commit, sin GPU.
- [x] Cada imagen se construyó una sola vez (el gateway en CI, Triton en el pipeline del modelo), con tag = commit o versión del modelo, en repos de ECR con tags inmutables.
- [x] La imagen de Triton llegó del pipeline del modelo, construida en la T4 y con la paridad FP16 contra PyTorch aprobada sobre esa misma imagen.
- [x] El digest está escaneado y firmado, y staging y producción apuntan a ese mismo digest.
- [x] La lifecycle policy conserva las últimas releases: el rollback siempre tiene a dónde volver.

CÓMO RECORDARLA

build once, promote the digest. Cada imagen se construye una sola vez (el gateway en CI, Triton en el pipeline del modelo); el tag dice de dónde salió (commit o versión del modelo) y el digest dice qué bytes corren en cada entorno.

TRAMPA

`:latest` o reconstruir para producción. Con un tag que se mueve no sabes qué versión corre ni a cuál volver, y reconstruir el mismo commit produce otra imagen con otro digest.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M5 · Registries y tags inmutables · Herramientas por categoría · Infraestructura como código, CI y seguridad · M15 · Supply chain: imágenes, dependencias y pesos verificados

HERRAMIENTAS DE ESTA ESTACIÓN

Git / GitHub

CONTROL DE VERSIONES

Git guarda la historia de cada cambio del repo (código, Dockerfiles, manifests); GitHub aloja el repo y los pull requests donde se revisan esos cambios.

COMO *Un historial de checkpoints de todo el repo: cada commit es una foto exacta a la que puedes volver.*

EN EL C01 Cada commit identifica una versión exacta: su hash corto (3f9c2ab) es el tag de la imagen del gateway. Con GitOps, los manifests de producción viven en git y desplegar o volver atrás es un PR o un revert.

¿CÓMO SE TRABAJA?

Se opera · administrado

No se programa ni se configura: lo **operas** con comandos (`git commit` , `git push` , abrir un PR), y GitHub es un servicio **administrado** donde vive el repo. Lo que pones en git (código, `Dockerfile` , `k8s/*.yaml`) es lo que dispara el CI y, con GitOps, lo que el cluster copia.

EJEMPLO

BASH release.sh del curso (M5 · registries), extracto

```bash
REG=123456789012.dkr.ecr.us-east-1.amazonaws.com
SHA=$(git rev-parse --short=7 HEAD)

# 1) CI construye UNA vez, para la arquitectura de los nodos, con el commit como tag
docker buildx build --platform linux/amd64 -t $REG/classifier-gateway:$SHA --push gateway/
```

DÓNDE CORRE

Git en tu máquina y en el runner de CI; GitHub como servicio en la nube.

ALTERNATIVAS

GitLab o Bitbucket para alojar el repo.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M5 · Registries y tags inmutables

GitHub Actions

CI/CD

El servicio de GitHub que corre solo tus pasos (tests, builds, pushes) en una máquina limpia cada vez que hay un push o un pull request.

COMO *Un robot que, en cada commit, repite en una máquina limpia exactamente los pasos que harías a mano.*

EN EL C01 El curso lo nombra como CI por defecto (o el que ya use la empresa): pytest del gateway en cada commit, build único de su imagen y push a ECR con credenciales OIDC de corta vida. El engine y la imagen de Triton los arma el pipeline del modelo en un runner con T4. El repo de referencia no trae un workflow.

¿CÓMO SE TRABAJA?

Se configura · administrado

Escribes un workflow en YAML dentro de `.github/workflows/` : cuándo corre (`on`), qué jobs y en qué máquina (`runs-on`) y sus pasos (`run` para comandos, `uses` para acciones ya hechas). GitHub pone los runners y ejecuta todo solo; los jobs corren en paralelo salvo que declares `needs` . Para entrar a AWS, el job declara `id-token: write` en `permissions` , pide un token OIDC y AWS lo cambia por credenciales temporales de un rol: no hay llaves guardadas.

EJEMPLO

YAML ilustrativo, con la sintaxis de docs.github.com: el C01 no trae un workflow

```yaml
on: [push, pull_request]
jobs:
  gateway-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-python@v7
        with: {python-version: "3.13"}
      - run: pip install -r requirements-dev.txt && pytest
        working-directory: reference/c01/gateway
```

DÓNDE CORRE

Runners hospedados por GitHub (CPU) para tests y builds; para el engine y la imagen de Triton, un runner con T4: self-hosted en una g4dn que se prende solo para el job, o un runner GPU de GitHub.

ALTERNATIVAS

GitLab CI, o el CI que ya use la empresa (Jenkins, Buildkite, CodeBuild, Cloud Build).

En el curso: M10 · CI/CD para modelos: la pirámide de tests · Herramientas por categoría · Infraestructura como código, CI y seguridad

Amazon ECR

REGISTRO DE IMÁGENES

El registro privado de imágenes de AWS: el almacén desde donde los nodos del cluster descargan cada imagen.

COMO *Un almacén con cajas etiquetadas: con tags inmutables, una etiqueta nunca cambia de caja.*

EN EL C01 Guarda classifier-gateway (tag = commit) y classifier-triton (tag = versión del modelo) con tags inmutables, en 123456789012.dkr.ecr.us-east-1.amazonaws.com; los manifests apuntan ahí. Una lifecycle policy debe conservar las últimas releases para el rollback.

¿CÓMO SE TRABAJA?

Administrado · se configura · se opera

Es un servicio **administrado**: no instalas nada. **Configuras** cada repositorio (tags inmutables, lifecycle policy) y el escaneo del registry, y lo **operas** con `docker push` desde CI, después de autenticarte con `aws ecr get-login-password`. ECR guarda las capas, escanea y aplica la lifecycle policy solo; con tags inmutables, subir otra imagen con un tag que ya existe falla con `ImageTagAlreadyExistsException`.

EJEMPLO

BASH ilustrativo: comandos de la documentación de ECR con los nombres del C01

```
aws ecr create-repository --repository-name classifier-gateway --image-tag-mutability IMMUTABLE
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 123456789012.dkr.ecr.us-east-1.amazonaws.com
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-gateway:3f9c2ab
```

DÓNDE CORRE

Servicio regional de AWS (us-east-1 en el C01). Los nodos bajan los manifiestos por los endpoints de ECR y las capas desde S3, así que conviene el gateway endpoint de S3 en la VPC.

ALTERNATIVAS

Artifact Registry (Google Cloud), Azure Container Registry, GitHub Container Registry replicado a un registry de tu nube para producción.

En el curso: M5 · Registries y tags inmutables · M13 · Costos ocultos: recorre los bytes

pytest

tests en Python · ficha completa en la estación 6 · Serving →

se programa · se opera

Estación 9 de 12 · tramo Operación · ● línea de días

Kubernetes

¿Quién decide dónde corre cada pod, cuántos hay y cómo les llega el tráfico?

Transbordo · El cluster (Amazon EKS). Aquí se tocan las líneas ● días, ● milisegundos y ● continuo. Qué cruza

En EKS el C01 corre dos Deployments: el gateway en nodos de CPU (3–12 pods, HPA por CPU) y Triton en nodos `g4dn.xlarge`, una T4 por pod (2–4 pods, KEDA por tiempo en cola), repartidos entre las dos zonas. Tú escribes en YAML el estado deseado; Kubernetes **reconcilia** sin parar (compara lo pedido con lo que corre y corrige la diferencia): agenda cada pod donde cabe, lo reinicia si muere y lo saca del balanceo si no está listo. A partir del Ingress, el AWS Load Balancer Controller crea el ALB.

ENTRA

- imagen en ECR (tag + digest)
- manifests k8s/*.yaml

→

SALE

- pods listos detrás del ALB interno — VIAJA A 10 · ROLLOUT
- HPA y ScaledObject escalando solos

EL PROCESO, PASO A PASO

PASO 1 — Preparar el cluster

EKS con node group g4dn.xlarge (label workload=gpu-t4 y taint nvidia.com/gpu), device plugin, Load Balancer Controller, KEDA y Prometheus.

plataforma/SRE · Amazon EKS (IaC y Helm)

PASO 2 — Escribir los manifests

YAML con el estado deseado: Deployments, Services, HPA, ScaledObject, Ingress, PodDisruptionBudget, NetworkPolicy y ServiceMonitors.

MLE · repo (k8s/*.yaml)

PASO 3 — kubectl apply

Namespace classifier con el readiness gate del ALB y apply de los 4 archivos: el API server guarda lo que pediste.

MLE · tu terminal → API server de EKS

PASO 4 — Agendar y arrancar pods

El scheduler pone cada Triton en una T4 libre, repartidos entre zonas; el kubelet baja la imagen y corre startup, readiness y liveness.

automático (Kubernetes) · nodos g4dn.xlarge (T4) y nodos de CPU

PASO 5 — Exponer con Service e Ingress

Services ClusterIP para el gateway y para Triton (este con ruteo por zona); el Load Balancer Controller crea el ALB interno con target-type ip.

automático (Kubernetes) · EKS y ALB de AWS

PASO 6 — Escalar solo

HPA del gateway por CPU al 60 % (3–12 pods) y KEDA para Triton por tiempo en cola con objetivo de 10 ms (2–4 pods).

automático (Kubernetes) · control plane de EKS, KEDA y Karpenter para los nodos

PASO 7 — Reconciliar siempre

Si un pod muere o cae un nodo, los controladores lo recrean; la readiness saca del balanceo al pod que no está listo.

automático (Kubernetes) · control plane de EKS y kubelet de cada nodo

ARCHIVOS DEL C01

ARCHIVO REAL **reference/c01/README.md**

Lo que plataforma deja listo antes del primer apply: device plugin, nodos T4 con label y taint, Load Balancer Controller, KEDA, Prometheus y un CNI que aplique NetworkPolicy.

TEXTO extracto

```
Prerequisitos del cluster: NVIDIA device plugin, node group de `g4dn.xlarge` con label
`workload=gpu-t4` y taint `nvidia.com/gpu=present:NoSchedule`, AWS Load Balancer Controller, KEDA,
Prometheus (kube-prometheus-stack, con kube-state-metrics), un CNI que aplique NetworkPolicy (VPC CNI
con `enableNetworkPolicy=true`, o Calico) y las imágenes en ECR. Kubernetes ≥ 1.35: 1.34 sale del
soporte upstream el 27 de octubre de 2026 y del soporte estándar de EKS el 2 de diciembre de 2026. Nada
en los manifests depende de 1.34: pasan la validación estricta contra 1.35–1.37 y
`trafficDistribution: PreferSameZone` es GA desde 1.35 (en 1.31–1.33 cámbialo por `PreferClose`).
```

ARCHIVO REAL **reference/c01/k8s/triton.yaml**

Adónde va Triton: solo a nodos T4 (label workload=gpu-t4) y con permiso para el taint que aleja a los pods de CPU; 60 s para apagarse.

YAML extracto

```yaml
    spec:
      # Tiempo total para apagar: preStop (10 s) + --exit-timeout-secs de Triton (30 s) + margen
      terminationGracePeriodSeconds: 60
      nodeSelector:
        workload: gpu-t4  # solo el node group de g4dn.xlarge (T4, sm_75): el engine se compiló para esa GPU
      tolerations:
        # Los nodos GPU tienen el taint nvidia.com/gpu=present:NoSchedule para que ningún pod CPU
        # (gateway, sistema) ocupe una máquina cara; Triton lo tolera explícitamente.
        - key: nvidia.com/gpu
          operator: Equal
          value: present
          effect: NoSchedule
```

ARCHIVO REAL **reference/c01/k8s/gateway.yaml**

Readiness del gateway: recibe tráfico solo si el modelo está listo en Triton. La liveness (/healthz) no mira a Triton.

YAML extracto

```yaml
          # readiness: ¿mandarle tráfico? /readyz = modelo listo en Triton (ver main.py)
          readinessProbe:
            httpGet:
              path: /readyz
              port: http
            periodSeconds: 5
            timeoutSeconds: 2
            failureThreshold: 2
```

ARCHIVO REAL **reference/c01/k8s/keda-triton.yaml**

El trigger de KEDA: espera media en la cola de Triton en ms contra un objetivo de 10; una query vacía es un error visible, no un 0.

YAML extracto

```yaml
        ignoreNullValues: "false"
        # Espera media en cola en ms (misma fórmula que la alerta TritonQueueTimeHigh).
        # clamp_min evita 0/0 = NaN sin tráfico; si todo falla (0 éxitos) el valor se dispara y escala.
        query: |
          sum(rate(nv_inference_queue_duration_us{model="classifier"}[2m]))
            /
          clamp_min(sum(rate(nv_inference_request_success{model="classifier"}[2m])), 1e-9)
            / 1000
        # Objetivo: 10 ms en cola. Normal con max_queue_delay de 2 ms es ≈ 1–3 ms; > 10 ms es saturación.
        # Ojo: la cola es no lineal (casi 0 hasta que la GPU se satura). Alternativa más estable:
        # AverageValue sobre sum(rate(nv_inference_count[1m])) con umbral ≈ 400 img/s por réplica.
        threshold: "10"
```

COMANDOS CLAVE

```
kubectl create namespace classifier
```

Una vez por cluster.

```
kubectl label namespace classifier elbv2.k8s.aws/pod-readiness-gate-inject=enabled
```

Readiness gate del ALB: el rolling update no mata un pod viejo hasta que el ALB ve sano al nuevo.

```
kubectl apply -f k8s/triton.yaml -f k8s/keda-triton.yaml -f k8s/gateway.yaml -f k8s/monitoring.yaml
```

Declarativo: guardas el estado deseado y los controladores hacen el resto.

```
kubectl get pods -n classifier -o wide
```

Fase, READY, reinicios y nodo: ¿un Triton por zona?

```
kubectl describe pod pod -n classifier
```

Primera parada si algo no arranca: los Events del final dicen por qué.

```
kubectl get nodes -L node.kubernetes.io/instance-type,topology.kubernetes.io/zone,workload
```

Tipo, zona y label `workload=gpu-t4` de cada nodo.

```
kubectl get hpa -n classifier
```

El HPA del gateway y `keda-hpa-triton`, con su métrica actual contra el objetivo.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
Pod de Triton en Pending: 0/6 nodes are available: 2 Insufficient nvidia.com/gpu, …	Todas las T4 están ocupadas y el nodo nuevo no llega: tope del node group, cuota de vCPU de instancias G o sin capacidad en la zona.	`kubectl describe pod pod -n classifier` y los eventos del autoscaler de nodos (Karpenter o Cluster Autoscaler).
Pending con node(s) had untolerated taint(s) o didn't match Pod's node affinity/selector.	Falta la toleration de nvidia.com/gpu o ningún nodo tiene la label workload=gpu-t4.	`kubectl describe node nodo` (bloque Taints) y kubectl get nodes -L workload.
El nodo GPU existe, pero su capacidad no muestra nvidia.com/gpu.	El NVIDIA device plugin no corre en ese nodo (o falló el driver): para Kubernetes no hay GPU.	`kubectl get ds -A | grep -i nvidia` y los logs de ese DaemonSet.
La cola de Triton sube y KEDA no escala; el ScaledObject marca READY en False.	Prometheus no scrapea Triton (falta el ServiceMonitor): la query vuelve vacía y, con ignoreNullValues en false, es un error.	`kubectl describe scaledobject triton -n classifier` (eventos KEDAScalerFailed).
Los gateways reinician bajo carga: Liveness probe failed … context deadline exceeded.	Liveness con un timeout corto contra un proceso saturado, o una liveness que revisa dependencias.	`kubectl describe pod pod -n classifier` ; la liveness solo mira el proceso y las dependencias van en la readiness.

✓ LISTA CUANDO…

- [x] kubectl get pods -o wide muestra 2 Triton READY en nodos de zonas distintas y al menos 3 gateways.
- [x] Los nodos GPU anuncian nvidia.com/gpu: 1, tienen el taint y ningún gateway corre en ellos.
- [x] kubectl get hpa muestra el HPA del gateway y keda-hpa-triton con métricas reales, sin unknown.
- [x] El Ingress tiene la dirección del ALB interno y sus targets pasan el health check en /readyz.
- [x] Los ServiceMonitors existen y las alertas cargaron como PrometheusRule.

CÓMO RECORDARLA

en Kubernetes **declaras, no ejecutas**. Escribes en YAML el estado deseado (imagen, réplicas, recursos, probes) y los controladores trabajan sin parar para que el cluster se parezca a eso, como un termostato.

TRAMPA

poner una dependencia en la liveness. Si `/healthz` preguntara por Triton, una caída de Triton reiniciaría en cascada gateways sanos; la dependencia va en la readiness (`/readyz`), que solo saca al pod del balanceo.

En el curso: M6 · Los objetos que sí debes conocer · M6 · GPUs en Kubernetes · M6 · HPA: por qué escalar por CPU no sirve para un servicio de GPU · M6 · KEDA: escalar por eventos y hasta cero · M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica

HERRAMIENTAS DE ESTA ESTACIÓN

Kubernetes (Amazon EKS) + kubectl

ORQUESTACIÓN DE CONTENEDORES

Un sistema que mantiene corriendo tus contenedores en un grupo de máquinas: decide dónde va cada uno, los reinicia si mueren, los escala y les da un nombre estable en la red.

COMO *Un termostato: tú fijas la temperatura (el estado deseado, en YAML) y él prende y apaga equipos hasta que el cuarto se parece a eso.*

EN EL C01 EKS corre el gateway (3–12 pods en nodos de CPU) y Triton (2–4 pods, una T4 cada uno, repartidos entre las dos zonas), con probes, PodDisruptionBudget, NetworkPolicy, HPA, KEDA y un Ingress que se vuelve un ALB interno.

¿CÓMO SE TRABAJA?

Se configura · se opera · administrado

No se programa: escribes manifests YAML con el **estado deseado** (qué imagen, cuántas réplicas, qué recursos, qué probes) y los aplicas con `kubectl apply`. Los controladores comparan sin parar lo deseado con lo real y corrigen la diferencia (eso es **reconciliar**): agendan, reinician, escalan. Lo **operas** con `kubectl` (get, describe, logs, rollout) y en EKS el control plane lo **administra** AWS (≈ US$73/mes).

EJEMPLO

YAML reference/c01/k8s/gateway.yaml (extracto)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gateway
  namespace: classifier
  labels:
    app.kubernetes.io/name: gateway
    app.kubernetes.io/part-of: classifier
spec:
  # Sin "replicas": las decide el HPA de abajo (3–12)
```

DÓNDE CORRE

El control plane lo opera AWS (EKS); los pods corren en nodos EC2 de tu cuenta: c6i.xlarge para el gateway y g4dn.xlarge (T4) para Triton.

ALTERNATIVAS

GKE o AKS; un endpoint administrado (SageMaker) si no hay equipo de plataforma; docker compose solo para desarrollo.

En el curso: M6 · Qué resuelve Kubernetes (y cuándo no lo necesitas) · M6 · Los objetos que sí debes conocer · Docker y kubectl · kubectl: mirar, entrar y conectar

NVIDIA device plugin / GPU Operator

GPUS EN KUBERNETES

El componente que le enseña a Kubernetes que un nodo tiene GPUs: las detecta y las anuncia como un recurso que los pods pueden pedir (nvidia.com/gpu).

COMO *El inventario del almacén: si la GPU no está registrada, para Kubernetes no existe, aunque esté físicamente en la máquina.*

EN EL C01 Prerrequisito del cluster. Cada nodo g4dn.xlarge anuncia nvidia.com/gpu: 1 y cada pod de Triton pide una T4 entera; la label workload=gpu-t4 y el taint nvidia.com/gpu=present:NoSchedule reservan esos nodos para Triton.

¿CÓMO SE TRABAJA?

Se configura · se opera

Plataforma lo **opera**: lo instala una vez como DaemonSet (con Helm) o mediante el GPU Operator, que también instala driver, toolkit y el exportador de métricas. Desde ahí funciona solo en cada nodo GPU. Tú solo **configuras** tu pod: `nvidia.com/gpu: 1` en `limits` (enteros y, por defecto, sin compartir), más `nodeSelector` y `tolerations` para caer en los nodos correctos.

EJEMPLO

YAML reference/c01/k8s/triton.yaml (extracto)

```yaml
          resources:
            requests:
              cpu: "2"  # Triton usa CPU para red, deserializar y el scheduler de batching
              memory: 6Gi  # librerías CUDA + buffers pinned; el modelo en sí vive en la GPU
              nvidia.com/gpu: 1  # recurso extendido del NVIDIA device plugin: una T4 entera por pod
            limits:
              memory: 6Gi  # memoria: límite = request (sin sorpresas de OOM por vecinos)
              nvidia.com/gpu: 1  # las GPUs no se comparten ni se sobre-asignan: request = limit
              # Sin límite de CPU: el nodo es solo para Triton y el throttling de CFS mete latencia
```

DÓNDE CORRE

Un pod por nodo GPU (DaemonSet). El driver lo trae la AMI acelerada de EKS o lo instala el GPU Operator.

ALTERNATIVAS

GPU Operator (driver, toolkit, device plugin, GPU Feature Discovery y DCGM exporter); AMIs que ya traen el plugin (Bottlerocket NVIDIA) o EKS Auto Mode.

En el curso: M6 · GPUs en Kubernetes · M5 · GPUs dentro del contenedor: NVIDIA Container Toolkit

HPA / KEDA

AUTOSCALING DE PODS

Controladores que cambian el número de réplicas de un Deployment según una métrica: el HPA viene con Kubernetes; KEDA lo extiende con métricas externas (Prometheus, colas) y escalado a cero.

COMO *Un lazo de control proporcional sobre el número de réplicas, con banda muerta (10 % de tolerancia) y una ventana para no oscilar.*

EN EL C01 Gateway: HPA por CPU al 60 % del request (3–12 pods). Triton: KEDA consulta a Prometheus cada 15 s el tiempo medio en cola, con objetivo de 10 ms (2–4 pods); KEDA crea y administra el HPA keda-hpa-triton.

¿CÓMO SE TRABAJA?

Se configura · se opera

Configuras en YAML un `HorizontalPodAutoscaler` (métrica, objetivo, mínimo y máximo) o un `ScaledObject` de KEDA (trigger, query y umbral); con el ScaledObject, KEDA crea por su cuenta el HPA `keda-hpa-triton` y le sirve la métrica. El controlador del HPA calcula cada 15 s `ceil(réplicas × métrica / objetivo)` y ajusta el Deployment: sube rápido y baja despacio. KEDA es un operador que plataforma instala una vez; tú lo **operas** con `kubectl get hpa` y `kubectl get scaledobject`.

EJEMPLO

YAML reference/c01/k8s/keda-triton.yaml (extracto)

```yaml
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: triton  # el Deployment de k8s/triton.yaml
  minReplicaCount: 2  # nunca menos de 2: una GPU por AZ (N+1), aunque 1 sola cubra el pico
  maxReplicaCount: 4  # techo de costo: 4 × g4dn.xlarge; si no alcanza, TritonQueueTimeHigh avisa
  pollingInterval: 15  # cada 15 s KEDA consulta Prometheus
```

DÓNDE CORRE

Dentro del cluster: el HPA en el control plane; KEDA como operador en su propio namespace. Los nodos los agrega otro lazo (Karpenter o Cluster Autoscaler).

ALTERNATIVAS

HPA con una métrica custom vía Prometheus Adapter (requests en espera); trigger cron de KEDA para escalar antes de un pico conocido.

En el curso: M6 · HPA: por qué escalar por CPU no sirve para un servicio de GPU · M6 · KEDA: escalar por eventos y hasta cero · Docker y kubectl · Autoscaling y capacidad

AWS Application Load Balancer (vía Ingress)

BALANCEO DE CARGA (L7)

El balanceador HTTP(S) administrado de AWS: recibe los requests de los callers y los reparte entre los pods sanos.

COMO *La recepción de un edificio: recibe a cada visitante, revisa su credencial y lo manda a una ventanilla abierta.*

EN EL C01 ALB interno (scheme internal) en HTTPS:443 que manda directo a la IP de cada pod del gateway (target-type ip), con health check en /readyz, drenado de 30 s e idle timeout de 60 s. También reparte por pesos entre la release estable y la canary.

¿CÓMO SE TRABAJA?

Administrado · se configura

No creas el ALB a mano: **configuras** un Ingress con `ingressClassName: alb` y anotaciones `alb.ingress.kubernetes.io/*`. El AWS Load Balancer Controller, que corre dentro del cluster, ve el Ingress y crea solo el ALB, sus target groups, listeners y reglas; AWS lo opera y lo escala como servicio **administrado**.

EJEMPLO

YAML reference/c01/k8s/gateway.yaml (anotaciones del Ingress, extracto)

```yaml
    alb.ingress.kubernetes.io/scheme: internal
    # ip: el ALB manda directo a la IP del pod (sin NodePort ni salto extra de kube-proxy)
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTPS": 443}]'
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-1:123456789012:certificate/00000000-0000-0000-0000-
000000000000
    # El health check del ALB usa la misma señal que la readiness de k8s
    alb.ingress.kubernetes.io/healthcheck-path: /readyz
    alb.ingress.kubernetes.io/success-codes: "200"
    # Drenado al sacar un pod: 30 s (el default de 300 s hace eternos los deploys)
    alb.ingress.kubernetes.io/target-group-attributes: deregistration_delay.timeout_seconds=30
```

DÓNDE CORRE

Servicio de AWS dentro de la VPC, en subredes de las dos zonas; el Load Balancer Controller corre como pod en el cluster.

ALTERNATIVAS

Amazon API Gateway (tope de 10 MB por payload y costo por request); Gateway API de Kubernetes.

En el curso: C01 · Arquitectura · M6 · Los objetos que sí debes conocer · M13 · Costos ocultos: recorre los bytes

NVIDIA Triton Inference Server

model server · ficha completa en la estación 5 · Export →

se configura | se opera

10

Estación 10 de 12 · tramo Operación · ● línea de días

Rollout seguro

¿Cómo cambias de modelo sin que nadie pague el error, y cómo vuelves atrás?

En el C01, desplegar una versión registrada es cambiar el tag de la imagen en los manifests, pero un modelo nuevo no entra con un simple rolling update: la release es un par **gateway vN + Triton vN** con su propio Service, que pasa por shadow (1–2 días), canary 5 % → 25 % → 100 % con rollback automático y, si el cambio es grande, un A/B por vendedor. Al promover, el alias champion se mueve en MLflow y la release anterior queda viva 24–48 h: volver es un cambio de ruteo de segundos, no un redeploy.

ENTRA

- challenger en MLflow (alias challenger)
- imagen de Triton vN en ECR

SALE

- versión nueva al 100 % (alias champion) — VIAJA A 11 · REQUEST
- release anterior viva 24–48 h para el rollback

EL PROCESO, PASO A PASO

PASO 1 — Release vN sin tráfico

Los manifests pasan al tag de la versión registrada (ya en ECR): Triton vN con su Service triton-vN y un gateway vN que solo habla con él, todavía sin tráfico.

automático (CD)

repo de manifests → EKS, con una T4 extra

PASO 2 — Shadow 1–2 días

El gateway copia de forma asíncrona una muestra al challenger y solo loguea: acuerdo con el champion, p99, errores y distribución.

automático (gateway)

gateway y Triton challenger en su propia T4

PASO 3 — Canary al 5 %

Pesos en el ALB (o Argo Rollouts): 5 % de los requests durante horas, incluido un pico diurno.

automático (CD)

ALB (target groups con peso)

PASO 4 — Análisis contra el estable

5xx > 0.5 %, p99 > 200 ms o 1.2× el estable, baja confianza > 1.5×, PSI > 0.1: el peso vuelve solo al estable.

automático (CD)

Argo Rollouts o el pipeline de CD, con queries a Prometheus en la misma ventana

PASO 5 — 25 % y, si aplica, A/B

Horas al 25 %; si el cambio es grande, A/B por vendedor de 1–2 semanas con el header X-Model-Arm y una regla del ALB.

MLE

ALB y backend de publicación

PASO 6 — Promover y dejar el bake

Una persona aprueba el 100 %, el alias champion se mueve en MLflow y la release anterior queda viva 24–48 h.

MLE · MLflow y EKS

ARCHIVOS DEL C01

ARCHIVO REAL **reference/c01/k8s/triton.yaml**

El tag de la imagen es la versión del modelo; un canary o un A/B es otra release con su propio Service (triton-vN), nunca dos modelos detrás del mismo Service.

YAML extracto

```yaml
        # Tag = versión del modelo (imagen horneada). Rollback de emergencia = volver al tag anterior. Un canary
        # o un A/B es otra release (Deployment + Service triton-<versión>): este Service nunca debe mezclar dos modelos
        image: 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-triton:2026-09-01-resnet50-fp16
```

ARCHIVO REAL **reference/c01/k8s/gateway.yaml**

El rolling update del gateway cuando cambia el código: pods nuevos antes de bajar los viejos y nunca menos capacidad.

YAML extracto

```yaml
  # Sin "replicas": las decide el HPA de abajo (3–12)
  revisionHistoryLimit: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%  # pods nuevos antes de bajar viejos
      maxUnavailable: 0  # capacidad completa durante todo el deploy
```

ARCHIVO REAL **reference/c01/triton/model_repository/classifier/config.pbtxt**

Una sola versión por imagen: el rollback es volver a desplegar la imagen anterior, no cargar dos modelos en la misma GPU.

PBTXT extracto

```
# Solo la versión más nueva del directorio. Con la imagen "horneada" hay una sola versión; el
# rollback es volver a desplegar la imagen anterior, no cargar dos versiones en la misma GPU.
version_policy: { latest: { num_versions: 1 } }
```

DESCRITO EN EL CURSO **rollout.yaml + analysis.yaml (M11 · canary)**

Ejemplo del curso para el gateway de C01 con Argo Rollouts: setWeight 5 → pausa de 4 h → 25 → pausa manual → 100, y un AnalysisTemplate que aborta si el canary pasa de 0.5 % de 5xx o si su baja confianza supera 1.5× la del estable. No está en reference/c01.

DESCRITO EN EL CURSO **Regla del listener del ALB para el A/B (C01 · evaluación)**

El backend asigna el brazo con un hash del seller_id y lo manda en X-Model-Arm; una regla con condición http-header lleva cada brazo al target group de su release. Descrita en el caso, sin archivo en el repo.

COMANDOS CLAVE

```
kubectl diff -f k8s/triton.yaml
```

Qué cambiaría antes de aplicar (exit 1 = hay diferencias).

```
kubectl rollout status deploy/triton -n classifier --timeout=25m
```

Espera el rollout: cada pod de Triton puede tardar minutos (nodo, imagen, engine).

```
kubectl argo rollouts promote gateway -n classifier
```

Con Argo Rollouts: sale de la pausa manual del canary.

```
kubectl argo rollouts abort gateway -n classifier
```

Devuelve el tráfico al estable; el Rollout queda `Degraded` hasta que el template vuelva a la versión anterior (con GitOps, un revert).

```
kubectl rollout undo deploy/triton -n classifier
```

Vuelve al ReplicaSet anterior: el tag anterior es el modelo anterior. Con GitOps, un revert en git.

```
kubectl rollout history deploy/triton -n classifier
```

Revisiones guardadas para volver: el C01 conserva 5.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
El deploy de Triton lleva más de 15 min con un pod Pending y rollout status falla con ProgressDeadlineExceeded.	Con maxSurge 1 y maxUnavailable 0, el pod nuevo necesita una T4 libre y el nodo no llegó (capacidad, cuota o límite del NodePool). Kubernetes no revierte solo.	`kubectl describe pod pod -n classifier` (Insufficient nvidia.com/gpu) y los eventos de Karpenter; los pods viejos siguen sirviendo.
Con 2 pods estables y 1 canary, el canary recibe cerca de un tercio del tráfico en vez del 5 %.	Sin trafficRouting, Argo Rollouts aproxima el peso con el número de réplicas.	Configura trafficRouting con el ALB para que el peso lo ponga el balanceador; kubectl argo rollouts get rollout gateway -n classifier.

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
En cada deploy del gateway aparecen 502 o 503 sueltos durante unos segundos.	El pod recibe SIGTERM mientras el ALB todavía le manda requests: falta el preStop, o el grace period no alcanza para preStop + drenado.	preStop sleep 15 (gateway.yaml) + --timeout-graceful-shutdown 20 (Dockerfile) + margen ≤ terminationGracePeriodSeconds 45, y drenado del target group de 30 s.
El canary pasó en verde y al día siguiente cae la tasa de aceptación.	El canary mide salud por request, no cómo reacciona quien publica; los cambios grandes necesitan un A/B.	¿Coincide con el deploy? Rollback por ruteo a la release anterior (sigue viva) y después un A/B por vendedor.
El log dice un model_version, pero las predicciones son de otro modelo.	Dos modelos detrás del mismo Service, o pods que leen @champion al arrancar y cargan versiones distintas.	`kubectl get deploy -n classifier -o wide` (columna IMAGES): cada release con su imagen y su Service triton-vN.

✓ **LISTA CUANDO…**

- [x] La release nueva (gateway vN + Triton vN con su Service) está lista antes de recibir tráfico.
- [x] Los criterios para revertir (5xx, p99, baja confianza, PSI) y para avanzar están escritos antes de empezar.
- [x] Cada etapa del canary juntó al menos 10,000 requests y cubrió un pico diurno.
- [x] Una persona aprobó el 100 %, el alias champion apunta a la versión nueva y la anterior sigue viva 24–48 h.
- [x] El rollback está ensayado: peso al estable en segundos; redeploy del tag anterior en ≈ 5–20 min.

CÓMO RECORDARLA

gate → shadow → canary → (A/B) → promoción. Cada peldaño expone más y responde otra pregunta: ¿es mejor?, ¿funciona?, ¿es seguro?, ¿ayuda al producto? Revertir es automático; promover lo aprueba una persona.

TRAMPA

creer que un rolling update es un despliegue seguro de un modelo. La readiness solo pregunta si el modelo cargó: un modelo que se equivoca con confianza alta pasa todas las probes.

En el curso: M11 · Shadow deployment: comparar sin afectar a nadie · M11 · Canary: porcentajes, tiempos y criterios para avanzar · M11 · Blue/green y rollback instantáneo · M11 · Rollback: criterios automáticos y qué guardar para volver · C01 · Evaluación offline y online

HERRAMIENTAS DE ESTA ESTACIÓN

Argo Rollouts (o pesos del ALB)

DESPLIEGUE PROGRESIVO (CANARY)

Un controlador de Kubernetes que reemplaza al Deployment por un Rollout: mueve el tráfico a la versión nueva por pasos, consulta métricas mientras avanza y vuelve solo al estable si algo se rompe.

COMO *Abrir una válvula de a poco mirando el manómetro: si la presión sale de rango, se cierra sola.*

EN EL C01 El curso da dos caminos para el canary 5 % → 25 % → 100 %: target groups con peso en el ALB o Argo Rollouts con trafficRouting por el ALB. Solo el gateway es un Rollout: su canary habla con triton-vN, desplegado antes y sin tráfico, y un AnalysisTemplate aborta si el canary pasa de 0.5 % de 5xx o si su baja confianza supera 1.5× la del estable.

¿CÓMO SE TRABAJA?

Se configura · se opera

Configuras en YAML un `Rollout` con sus pasos (`setWeight`, `pause`) y un `AnalysisTemplate` con queries de Prometheus y su condición de éxito. El controlador avanza solo, cambia los pesos del ALB a través de las anotaciones del Ingress y aborta si una medición falla. Lo **operas** con el plugin de kubectl: `kubectl argo rollouts promote` después de una pausa manual o `abort` para volver.

EJEMPLO

YAML rollout.yaml de M11 · canary (extracto, sin los Services ni el bloque analysis); no está en reference/c01

```yaml
  strategy:
    canary:
      trafficRouting:
        alb: {ingress: classifier, servicePort: 8080}   # el Ingress apunta a gateway-stable; pesos exactos en el ALB
      steps:
        - setWeight: 5
        - pause: {duration: 4h}          # horas, no minutos: tiene que cubrir tráfico real
        - setWeight: 25
        - pause: {}                      # sin duración: promoción manual después de un pico diurno
        - setWeight: 100
```

DÓNDE CORRE

El controlador corre dentro del cluster; el plugin, en tu terminal.

ALTERNATIVAS

Flagger (con una malla o un ingress soportado); pesos del ALB cambiados por tu pipeline de CD; guardrails de despliegue de SageMaker.

En el curso: M11 · Canary: porcentajes, tiempos y criterios para avanzar · M11 · Rollback: criterios automáticos y qué guardar para volver · C01 · Evaluación offline y online

AWS Application Load Balancer (vía Ingress)

balanceo de carga (L7) · ficha completa en la estación 9 · Kubernetes →

administrado · se configura

Prometheus + Alertmanager

métricas y alertas · ficha completa en la estación 12 · Monitoreo →

se configura · se programa

MLflow (tracking + model registry)

tracking de experimentos y registry de modelos · ficha completa en la estación 3 · Entrenamiento →

se programa · se opera

Estación 11 de 12 · tramo Operación · línea de milisegundos

La request en vivo

¿Qué le pasa a una foto desde que llega al ALB hasta que vuelve el top-5, y cuánto cuesta cada salto?

La foto entra por el ALB interno; el gateway la valida, la decodifica y la convierte en un tensor de 602 KB en CPU; Triton la junta con otras (≤ 2 ms) y la T4 la pasa por el engine TensorRT FP16. El SLO es **p99 ≤ 200 ms server-side**: el presupuesto típico da ≈ 20–45 ms en p50 y ≈ 150 ms en un p99 pesimista. Los timeouts bajan en cascada (`300 → 250 → 150 → 100 ms`) y, si algo falla, el caller muestra el selector manual.

ENTRA

foto (≈ 200 KB) desde el backend de publicación

SALE

una línea de log JSON y métricas por request
VIAJA A 12 · MONITOREO

top-5 con model_version y X-Request-ID para el caller

EL PROCESO, PASO A PASO

PASO 1
Backend → ALB

p50 1–3 ms · p99 5–10 ms: el backend manda los bytes con su JWT (timeout de 300 ms); el ALB termina TLS, aplica el peso estable/canary y elige un pod listo.

automático (ALB)

ALB interno de AWS, en dos zonas

PASO 2
Admisión y body

p50 1–3 ms · p99 10–30 ms: 16 requests ya en vuelo (503 + Retry-After), Content-Type (415), más de 10 MB (413) y lectura del body.

automático (gateway)

pod del gateway (nodo de CPU)

PASO 3
Decode y preproceso

p50 6–13 ms · p99 27–52 ms: threadpool con 2 slots; EXIF → RGB → 256 → crop 224 → normalizar → tensor FP32 de 602 KB.

automático (gateway)

pod del gateway (CPU)

PASO 4
Salto a Triton

p50 2–4 ms · p99 5–10 ms: KServe v2 por HTTP con tensores binarios al Triton de la misma zona, con timeout de min(150 ms, lo que queda).

automático (gateway)

red de la VPC (Service triton)

PASO 5
Cola de batching

p50 0–2 ms · p99 2 ms: Triton espera hasta 2 ms para juntar requests; si uno lleva 100 ms en cola, lo rechaza.

automático (Triton)

pod de Triton

PASO 6
Cómputo en la T4

p50 3–6 ms · p99 8–15 ms: engine TensorRT FP16 con batch de 1–8 y dos instancias por GPU; devuelve 500 logits FP32.

automático (Triton)

GPU T4 (g4dn.xlarge)

PASO 7
Respuesta y log

p50 6–11 ms · p99 12–33 ms con el overhead de Python: softmax, top-5, JSON con model_version y una línea de log a stdout.

automático (gateway)

pod del gateway → backend de publicación

ARCHIVOS DEL C01

ARCHIVO REAL **`reference/c01/k8s/gateway.yaml`**

Qué deja pasar el ALB: solo /v1/classify hacia el Service gateway; /metrics, /healthz y /readyz no salen del cluster.

YAML extracto

```yaml
spec:
  ingressClassName: alb  # lo atiende el AWS Load Balancer Controller
  rules:
    - http:
        paths:
          # Solo la API: /metrics, /healthz y /readyz NO quedan expuestos fuera del cluster
          - path: /v1/classify
            pathType: Exact
            backend:
              service:
                name: gateway
```

✓ ARCHIVO REAL **reference/c01/gateway/app/main.py**

Deadline propagation: la llamada a Triton usa lo que queda del presupuesto de 250 ms, tope 150 ms, y se mide aparte del preproceso.

PYTHON extracto

```python
    left = _budget_left(deadline)  # no mandes a la GPU un request que el caller ya abandonó
    t = time.perf_counter()
    logits = await _infer(  # I/O: aquí el await sí cede el loop
        request.app.state.triton, batch, record["request_id"], timeout=min(INFER_TIMEOUT_S, left)
    )
    inf_s = time.perf_counter() - t
    INFERENCE.observe(inf_s)
    record["inference_ms"] = round(inf_s * 1000, 1)
```

✓ ARCHIVO REAL **reference/c01/k8s/triton.yaml**

El salto pasa por el Service de Triton: prefiere el pod de la misma zona y balancea por conexión (L4), por eso HTTP con un pool y no gRPC.

YAML extracto

```yaml
  trafficDistribution: PreferSameZone
  # kube-proxy balancea por CONEXIÓN (L4). Con HTTP/1.1 y un pool de conexiones por gateway el
  # reparto es razonable. Con gRPC (HTTP/2, una conexión multiplexada) todo el tráfico de un
  # gateway iría a UN solo pod: necesitarías balanceo L7 (service mesh) o client-side.
```

✓ ARCHIVO REAL **reference/c01/triton/model_repository/classifier/config.pbtxt**

El último tramo: dos copias del engine en la misma T4, para que una calcule mientras la otra mueve datos. La cola de batching está en la estación 6.

PBTXT extracto

```
# 2 instancias del engine en la MISMA GPU: mientras una copia datos o espera, la otra computa.
# Cada instancia reserva su propia memoria de activaciones; con ResNet-50 FP16 sobra en 16 GB.
instance_group [
  {
    count: 2
    kind: KIND_GPU
  }
]
```

COMANDOS CLAVE

```
curl -s -H "Content-Type: image/jpeg" --data-binary @foto "localhost:8080/v1/classify?top_k=5"
```

En local (compose): la respuesta trae `model_version` y `latency_ms`, y el header `X-Request-ID`.

```
curl -i localhost:8080/readyz
```

200 solo si el modelo está listo en Triton; en EKS, con 503 el pod sale del ALB.

```
kubectl logs deploy/gateway -n classifier --tail=5
```

Una línea JSON por request: `request_id`, `preprocess_ms`, `inference_ms`, `status`.

```
kubectl port-forward -n classifier svc/triton 8000:8000 8002:8002
```

Túnel a un pod de Triton sin exponerlo: sirve para mirar, no para load tests.

```
curl -s localhost:8002/metrics | grep nv_inference_queue_duration_us
```

El tiempo acumulado en la cola de Triton: la señal con la que escala KEDA.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
504 timeout de inferencia a los ≈ 150 ms y la alerta TritonQueueTimeHigh.	La GPU está saturada: la cola de Triton crece más rápido de lo que KEDA agrega réplicas, y cada una tarda minutos.	Espera media en cola (nv_inference_queue_duration_us ÷ nv_inference_request_success) y réplicas de KEDA contra su máximo de 4.

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
Ráfagas de 503 gateway saturado con Retry-After: 1.	Load shedding: el pod ya tiene 16 requests en vuelo o no consiguió un slot de CPU en 50 ms.	CPU de los pods y kubectl get hpa -n classifier: ¿el HPA está escalando o topó en 12?
El p99 crece con la concurrencia aunque la CPU no está llena.	Trabajo de CPU dentro de un async def que bloquea el event loop, o CPU throttling por un limits.cpu.	classifier_preprocess_seconds contra classifier_inference_seconds, y la fracción de periodos throttled del contenedor.
502 esporádicos en el ALB.	El backend cierra la conexión antes que el ALB: keep-alive de uvicorn menor que el idle timeout.	--timeout-keep-alive 75 (Dockerfile) tiene que ser mayor que idle_timeout 60 (anotación del Ingress).
Los callers reciben 503 del ALB y ClassifierHigh5xxRate no dispara.	Triton cayó: todos los gateways quedan NotReady y el ALB responde sin llegar al gateway.	La alerta TritonNotReady y HTTPCode_ELB_5XX_Count del ALB en CloudWatch.

✓ **LISTA CUANDO…**

- [x] Cada respuesta trae model_version y X-Request-ID, y cada línea de log trae preprocess_ms e inference_ms.
- [x] El p99 del histograma del gateway, sumado entre pods, queda bajo 200 ms en el pico de diseño (500 req/s).
- [x] Los timeouts están encadenados: 300 ms el caller, 250 ms el deadline, 150 ms a Triton y 100 ms en la cola.
- [x] Bajo sobrecarga ves 503 y 504 rápidos (y el selector manual), no requests colgados.
- [x] Cada gateway llama al Triton de su propia zona.

CÓMO RECORDARLA

lo barato primero, la GPU al final. Rechazar cuesta microsegundos, el decode va en CPU y a la T4 solo llega lo que sigue vivo; cada salto hereda lo que queda del presupuesto (`300 → 250 → 150 → 100 ms`).

TRAMPA

creer que el modelo domina la latencia. El cómputo en la T4 son 8–15 ms de un p99 pesimista de ≈ 150 ms; casi todo se va en mover y preparar bytes en CPU, así que comprar GPU no arregla un p99 de preproceso.

En el curso: C01 · Arquitectura · C01 · Estimaciones back-of-the-envelope · M3 · Presupuesto de latencia end-to-end · C01 · Serving · M7 · Protocolos: REST, gRPC, KServe v2 y OpenAI-compatible

HERRAMIENTAS DE ESTA ESTACIÓN

OpenTelemetry

TRAZAS DISTRIBUIDAS

Un estándar abierto con SDKs para generar trazas, métricas y logs, y un Collector que los recibe y los manda al backend que elijas; no guarda ni muestra datos por sí mismo.

COMO *Un número de guía pegado al paquete que se registra en cada estación: después reconstruyes dónde se demoró.*

EN EL C01 El curso lo describe para seguir un request del gateway a Triton (Triton exporta trazas OpenTelemetry), muestreado ≈ 1 %, con el mismo request_id en logs y trazas y exemplars en el histograma de latencia para saltar del p99 a un request concreto. El código de referencia todavía no lo implementa.

¿CÓMO SE TRABAJA?

{ } Se programa · se configura

Lo **programas**: agregas el SDK al gateway (o la instrumentación automática de FastAPI, que crea un span por request sola) y propagas el contexto en los headers de la llamada a Triton. Luego **configuras** el muestreo con variables estándar (`OTEL_TRACES_SAMPLER` , `OTEL_TRACES_SAMPLER_ARG`), el Collector que recibe y reenvía los spans y, en Triton, `--trace-config mode=opentelemetry` . El backend (CloudWatch/X-Ray u otro) guarda y muestra las trazas.

EJEMPLO

BASH ilustrativo: variables de la especificación de OpenTelemetry y flags de la guía de trazas de Triton; el C01 no lo trae

```
# Gateway: 1 % de las trazas nuevas (variables estándar del SDK)
export OTEL_SERVICE_NAME=classifier-gateway
export OTEL_TRACES_SAMPLER=parentbased_traceidratio
export OTEL_TRACES_SAMPLER_ARG=0.01
# Triton ≥ 24.01: con rate=0 traza solo los requests que traen el contexto del gateway
tritonserver --model-repository=/models --trace-config mode=opentelemetry \
  --trace-config level=TIMESTAMPS --trace-config rate=0 \
  --trace-config opentelemetry,url=http://otel-collector:4318/v1/traces
```

DÓNDE CORRE

El SDK dentro del proceso del gateway; el Collector como pod en el cluster; Triton exporta por OTLP/HTTP; el backend de trazas, aparte.

ALTERNATIVAS

CloudWatch/X-Ray, Cloud Trace o Application Insights reciben datos de OpenTelemetry; Datadog o New Relic si la empresa ya paga uno.

En el curso: C01 · Monitoreo, alertas y reentrenamiento · Herramientas por categoría · Monitoreo: sistema y modelo

AWS Application Load Balancer (vía Ingress)

balanceo de carga (L7) · ficha completa en la estación 9 · Kubernetes →

administrado · se configura

FastAPI + Uvicorn

framework web (API en Python) · ficha completa en la estación 6 · Serving →

se programa · se configura

NVIDIA Triton Inference Server

model server · ficha completa en la estación 5 · Export →

se configura · se opera

12

Estación 12 de 12 · tramo Operación · ● línea de continuo

Monitoreo y reentrenamiento

¿Cómo te enteras de que algo falló, en minutos o en días, y cómo vuelve el sistema a la estación 1?

Un modelo falla de dos maneras: **ruidosamente** (errores, latencia, pods caídos: Prometheus lo ve en minutos) y **en silencio** (responde 200 OK y rápido, pero mal: solo lo ven la confianza, el PSI y la aceptación, a veces días después). Las alertas de síntomas hacen page; las de calidad abren ticket. Los logs de predicción alimentan el etiquetado y un disparador adelanta el reentrenamiento: así se cierra el ciclo hacia la estación 1.

ENTRA

- métricas en /metrics (gateway :8080, Triton :8002)
- logs de predicción y eventos del caller

→

SALE

- disparador de reentrenamiento y muestras para etiquetar
  VUELVE A 1 · DATOS (REENTRENAMIENTO)
- alertas (page, warning, ticket) y dashboards

EL PROCESO, PASO A PASO

PASO 1

Golden signals por tier

Latencia, tráfico, errores y saturación: el ALB en CloudWatch; el gateway (:8080) y Triton (:8002) en Prometheus vía ServiceMonitor, cada 15 s.

automático (Prometheus)

kube-prometheus-stack en EKS y CloudWatch

→

PASO 2

Métricas de modelo y datos

Sin labels y en minutos: histograma del score top-1, tasa de baja confianza por model_version y bytes de cada imagen.

automático (gateway)

/metrics del gateway

→

PASO 3

Reglas y alertas

1 regla de grabación y 6 alertas como PrometheusRule; Alertmanager manda page (p99, 5xx, Triton), warning (cola, GPU) o ticket (confianza).

automático (Prometheus)

Prometheus y Alertmanager

→

PASO 4

Dashboards

Servicio por tier y caller, Triton y GPU, modelo partido por model_version, y negocio diario.

MLE

Grafana

→

PASO 5

Logs y trazas

Una línea JSON por request (≈ 300 B, sin PII) → Fluent Bit → S3 en Parquet; trazas OpenTelemetry al ≈ 1 % con el mismo request_id.

automático

stdout del pod → DaemonSet de Fluent Bit → S3

→

PASO 6

Job diario de drift

PSI de categorías predichas contra las 4 semanas previas, tasa de aceptación (eventos del caller) y re-categorización; publica métricas y abre tickets.

automático (job batch)

job batch sobre los logs en S3

→

PASO 7

Reentrenar y cerrar el ciclo

Mensual, o antes por PSI > 0.2 tres días, −3 pp de aceptación o categorías nuevas: ticket → pipeline → gate → shadow → canary → champion; las muestras vuelven a etiquetado.

automático (pipeline)

orquestador del pipeline y cola de etiquetado (estación 1)

ARCHIVOS DEL C01

ARCHIVO REAL **reference/c01/monitoring/alerts.yml**

La alerta de calidad sin labels: si se duplica la fracción de predicciones con score top-1 menor a 0.3 contra su promedio de 7 días durante 30 min, abre un ticket.

YAML extracto

```yaml
      - alert: ClassifierLowConfidenceSpike
        expr: |
          classifier:low_confidence_share:rate30m
            > 2 * avg_over_time(classifier:low_confidence_share:rate30m[7d])
          and sum(rate(classifier_top1_score_count[30m])) > 1
        # Necesita 7 días de historia de la regla de grabación (retención de Prometheus ≥ 7 d)
        for: 30m
        labels:
          severity: ticket
```

ARCHIVO REAL **reference/c01/k8s/monitoring.yaml**

Cómo encuentra Prometheus al gateway: por las labels del Service y la label release que selecciona tu Prometheus. Sin esto, KEDA no ve la cola de Triton y las alertas se quedan sin datos.

YAML extracto

```yaml
kind: ServiceMonitor
metadata:
  name: gateway
  namespace: classifier
  labels:
    release: kube-prometheus-stack  # = serviceMonitorSelector de tu Prometheus (nombre del release de Helm)
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: gateway  # labels del Service de k8s/gateway.yaml
  endpoints:
    - port: http  # nombre del puerto del Service: 8080/metrics del gateway (no expuesto en el Ingress)
```

ARCHIVO REAL **reference/c01/gateway/app/main.py**

La métrica de modelo más barata: el histograma del score top-1 por model_version, que separa el canary del estable.

PYTHON extracto

```python
TOP1 = Histogram(  # distribución de confianza: la señal de drift más barata que existe (sin labels)
    "classifier_top1_score", "Score top-1 tras softmax", ["model_version"],  # separa canary de estable
    buckets=tuple(round(0.1 * i, 1) for i in range(1, 11)),  # 0.1 … 1.0 → el bucket le="0.3" alimenta una alerta
)
```

DESCRITO EN EL CURSO **Fluent Bit → S3 (logs de predicción)**

main.py escribe una línea JSON por request a stdout; un DaemonSet de Fluent Bit la manda a S3 en lotes y se convierte a Parquet. El repo no trae esa configuración.

DESCRITO EN EL CURSO **Job diario de drift y aceptación**

Sobre los logs en S3: PSI contra las 4 semanas previas (ticket si pasa de 0.2 tres días), aceptación (ticket si cae más de 3 pp semana contra semana) y re-categorización. alerts.yml lo menciona; el repo no lo trae.

COMANDOS CLAVE

```
kubectl apply -f k8s/monitoring.yaml
```

ServiceMonitors del gateway y de Triton; las reglas de `alerts.yml` se cargan aparte como PrometheusRule (comando en el README).

```
histogram_quantile(0.99, sum by (le) (rate(classifier_request_latency_seconds_bucket[5m])))
```

PromQL del p99: primero sumar buckets entre pods, después el cuantil.

```
sum(rate(classifier_requests_total{status=~"5.."}[5m])) / sum(rate(classifier_requests_total{status!~"4.."}[5m]))
```

Fracción de 5xx sobre requests válidos: los 4xx no cuentan.

```
sum(rate(nv_inference_count{model="classifier"}[5m])) / sum(rate(nv_inference_exec_count{model="classifier"}[5m]))
```

Batch promedio real de Triton: mídelo, no lo supongas.

```
kubectl describe scaledobject triton -n classifier
```

Si Prometheus no ve a Triton, aquí aparece el error del scaler, no un 0 silencioso.

QUÉ FALLA AQUÍ

LO QUE VES	POR QUÉ PASA	PRIMER CHEQUEO
El dashboard de servicio está en verde y la tasa de aceptación cae.	Falla silenciosa: cambió el input (una versión nueva de la app comprime distinto) o labels.json quedó desalineado; todo responde 200 y rápido.	Tasa de baja confianza por model_version y classifier_image_bytes contra la semana anterior; cruzar con los deploys.
Triton no pasa de 2 réplicas y las alertas que leen métricas del gateway o de Triton nunca disparan.	No se aplicó k8s/monitoring.yaml, o la label release de los ServiceMonitors no coincide con el serviceMonitorSelector de tu Prometheus.	`kubectl describe scaledobject triton -n classifier` y la lista de targets en la UI de Prometheus.
ClassifierLowConfidenceSpike no dispara durante días, aunque la confianza cambió.	La regla de grabación guardó NaN (0/0 en un rato sin tráfico, si alguien quitó el > 0 del denominador) y el promedio de 7 días lo arrastra una semana.	Busca NaN en classifier:low_confidence_share:rate30m y confirma el > 0 del denominador en alerts.yml; la retención debe cubrir 7 días.
El PSI del dashboard lleva días sin moverse.	El job diario de drift dejó de correr y se sigue viendo su último valor publicado.	Una alerta de frescura sobre el último push del job (más de 26 h para un job diario).
Prometheus se queda sin memoria.	Alguien agregó un label con valores ilimitados (request_id, user_id): cada valor es una serie nueva.	Labels acotados (status, source, model_version); los ids van a los logs y a los exemplars.

✓ LISTA CUANDO…

- [x] Las 6 alertas evalúan con su severidad (page, warning o ticket) y cada una tiene runbook.
- [x] Hay dashboards de servicio, de Triton y GPU, de modelo (partido por model_version) y de negocio.
- [x] El 100 % de las predicciones llega a S3 sin PII y el job diario publica el PSI y la aceptación.
- [x] Un disparador (mensual, PSI mayor a 0.2 por 3 días, −3 pp de aceptación o categoría nueva) abre ticket y adelanta el pipeline; nada se promueve sin gate y canary.
- [x] La baja confianza, los desacuerdos y una muestra aleatoria llegan a la cola de etiquetado (estación 1).

CÓMO RECORDARLA

ruidoso en minutos, silencioso en días. Las métricas de sistema atrapan errores y latencia; las de modelo (confianza, PSI, aceptación) atrapan al modelo que responde 200 OK y se equivoca. Page solo por síntomas que alguien ya está sufriendo.

TRAMPA

dejar que el monitoreo despliegue. Un disparador abre un ticket y adelanta el pipeline; el candidato pasa por el mismo gate, shadow y canary que cualquier otro, y nadie se despierta a las 3 a. m. porque subió el PSI.

En el curso: C01 · Monitoreo, alertas y reentrenamiento · M12 · Métricas de sistema: golden signals con detalles de ML · M12 · Métricas de modelo: con y sin labels · M12 · Prometheus, Grafana y alertas por SLO · M12 · Cuándo reentrenar (y cuándo no) · M10 · Reentrenamiento: programado, disparado o continuo

HERRAMIENTAS DE ESTA ESTACIÓN

Prometheus + Alertmanager

MÉTRICAS Y ALERTAS

Prometheus lee cada pocos segundos las métricas que exponen tus servicios, las guarda como series de tiempo y evalúa reglas de alerta; Alertmanager decide a quién avisar.

COMO *Un SCADA: lee los sensores (/metrics) cada 15 s, guarda las series y dispara alarmas con reglas.*

EN EL C01 kube-prometheus-stack scrapea el gateway (:8080/metrics) y Triton (:8002) cada 15 s vía ServiceMonitors y evalúa 1 regla de grabación y 6 alertas: p99 > 200 ms, 5xx > 1 %, menos de 2 Triton listos, baja confianza, cola > 20 ms y GPU > 85 %. KEDA le consulta la cola para escalar.

¿CÓMO SE TRABAJA?

Se configura · se programa

Las métricas del gateway las **programas** (`Counter` e `Histogram` de `prometheus_client` en `main.py`); Triton expone las suyas solo. Después **configuras** qué scrapear (ServiceMonitor), las reglas en YAML con PromQL (`expr` , `for` , `labels.severity`) y las rutas de Alertmanager. Prometheus hace solo el scrape y la evaluación; Alertmanager agrupa, deduplica, silencia y enruta: page al on-call, ticket al equipo de ML.

EJEMPLO

YAML reference/c01/monitoring/alerts.yml (extracto)

```yaml
      - alert: ClassifierHighP99Latency
        expr: |
          histogram_quantile(0.99,
            sum by (le) (rate(classifier_request_latency_seconds_bucket[5m]))
          ) > 0.2
        for: 10m  # 10 min sostenido: ignora picos de un GC o de un deploy
        labels:
          severity: page
```

DÓNDE CORRE

En el cluster (kube-prometheus-stack, namespace monitoring); en local, el contenedor prom/prometheus:v3.13.3 de compose.

ALTERNATIVAS

Amazon Managed Service for Prometheus; Datadog o New Relic si la empresa ya paga uno.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Métricas de sistema: golden signals con detalles de ML · C01 · Monitoreo, alertas y reentrenamiento

Grafana

DASHBOARDS

La herramienta para ver las métricas: dashboards con paneles que consultan a Prometheus con PromQL. No guarda las métricas: solo las lee y las dibuja.

COMO *El tablero de instrumentos: muestra lo que miden los sensores, no mide nada por sí mismo.*

EN EL C01 El curso describe cuatro dashboards: servicio (tasa, errores y latencia por tier y caller, con marcadores de deploy), Triton y GPU, modelo (score top-1, baja confianza y top-20 categorías, partido por model_version) y negocio diario. El repo de referencia no trae dashboards.

¿CÓMO SE TRABAJA?

Se opera · se configura · administrado

Lo **operas** en una UI: armas cada panel escribiendo su query PromQL, y Grafana consulta a Prometheus y redibuja los paneles solo. Para que no se pierdan, los dashboards se exportan como JSON y se **configuran** por provisioning desde git. También existe como servicio **administrado** (Amazon Managed Grafana, Grafana Cloud).

EJEMPLO

TEXTO consultas PromQL del curso (M12) sobre métricas reales del C01: dos paneles

```
# p99 del servicio en ventanas de 5 min: primero sumar buckets entre pods, después el cuantil
histogram_quantile(0.99,
  sum by (le) (rate(classifier_request_latency_seconds_bucket[5m])))
# Batch promedio en Triton: inferencias por ejecución (cerca de max_batch_size = sin margen)
sum(rate(nv_inference_count{model="classifier"}[5m]))
  / sum(rate(nv_inference_exec_count{model="classifier"}[5m]))
```

DÓNDE CORRE

En el cluster junto a Prometheus, o como servicio administrado.

ALTERNATIVAS

Amazon Managed Grafana, Grafana Cloud o el producto de observabilidad que ya use la empresa.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Dashboards, runbooks y on-call

OpenTelemetry

Amazon S3

trazas distribuidas · ficha completa en la estación 11 · Request →

se programa · se configura

almacenamiento de objetos · ficha completa en la estación 1 · Datos →

administrado · se configura · se opera

¿Se programa, se configura o se opera?

Las 23 herramientas del recorrido según cómo trabajas con ellas. Casi ninguna es de un solo tipo: el ícono chico dice qué más haces con cada una.

Se programa 7

Escribes código (Python, bash) con la lógica, y la herramienta lo ejecuta tal cual: si no lo escribes, no pasa. Ejemplo del C01: gateway/app/main.py (FastAPI) valida la foto, la decodifica y llama a Triton; model/parity_check.py decide si el engine pasa.

- **Orquestador de pipelines (Argo Workflows / Airflow / Kubeflow Pipelines)**
  3 · Entrenamiento
  configura · opera
- **PyTorch**
  3 · Entrenamiento
- **MLflow (tracking + model registry)**
  3 · Entrenamiento
  opera
- **ONNX**
  5 · Export
- **FastAPI + Uvicorn**
  6 · Serving
  configura
- **pytest**
  6 · Serving
  opera
- **OpenTelemetry**
  11 · Request
  configura

Se configura 8

Escribes un archivo declarativo (YAML, pbtxt) que describe qué quieres, no cómo lograrlo, y la herramienta llega a ese estado y lo mantiene. Kubernetes es esto: no lo programas, le das k8s/triton.yaml (una GPU por pod, probes, repartido en dos zonas) y keda-triton.yaml (de 2 a 4 réplicas según la cola), y el cluster crea, reinicia y reparte los pods solo.

- **NVIDIA Triton Inference Server**
  5 · Export
  opera
- **Docker (+ docker compose)**
  7 · Empaquetado
  opera
- **GitHub Actions**
  8 · CI/CD
  administrado
- **Kubernetes (Amazon EKS) + kubectl**
  9 · Kubernetes
  opera · administrado
- **NVIDIA device plugin / GPU Operator**
  9 · Kubernetes
  opera
- **HPA / KEDA**
  9 · Kubernetes
  opera
- **Argo Rollouts (o pesos del ALB)**
  10 · Rollout
  opera
- **Prometheus + Alertmanager**
  12 · Monitoreo
  programa

Se opera 5

Le das órdenes o miras su estado en el momento, con comandos o una UI, sin cambiar lo que está escrito en git. Ejemplo del C01: kubectl describe pod para ver por qué un Triton está en Pending, kubectl rollout undo para volver atrás o Grafana durante un incidente.

- **Label Studio**
  1 · Datos
  configura
- **DVC / lakeFS**
  2 · Versionado
  configura
- **NVIDIA TensorRT**
  5 · Export
- **Git / GitHub**
  8 · CI/CD
  administrado
- **Grafana**
  12 · Monitoreo
  configura · administrado

Administrado 3

Un servicio que rentas en la nube: el proveedor lo instala, lo actualiza y lo mantiene vivo; tú lo configuras y pagas por uso. Ejemplo del C01: S3 y ECR guardan datos e imágenes sin que operes discos, y en EKS AWS opera el control plane, pero los manifests siguen siendo tuyos (casi ninguna herramienta es de un solo tipo).

- **Amazon S3**
  1 · Datos
  configura · opera
- **Amazon ECR**
  8 · CI/CD
  configura · opera
- **AWS Application Load Balancer (vía Ingress)**
  9 · Kubernetes
  configura

4 patrones frente al C01

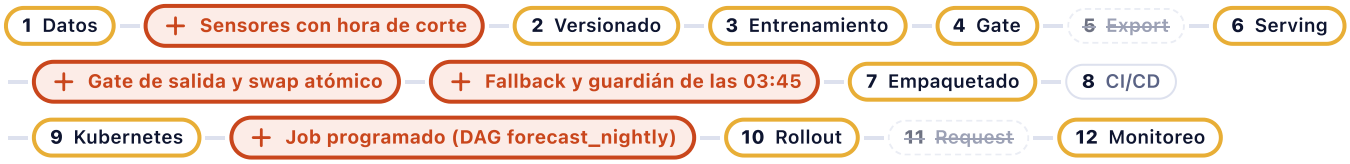
Qué estaciones desaparecen, cuáles cambian y cuáles aparecen cuando el sistema no es una API síncrona.

CÓMO LEER LAS TIRAS — igual que en el C01 · cambia · ~~desaparece~~ · + estación nueva

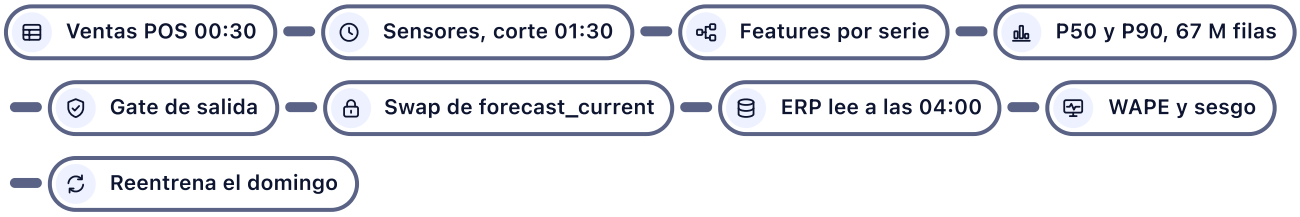
BATCH Pronóstico de demanda (C14)

Las ventas, el calendario y las promociones existen antes y nadie espera en línea: un job programado pronostica cada noche las 4.8 M series y a las 04:00 el ERP lee una tabla ya publicada. Frente al C01 desaparecen el export, la request y el autoscaling, y aparece lo que protege una ventana fija: sensores con hora de corte, un **gate de salida**, la **publicación atómica** y un fallback que nunca deja al ERP sin pronóstico.

FRENTE AL C01 · 9 CAMBIAN · 2 DESAPARECEN · 4 NUEVAS · 1 IGUAL



SU PROPIA LÍNEA



VELOCIDADES No hay línea de milisegundos: nadie llama en línea. Aparece una línea diaria con hora límite: la corrida nocturna, de 00:30 a 03:30, antes de que el ERP lea a las 04:00. El modelo se reentrena cada domingo (≈ 2–3 h en spot) y el código sale en días. La operación mira dos relojes: la noche (minutos, page) y la precisión (días, cuando llegan las ventas; ticket).

1 Datos	CAMBIA	Las ventas del POS (`sales_daily`) llegan cada noche a las 00:30 y son el label: no hace falta etiquetado humano. Los días agotados (`oos_flag`) salen de los targets porque su venta no es la demanda.
+ Sensores con hora de corte	NUEVA	Esperan las ventas y el inventario de cada tienda hasta que reportan todas o hasta la 01:30. Las que faltan se imputan con su mismo día de la semana pasada y se marcan `imputed_input` .
2 Versionado	CAMBIA	Cada entrenamiento fija un snapshot (time travel de Iceberg o Delta, o una copia con fecha), y cada noche publicada queda como partición inmutable con nombre de hash: el archivo de lo que el ERP usó.
3 Entrenamiento	CAMBIA	Cada domingo, dos GBDT por cuantiles (P50 y P90) sobre ≈ 490 M filas point-in-time, en un nodo de 96 vCPU en spot (≈ 2–3 h). Sin GPU.
4 Gate	CAMBIA	Backtest de origen móvil con 8 orígenes semanales: mejor WAPE que el champion y ningún segmento principal peor por más de 2 puntos.
~~5 Export~~	DESAPARECE	El GBDT predice en CPU con la misma imagen que lo entrena: no hay ONNX ni TensorRT.
6 Serving	CAMBIA	No hay endpoint: el serving es el job `run_forecast.py` , que imputa, arma features, predice 67 M filas y escribe una partición nueva.
+ Gate de salida y swap atómico	NUEVA	Write–audit–publish: el gate relee lo escrito (filas = series × 14, p90 ≥ p50, bandas por categoría) y publicar es un compare-and-swap del puntero `forecast_current` . El ERP nunca ve una tabla a medias.
+ Fallback y guardián de las 03:45	NUEVA	Si el job falla o el gate bloquea, se publica el pronóstico de ayer desplazado un día; un DAG aparte lo publica a las 03:45 si nada salió.
7 Empaquetado	CAMBIA	Una sola imagen (`forecast-job` , fijada por digest) entrena y predice, porque un modelo guardado con pickle de scikit-learn no es portable entre versiones.

8 CI/CD	IGUAL	Pruebas e imagen como en el C01; además se despliega el DAG `forecast_nightly` a Airflow.
9 Kubernetes	CAMBIA	Sin Deployment, HPA ni ALB: Airflow lanza el job como pod (KubernetesPodOperator), el primer intento en un nodo spot de Karpenter y el reintento en on-demand.
+ Job programado (DAG forecast_nightly)	NUEVA	Airflow lo dispara a las 00:30 con `catchup=False` y `max_active_runs=1`: un pronóstico de una noche pasada no sirve para pedir hoy, y dos noches a la vez se pelearían el puntero.
10 Rollout	CAMBIA	Sin canary de tráfico: el challenger corre una noche en sombra por el mismo gate y se promueve moviendo el alias `champion`. El valor de negocio se mide con 40 tiendas piloto contra 40 de control.
~~11 Request~~	DESAPARECE	Nadie llama en línea: el ERP lee la vista `forecast_current` a las 04:00, por fecha objetivo.
12 Monitoreo	CAMBIA	Dos relojes: la noche (`ForecastRunLate`, `ForecastRunFailed`: page), con métricas que el job empuja al Pushgateway, y la precisión (`WAPEDegraded`: ticket), que llega días después con las ventas.

En el curso: C14 · Arquitectura · C14 · Serving · C14 · Monitoreo, alertas y reentrenamiento · M2 · Batch: precomputar en vez de servir

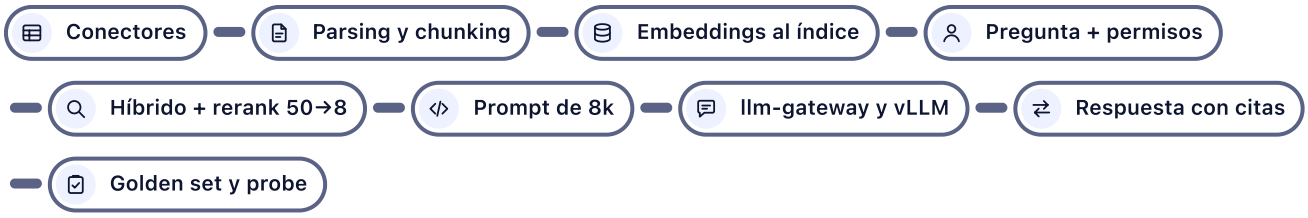
LLM / RAG RAG empresarial (C11)

En RAG casi nada se entrena: el LLM ya existe y lo sirve la plataforma del C10 (`llm-gateway` → vLLM), y el C11 es un tenant más. El trabajo se mueve a la ingestión (conectores, chunking, embeddings y permisos) y a la evaluación con un golden set, porque ahí están los errores que el LLM no puede arreglar. La request pasa de milisegundos a segundos, con streaming de tokens y citas.

FRENTE AL C01 · 8 CAMBIAN · 2 DESAPARECEN · 5 NUEVAS · 2 IGUAL

1 Datos — + Ingestión con conectores — + Carril rápido de permisos — 2 Versionado — ~~3 Entrenamiento~~
— + Índice híbrido en OpenSearch — 4 Gate — ~~5 Export~~ — 6 Serving — + Plataforma LLM del C10
— 7 Empaquetado — 8 CI/CD — 9 Kubernetes — 10 Rollout — 11 Request — 12 Monitoreo
— + Probe de permisos

SU PROPIA LÍNEA



VELOCIDADES La línea de semanas no entrena: la reemplaza la ingestión continua (frescura p95 ≤ 15 min, revocaciones en ≤ 5 min) y, cada trimestre, la evaluación de embeddings candidatos. La de días mueve código, prompt, índice o la revisión del LLM, todos por el mismo gate. La de milisegundos se vuelve de segundos (retrieval + rerank p95 ≤ 700 ms, primer token p95 ≤ 2.5 s). La continua corre el probe de permisos cada pocos minutos y el canary de groundedness cada 4 h.

1 Datos	CAMBIA	No hay dataset de entrenamiento: los datos son ≈ 1.5 M documentos de Confluence, Drive, Jira y políticas, cada uno con su ACL, más un golden set de ≈ 600 preguntas escritas con especialistas de cada área.
+ Ingestión con conectores	NUEVA	Un conector por fuente detecta cambios por webhook o polling; después vienen parsing y OCR, y chunks de ≈ 400 tokens con su ruta de títulos y su ACL.
+ Carril rápido de permisos	NUEVA	ACL sync reescribe solo los campos `acl_0…acl_2` de los chunks afectados: una revocación aplica en ≤ 5 min sin re-embeber ni esperar detrás de un PDF.
2 Versionado	CAMBIA	Se versionan el texto parseado (en S3, por hash), cada chunk con su hash y la `model_version` del embedding, el índice detrás de un alias y el golden set como dataset.

~~3 Entrenamiento~~	DESAPARECE	El LLM (`chat-default` del C10) y el modelo de embeddings vienen hechos; en su lugar se embeben ≈ 12 M chunks (≈ 44 L4-horas).
+ Índice híbrido en OpenSearch	NUEVA	El embed-worker (L4, d = 1024) hace upsert solo de los chunks cuyo hash cambió; BM25 y HNSW viven en el mismo índice, detrás de un alias.
4 Gate	CAMBIA	El golden set corre contra el sistema completo: recall@8, faithfulness y precisión de citas con un juez calibrado, y permisos al 100 %. Lo pasa cualquier cambio de chunker, embeddings, reranker, prompt o LLM.
~~5 Export~~	DESAPARECE	No hay ONNX ni TensorRT propios: en la plataforma del C10, vLLM carga pesos FP8 de una revisión fijada, copiados de S3 al NVMe y verificados con sha256.
6 Serving	CAMBIA	`rag-api` (FastAPI, `POST /v1/ask` con SSE): principals → BM25 ∥ k-NN con RRF (k = 60) → reranker 50 → 8 → prompt de 8k → `llm-gateway` → citas validadas.
+ Plataforma LLM del C10	NUEVA	`llm-gateway` con la llave del tenant `employee-assistant`, cuota en tokens/min y alias `chat-default` → réplicas de vLLM (≈ 70B, 8k de contexto).
7 Empaquetado	IGUAL	Contenedores como en el C01 para `rag-api`, encoder y reranker; vLLM corre con su imagen oficial en la plataforma del C10.
8 CI/CD	IGUAL	Pruebas e imágenes como en el C01; lo que decide si un cambio sale es el gate del golden set.
9 Kubernetes	CAMBIA	El LLM corre en el EKS del C10 (pods de vLLM con 2 × H100); el C11 suma L4 para encoder y reranker y su propio clúster de OpenSearch de 3 nodos.
10 Rollout	CAMBIA	Un modelo de embeddings nuevo es un índice nuevo (blue/green): se compara en shadow con preguntas reales y se cambia el alias junto con el encoder; el índice viejo queda unos días para rollback.
11 Request	CAMBIA	De milisegundos a segundos: retrieval + rerank p95 ≤ 700 ms, primer token p95 ≤ 2.5 s y respuesta en streaming ≤ 10 s. Si la foto de grupos del IdP tiene más de 5 min, responde 503.
12 Monitoreo	CAMBIA	Se suman el atraso de la ingestión y de los permisos (`ACLSyncStale`), la tasa de retrieval sin resultados y un canary de groundedness cada 4 h. No hay reentrenamiento: el lazo se cierra con cambios de chunking, prompt o índice que pasan el mismo gate.
+ Probe de permisos	NUEVA	Cuentas sintéticas, con y sin acceso, preguntan cada pocos minutos por documentos canario: si alguien sin acceso los recupera, es un incidente de seguridad, no un ticket.

En el curso: C11 · Arquitectura · C11 · Pipeline de datos y entrenamiento · C11 · Serving · C10 · Arquitectura · M14 · RAG: el modelo con tus documentos

STREAMING Speech-to-text en vivo (C13)

Es **streaming inference sobre una conexión larga**, no stream processing sobre Kafka: cada participante abre una sesión y manda audio en fragmentos de 160 ms mientras el modelo avanza con estado en una GPU. Por eso cambian el balanceo (por streams activos, con afinidad), el autoscaling (sube por sesiones y baja drenando) y el monitoreo (latencia de parciales y un canary que habla en producción, porque el audio de clientes no se guarda).

FRENTE AL C01 · 8 CAMBIAN · 0 DESAPARECEN · 4 NUEVAS · 4 IGUAL

1 Datos — 2 Versionado — 3 Entrenamiento — 4 Gate — 5 Export — 6 Serving
+ Sesión por pista (gRPC bidireccional) — 7 Empaquetado — 8 CI/CD — 9 Kubernetes
+ Router de sesiones con afinidad — 10 Rollout — 11 Request — + caption-bus por reunión — 12 Monitoreo
+ Canary golden en producción

SU PROPIA LÍNEA

Audio con licencia — Fine-tune y WER — Engine para L4 — Sesión por pista
Router con afinidad — asr-worker con estado — caption-bus a pantalla

Escala por streams — Canary golden

VELOCIDADES La línea de milisegundos dura lo que la reunión: un fragmento cada 160 ms, parciales p95 ≤ 500 ms y sesiones de horas. La continua sigue la curva del día (≈ 60 L4 de noche, ≈ 172 en el pico) con un cron antes de la rampa de la mañana. La del modelo hace fine-tune por idioma cuando un disparador lo pide, revisa cada trimestre y entra por canary de sesiones nuevas.

1 Datos	CAMBIA	El audio de clientes no se guarda: el audio etiquetado sale de corpus con licencia, grabaciones por acento con consentimiento y clientes con opt-in, transcrito por personas con una guía.
2 Versionado	IGUAL	Golden sets congelados por idioma y slice (acento, ruido, dispositivo), separados por hablante de los datos de entrenamiento.
3 Entrenamiento	CAMBIA	Se parte de un ASR streaming preentrenado (transducer/CTC) y, si hace falta, se ajusta por idioma: nada se entrena desde cero.
4 Gate	CAMBIA	WER por idioma y slice con normalización fija, más latencia y streams por GPU medidos en una L4, contra el modelo en producción.
5 Export	IGUAL	Como en el C01, el engine se compila para la GPU que sirve (L4); el registry guarda además la config de streaming (fragmento, lookahead, endpointing).
6 Serving	CAMBIA	`asr-gateway` en CPU (Opus → 16 kHz, VAD, fragmentos de 160 ms) y `asr-worker` en L4 con el sequence batcher de Triton: estado por stream, parciales y finales.
+ Sesión por pista (gRPC bidireccional)	NUEVA	El SFU abre un stream por participante hacia el `asr-gateway` (≈ 200k conectadas en el pico); con VAD, solo las pistas con voz gastan GPU.
7 Empaquetado	IGUAL	Gateway y worker en contenedores, como en el C01; el worker corre como Deployment de GPU.
8 CI/CD	IGUAL	Pruebas e imágenes como en el C01.
9 Kubernetes	CAMBIA	KEDA escala `asr-worker` por streams activos por GPU (objetivo 350) y un cron sube el mínimo antes de la mañana. Un nodo L4 tarda 5–13 min, y bajar réplicas es drenar (`preStop`).
+ Router de sesiones con afinidad	NUEVA	Asigna cada sesión a un `asr-worker` de su zona contando streams activos, no requests ni CPU, y la mantiene ahí (sticky).
10 Rollout	CAMBIA	Canary por porcentaje de sesiones nuevas, no de requests, porque cada sesión queda asignada a su worker (sticky).
11 Request	CAMBIA	En vez de una request, una sesión gRPC larga por pista: parciales p95 ≤ 500 ms y finales p95 ≤ 1.5 s después del fin de la frase.
+ caption-bus por reunión	NUEVA	Pub/sub en memoria (NATS o Redis Pub/Sub): los workers publican parciales y finales, y la señalización WebSocket existente los empuja a cada cliente.
12 Monitoreo	CAMBIA	Latencia por tramo, streams activos (también la señal del autoscaling), reconexiones y fracción de voz; el canary golden mide WER en producción. Se reentrena por disparadores, con revisión trimestral.
+ Canary golden en producción	NUEVA	Un bot entra a una reunión interna y reproduce audio del golden set: mide WER y latencia de punta a punta sin tocar audio de clientes.

En el curso: C13 · Arquitectura · C13 · Serving · C13 · Monitoreo, alertas y reentrenamiento · M2 · Streaming: inferencia sobre eventos

EDGE Inspección en planta (C03)

El modelo corre junto a la línea, en un Jetson, porque la línea no espera a la red ni puede parar si se cae internet; la nube entrena y monitorea. Nada de la nube está en el camino de una pieza: Kubernetes y el autoscaling desaparecen (4 dispositivos fijos con Docker), el release se vuelve **OTA** a una flota a través de la DMZ y el monitoreo funciona sin internet.

FRENTE AL C01 · 10 CAMBIAN · 1 DESAPARECEN · 4 NUEVAS · 1 IGUAL

1 Datos — 2 Versionado — 3 Entrenamiento — 4 Gate — 5 Export — 6 Serving — 7 Empaquetado — 8 CI/CD



VELOCIDADES La línea de milisegundos vive en el dispositivo y es dura: 2 piezas/s por línea y decisión en ≤ 350 ms, sin red. La del modelo es mensual (o antes, por una variante o un defecto nuevo, drift o un escape que reporta la armadora) y llega en días: descarga de noche, un turno en shadow, canary y las 4 líneas. La continua es local (Prometheus y andon en la planta) y lo demás sube a la nube cuando hay enlace.

1 Datos	CAMBIA	Las imágenes salen de la cámara de la línea, y el veredicto que registra quien inspecciona, en la tablet, es el label más barato. Al inicio hay ≈ 2,000 piezas buenas y ≈ 300 defectos.
2 Versionado	IGUAL	Snapshots inmutables en S3, golden sets congelados por tipo de defecto y linaje pieza ↔ imagen ↔ modelo ↔ decisión.
3 Entrenamiento	CAMBIA	Job de GPU en la nube, bajo demanda: anomalías entrenadas con piezas buenas y, cuando hay defectos suficientes, un clasificador por tipo. Mensual o por disparador.
4 Gate	CAMBIA	Recall por tipo de defecto a FP fijo en los golden sets, y umbral elegido por costo esperado con recall ≥ 99.5 % (`choose_threshold.py`).
5 Export	CAMBIA	El engine se construye en un Jetson de CI idéntico a los de la línea (mismo módulo, JetPack y modo de energía), con paridad contra el ONNX y latencia a batch fijo.
6 Serving	CAMBIA	Un proceso en el Jetson Orin NX: adquiere la imagen, arma 20 tiles, corre TensorRT FP16 y escribe al PLC por Modbus TCP; si no alcanza, un deadline guard escribe `DIVERT` .
7 Empaquetado	CAMBIA	Imagen arm64 sobre `l4t-tensorrt` , con `docker compose` (`runtime: nvidia` , red del host); los engines se montan aparte para volver al anterior sin red.
8 CI/CD	CAMBIA	Cada versión se registra firmada (ONNX, engine por módulo y JetPack, umbrales y contenedor) junto con su plan de rollout por líneas.
+ OTA por la DMZ	NUEVA	La DMZ baja los artefactos firmados con conexiones salientes, verifica la firma y los copia a un registry local de la red OT: los dispositivos nunca salen a internet.
~~9 Kubernetes~~	DESAPARECE	4 dispositivos fijos no justifican Kubernetes: Docker con restart policy y un watchdog. Tampoco hay autoscaling: se dimensiona por dispositivo.
10 Rollout	CAMBIA	Por línea: de noche el engine baja al slot B con un smoke test, la línea 1 corre un turno en shadow, después canary y las 4 líneas. El rollback es local, sin red.
11 Request	CAMBIA	No hay HTTP: la fotocelda dispara la cámara por hardware en cada pieza (2 por segundo por línea) y la decisión tiene un deadline de 350 ms.
+ PLC con default seguro	NUEVA	El PLC deja pasar una pieza solo si llegó un PASS con su número antes del deadline. Jetson caído, red caída o decisión tarde terminan igual: desviada.
12 Monitoreo	CAMBIA	Prometheus y andon en la planta, porque puede no haber internet; a la nube suben heartbeats y agregados. Si la tasa de rechazo supera 2× su base durante 30 min, lo probable es luz, lente o foco, no defectos.
+ Store-and-forward a la nube	NUEVA	Decisiones, imágenes de rechazos y veredictos esperan en el almacén de planta y suben cuando hay enlace: la línea opera ≥ 72 h sin internet.
+ Piezas patrón en cada turno	NUEVA	Cada turno empieza con piezas buenas y defectuosas conocidas; sin ese chequeo, ninguna pieza recibe PASS. Así se detectan un lente sucio, un foco movido o una luz degradada antes de producir.

En el curso: C03 · Arquitectura | C03 · Serving | C03 · Monitoreo, alertas y reentrenamiento | M2 · Edge / on-device

Un mini-mapa por caso (14)

Los mismos cuatro tramos en cada caso del curso, agrupados por patrón de inferencia. El C01 es el online síncrono de toda esta hoja.

C09 Fraude en tiempo real

ONLINE SÍNCRONO

Cada pago con tarjeta no presente se puntúa dentro de la autorización (aprobar, rechazar, step-up con 3DS o revisión) en p99 ≤ 100 ms.

- **DATOS**
  - contracargos de hasta 120 días
  - Kafka txn-events → lake
  - training set point-in-time, misma definición
- **MODELO**
  - GBDT (LightGBM-class), ≈ 300 features
  - umbrales por costo y segmento
  - reentrenamiento mensual, split temporal
- **RELEASE**
  - challenger en shadow → fracción → champion
  - policy.yaml con la versión del modelo
  - cada regla nueva pasa por shadow
- **OPERACIÓN**
  - fraud-scoring en CPU: 3 × c7i.2xlarge por zona
  - contadores síncronos en Redis + Flink
  - fallback: solo reglas, nunca bloquear todo

FRENTE AL C01

- Árboles en CPU (≈ 1–2 ms por predicción): no hay GPU, Triton ni export a TensorRT.
- Las features viven fuera del modelo: contadores síncronos y un online store que llena Flink, con la misma definición online y offline.
- Los labels tardan hasta 120 días: se evalúan cohortes maduras y se monitorea con proxies.

pico de diseño ≈ 2,000 TPS, p99 ≤ 100 ms

disponibilidad 99.99 %

En el curso: C09 · Arquitectura

M2 · Streaming: inferencia sobre eventos

C15 CTR de anuncios

ONLINE SÍNCRONO

El ad server de un medio de noticias elige 3 anuncios por página en ≤ 50 ms: ordena por bid × pCTR y cobra con segundo precio, así que el pCTR tiene que estar calibrado.

- **DATOS**
  - Kafka ad-events: decisiones, impresiones, clics
  - label: clic ≤ 30 min tras una impresión viewable
  - negativos submuestreados al 5 % (w)
- **MODELO**
  - regresión logística con hashing → DNN chico
  - corrección por w + isotónica
  - gate: NE y calibración 0.9–1.1
- **RELEASE**
  - un artefacto: pesos + calibrador + w
  - carga en caliente en ad-api
  - canary por hash del request
- **OPERACIÓN**
  - ad-api en CPU: 7 × c7i.4xlarge en el pico
  - Redis: topes de frecuencia y features
  - pacing cada minuto por campaña

FRENTE AL C01

- El pCTR fija el precio por clic, no solo el orden: la calibración por slot y sección es gate y alerta.
- Se reentrena cada día (warm start sobre 14 días) con negativos submuestreados; el C01 reentrena cada mes.
- Modelo en CPU con 50 ms de presupuesto: si no responde, se ordena por bid × CTR histórico, y si ad-api falla entra el programático.

≈ 100 M pageviews/día, pico de diseño ≈ 6,000 RPS

1 % más de RPM ≈ US$75k/mes

En el curso: C15 · Arquitectura

M2 · Online síncrona (request/response)

C14 Pronóstico de demanda

BATCH

Cada noche pronostica P50 y P90 a 14 días de 4.8 M series tienda × producto; a las 04:00, el ERP calcula con eso los pedidos de ≈ 400 tiendas.

- **DATOS**
  - sales_daily (00:30) y stock_daily con oos_flag
  - price_promo + calendar_mx (quincenas, Buen Fin)
  - días agotados fuera de los targets
- **MODELO**
  - GBDT global por cuantiles (α 0.5 y 0.9)
  - backtest de 8 orígenes semanales (WAPE)
  - domingo: ≈ 490 M filas en spot
- **RELEASE**
  - imagen forecast-job fijada por digest
  - DAG forecast_nightly en Airflow 3
  - una noche en sombra → alias champion
- **OPERACIÓN**
  - pod en spot, reintento en on-demand
  - gate + swap atómico de forecast_current
  - fallback: ayer desplazado; guardián 03:45

FRENTE AL C01

- No hay endpoint: un job programado escribe ≈ 67 M filas por noche y el ERP las lee de una vista.
- Publicar es mover un puntero tras un gate; si algo falla, sale el pronóstico de ayer desplazado un día.
- GBDT en CPU (≈ 67 core-min por noche): sin GPU, sin ONNX ni TensorRT.

≈ 67 M filas por noche, publicadas antes de las 03:30

WAPE ≈ 45 % hoy → meta ≤ 35 %

En el curso: C14 · Arquitectura

M2 · Batch: precomputar en vez de servir

C04 Segmentación de imágenes

ASÍNCRONO CON COLAS

Segmenta tumor en ≈ 1,200 láminas completas (WSI) al día y deja en el visor el área y el % de tumor, que se aceptan o corrigen antes de firmar; nadie espera en milisegundos.

DATOS

- WSI de ≈ 1 GB (SVS y DICOM) en S3
- correcciones de patología (labels sesgados)
- active learning: 20 h de patología al mes

MODELO

- U-Net-class de ≈ 30 M a 20×
- golden set de 100 láminas, gate por slice
- engine TensorRT FP16 en L4

RELEASE

- imagen con el engine horneado
- silent mode en la cola slide-jobs-backfill
- rollback: kubectl rollout undo

OPERACIÓN

- cola SQS slide-jobs + DLQ tras 3 intentos
- seg-worker en g6.2xlarge, KEDA 0–4
- results-api + visor; audit log con Object Lock

FRENTE AL C01

- Asíncrono: el SLO es turnaround p95 ≤ 60 min, no un p99 en ms, y una cola SQS absorbe las ráfagas de los escáneres.
- KEDA escala los workers de 0 a 4 por la cola: de noche no hay GPU prendida, en vez de N+1 todo el día.
- El costo dominante es el tiempo experto: ≈ US$4k/mes de patología contra ≈ US$1.2k de infra.

≈ 1,200 WSI/día, ≈ 3,000 tiles por lámina

turnaround p95 ≤ 60 min

En el curso: C04 · Arquitectura

M2 · Asíncrona con colas

C05 Documentos y OCR

ASÍNCRONO CON COLAS

Una financiera PyME automatiza la captura de ≈ 5,000 documentos al día (CFDI, INE, comprobantes, estados de cuenta) y manda a revisión humana solo los campos dudosos.

DATOS

- 2 años de capturas sin cajas (labels débiles)
- alineación fuzzy a las palabras del OCR
- gold set de 2,000 docs con doble captura

MODELO

- extractor layout-aware por familia → ONNX
- calibración y umbral por campo
- gate: ningún campo bajo su objetivo

RELEASE

- modelo + archivo de umbrales juntos
- alias de producción por tipo de documento
- statemachine.asl.json (Step Functions)

OPERACIÓN

- docs-api: quality gate p95 ≤ 3 s
- doc-worker en CPU (ONNX Runtime) + Textract
- revisión ≈ 45 s/doc; QA audita 2 % de STP

FRENTE AL C01

- Una cadena (clasificar → texto → extraer → validar → rutear) orquestada con Step Functions, una ejecución por documento, no una sola llamada al modelo.
- Todo en CPU con ONNX Runtime: menos de una página por segundo en el pico no justifica GPU ni Triton.
- La salida es una decisión por campo (STP o revisión humana), con exactitud ≥ 99.5 % en lo aceptado.

≈ 5,000 docs/día, ≈ 12,000 páginas

STP ≥ 60 % al lanzar → 80 % en 12 meses

En el curso: C05 · Arquitectura

M2 · Asíncrona con colas

C10 Servir un LLM propio

LLMS Y RAG

El marketplace deja de mandar todo a una API externa y sirve un modelo open-weights de ≈ 70B en su propia flota de H100, con la API como respaldo gobernado.

DATOS

- golden sets por producto (soporte: ≈ 1,000 chats)
- evaluadores con rúbrica que calibran al juez
- uso por tenant: tokens y US$

MODELO

- pesos FP8 en safetensors, revisión fijada
- vllm bench serve → capacidad por réplica
- gate con juez LLM por pares

RELEASE

- registry: revisión + reporte de eval
- alias chat-default en la config del gateway
- canary eval cada hora

OPERACIÓN

- llm-gateway: llaves y cuotas en tokens/min
- vllm-70b: 2 × H100, TP=2, 16 réplicas
- batch-runner: ≈ 2 M ítems en 8 h

FRENTE AL C01

- No se entrena: se baja una revisión de pesos abiertos (con la licencia revisada) y se evalúa contra golden sets por producto.
- Se mide por token (TTFT p95 ≤ 1.0 s, TPOT p95 ≤ 50 ms) y la concurrencia la limita el KV cache.
- Una réplica nueva tarda 5–10 min en arrancar: la flota del pico queda caliente en horario laboral y los picos se absorben con cuotas y colas en el gateway.

16 réplicas = 32 H100 = 4 × p5.48xlarge

≈ US$160.6k/mes propio vs ≈ US$442k/mes de API

En el curso: C10 · Arquitectura

M14 · API de terceros vs modelo propio

C11 RAG empresarial

LLMS Y RAG

Un asistente interno responde con los documentos de la empresa (Confluence, Drive, Jira) y cita sus fuentes, respetando los permisos de cada persona.

DATOS

- Confluence, Drive, Jira y políticas
- ≈ 1.5 M docs → ≈ 12 M chunks de ≈ 400 tokens
- golden set de ≈ 600 preguntas con SMEs

MODELO

- embeddings multilingües, d = 1024
- reranker cross-encoder ≈ 0.6 B: 50 → 8
- chat-default del C10 (≈ 70B, 8k)

RELEASE

- índice nuevo blue/green detrás de un alias
- gate: recall@8, faithfulness, permisos
- shadow del índice nuevo con preguntas reales

OPERACIÓN

- rag-api: POST /v1/ask con SSE
- OpenSearch BM25 ∥ k-NN con la ACL adentro
- ACL sync ≤ 5 min + probe de permisos

FRENTE AL C01

- Nada se entrena: el trabajo es la ingestión (parsing, chunking, embeddings) y mantener índice y permisos al día.
- El permiso es un filtro dentro del retrieval: un chunk que la persona no puede abrir nunca llega al prompt.
- La generación corre en la plataforma del C10 (llm-gateway → vLLM): el C11 es un tenant más.

≈ 25k preguntas/día, pico ≈ 2 qps

primer token p95 ≤ 2.5 s, revocación ≤ 5 min

En el curso: C11 · Arquitectura

M14 · RAG: el modelo con tus documentos

C12 Agente con herramientas

LLMS Y RAG

El asistente de soporte pasa de responder a resolver: consulta pedidos, cambia direcciones y hace reembolsos con herramientas reales, dentro de límites y de una política en código.

DATOS

- step log en Postgres, paso por paso
- ≈ 400 escenarios con backends simulados
- reaperturas en 7 días, CSAT, reembolsos

MODELO

- chat-default del C10 con tool calling
- revisión = modelo + prompt + schemas + política
- gate: 0 violaciones de política

RELEASE

- eval runner: ≈ 4 corridas por escenario
- canary por tipo de conversación
- chat-default fijado por tenant

OPERACIÓN

- loop: ≤ 8 tools/turno, ≤ 12 llamadas, 40k tokens
- reembolso automático ≤ US$50; si no, aprobación
- traza OTel con un span por paso

FRENTE AL C01

- Cada conversación es un loop de ≈ 6 llamadas al LLM con efectos reales: idempotency keys y política en código, no en el prompt.
- Lo que sale a producción es una revisión del agente que pasa ≈ 400 escenarios con cero violaciones, no solo un modelo.
- El estado es durable: un turno se reanuda tras una caída desde el step log, y una aprobación puede esperar horas.

≈ 150k conversaciones/día, ≈ 31 rps al LLM en el pico

primera respuesta p95 ≤ 4 s, turno p95 ≤ 12 s

En el curso: C12 · Arquitectura

M14 · Agentes: loops con herramientas y límites

C06 Moderación de contenido

HÍBRIDOS

Una red social revisa 30 M posts y 150 M comentarios al día contra 7 políticas: chequeo rápido antes de publicar, análisis profundo después y ≈ 600 puestos de moderación humana para lo dudoso.

DATOS

- decisiones con la versión de la política
- muestra aleatoria de vistas (prevalencia)
- listas de hashes PDQ (industria + removidos)

MODELO

- clasificadores destilados de texto e imagen
- deep-scorer multimodal con OCR
- LLM con la política como prompt (provisional)

RELEASE

- policies.yaml: política × severidad × superficie
- gate por política → shadow → canary
- reentrenamiento semanal

OPERACIÓN

- mod-gateway p99 ≤ 150 ms, fail-open S1–S3
- Triton etapa 1: 4 L4 texto + 2 L4 imagen
- Kafka mod-events → deep-scorer → review-router

FRENTE AL C01

- Una cascada (hashes → modelo barato → modelo caro → persona) en lugar de un modelo: solo ≈ 3 % de los ítems llega a la etapa cara.
- Umbrales y acciones viven en una config de política versionada, revisada por los equipos de política y legal, aparte del código del modelo.
- Las personas son parte del sistema y del costo: ≈ US$900k/mes de moderación contra ≈ US$5.9k de GPUs.

pico ≈ 6,250 textos/s + ≈ 625 imágenes/s

≈ 216k revisiones/día de capacidad

En el curso: C06 · Arquitectura

M2 · Patrones híbridos

C07 Búsqueda con embeddings

HÍBRIDOS

Buscar entre ≈ 100 M publicaciones del marketplace del C01 describiendo lo que quieres o subiendo una foto: embeddings tipo CLIP y BM25 en paralelo.

DATOS
- listing-events en Kafka (key = id)
- juicios humanos 0–3 (consulta, listing)
- clics y contactos en search-logs

MODELO
- dual encoder CLIP-class, d = 512
- gate NDCG@10 en head, torso y tail
- reranker cross-encoder sobre el top-30

RELEASE
- re-embed de 100 M: ≈ 3 h en 8 L4
- índice green + shadow → swap del alias
- el mismo artefacto en encoder y embed-worker

OPERACIÓN
- OpenSearch k-NN: 4 shards × 2, HNSW int8
- BM25 ∥ ANN → RRF (k = 60)
- tombstones en Redis: vendido fuera en ≤ 1 min

FRENTE AL C01

- Cambiar de modelo obliga a re-embeber el catálogo y construir un índice nuevo (blue/green); no basta con cambiar un engine.
- Hay un índice con estado que mantener fresco: lo nuevo es buscable en p95 ≤ 5 min vía Kafka.
- La calidad se mide como ranking (NDCG@10, recall del ANN ≥ 0.95 contra búsqueda exacta), no como accuracy.

≈ 1,750 QPS en el pico, texto p99 ≤ 300 ms

100 M × 512 en FP32 ≈ 205 GB → 51.2 GB en int8

En el curso: C07 · Arquitectura

M2 · Patrones híbridos

C08 Sistema de recomendación

HÍBRIDOS

El home del marketplace muestra a cada persona 50 publicaciones personalizadas: candidatos de 4 fuentes, ranking y re-ranking en ≤ 200 ms.

DATOS
- home-impressions con snapshot de features
- contacto en 24 h (label demorado)
- Flink: contadores de 1 h y 24 h

MODELO
- two-tower semanal (user + item tower)
- ranker multi-task de ≈ 10 M, diario
- co-visitación diaria en Spark

RELEASE
- ranker en Triton con canary
- índice del item tower en blue/green
- A/B por cuenta: contactos por DAU

OPERACIÓN
- home-api: 4 fuentes con timeout propio
- ranker en Triton: 4 × L4
- online store en Valkey (4 × 2)

FRENTE AL C01

- Un embudo por request (≈ 1,000 candidatos de 4 fuentes → 50), no un solo modelo.
- Las features se leen online (feature store + Flink) y se loguean con cada impresión para entrenar sin skew.
- El ranker se entrena cada día con labels que tardan 24 h; el C01 reentrena cada mes.

≈ 3,100 RPS en el pico, p99 ≤ 200 ms

≈ 3.1 M item-scores/s en el pico

En el curso: C08 · Arquitectura

M2 · Patrones híbridos

C13 Speech-to-text en vivo

STREAMING

Subtítulos en vivo para videollamadas en español, inglés y portugués: el audio de cada participante se transcribe mientras habla, con parciales en ≤ 500 ms.

DATOS
- corpus con licencia + grabaciones por acento
- transcripción humana con guía, doble pasada
- golden sets por idioma, acento y ruido

MODELO
- ASR streaming nativo (transducer/CTC)
- fine-tune por idioma, WER por slice
- engine para L4 + config de streaming

RELEASE
- canary por % de sesiones nuevas
- preStop que drena el worker
- bot que reproduce audio golden en prod

OPERACIÓN
- asr-gateway: VAD + fragmentos de 160 ms
- asr-worker en L4 con sequence batching
- KEDA por streams activos (350 por GPU)

FRENTE AL C01

- No hay request/response: una sesión gRPC larga por participante, con estado en una GPU mientras habla.
- Se balancea y escala por streams activos, no por requests ni CPU; para bajar réplicas se drena.
- El audio de clientes no se guarda: la calidad se mide con golden sets y un bot que habla en producción.

≈ 60k streams activos en el pico → ≈ 172 L4

parciales p95 ≤ 500 ms, finales p95 ≤ 1.5 s

En el curso: C13 · Arquitectura

M2 · Streaming: inferencia sobre eventos

C02 Detección en video

EDGE Y ON-DEVICE

1,000 cámaras en 25 centros de distribución: una caja con una L4 por sitio detecta personas y vehículos en zonas prohibidas, y la nube verifica antes de alertar al SOC.

DATOS
- veredictos del SOC como labels
- hard negatives: sombras, animales, faros
- dataset con split por cámara y fecha

MODELO
- detector ligero YOLO-class a 640 px
- verificador a 1280 px sobre 3 frames
- engine TensorRT en CI sobre L4

RELEASE
- bundle en /opt/c02/models + symlink current
- OTA: shadow en 2 sitios → canary → olas
- docker compose up -d en cada caja

OPERACIÓN
- DeepStream: 1 × L4 por sitio, batch 40
- forwarder → API → SQS → verificador
- heartbeat cada 30 s (EdgeBoxOffline)

FRENTE AL C01

- El modelo corre en una caja por sitio, no en un cluster: a la nube suben eventos con clip (≈ 5 MB) en vez de ≈ 60 Mb/s de video por sitio.
- No hay request de una persona: entran 200 frames/s por sitio y sale una alerta al SOC en p95 ≤ 5 s.
- El modelo llega por OTA en olas de sitios, con la versión anterior en disco para rollback, no repartiendo tráfico por pesos en el ALB.

1,000 cámaras, 200 frames/s por sitio

alerta p95 ≤ 5 s (incluye 1.5 s de dwell)

En el curso: C02 · Arquitectura

M2 · Edge / on-device

C03 Inspección visual en planta

EDGE Y ON-DEVICE

Un Jetson por línea decide en ≤ 350 ms si una pieza metálica pasa o se desvía, sin depender de la red; aparte, una caja con DeepStream vigila el EPP en 12 cámaras.

DATOS
- imágenes de rechazos + muestra de aprobadas
- veredicto de inspección en la tablet
- ≈ 2,000 piezas buenas y ≈ 300 defectos

MODELO
- anomalías + clasificador por tipo
- umbral por costo con recall ≥ 99.5 %
- engine en un Jetson de CI idéntico

RELEASE
- imagen sobre l4t-tensorrt + docker compose
- DMZ → registry local en la red OT
- shadow 1 turno → canary → 4 líneas

OPERACIÓN
- Jetson Orin NX, TensorRT FP16, 20 tiles
- PLC por Modbus TCP: sin PASS, desvía
- Prometheus local + piezas patrón por turno

FRENTE AL C01

- No hay Kubernetes ni autoscaling: 4 dispositivos fijos con Docker, restart policy y watchdog.
- El deadline es duro y la falla es segura: sin decisión a los 350 ms, el PLC desvía la pieza.
- La línea opera ≥ 72 h sin internet: alertas locales y datos que suben por store-and-forward.

2 piezas/s por línea, decisión ≤ 350 ms

recall ≥ 99.5 % con FP ≤ 1.5 %

En el curso: C03 · Arquitectura

M2 · Edge / on-device

¿En qué estación falló?

24 síntomas reales del C01.

Para practicar en papel, tapa las dos columnas de la derecha y di la estación y el primer chequeo.

SÍNTOMA	ESTACIÓN	PRIMER CHEQUEO
Después de un merge a main, el manifest del gateway apunta al tag del commit nuevo y los pods nuevos quedan en ImagePullBackOff. kubectl describe pod muestra «Failed to pull image ".../classifier-gateway:...": ... failed to resolve reference ... not found». Los pods viejos siguen sirviendo.	**8 · CI/CD**	`aws ecr describe-images --repository-name classifier-gateway --image-ids imageTag=`*commit*`; si no existe, revisa el job de CI de ese commit (build y push).`
Tras un PR que «limpió» el manifest de Triton, el pod nuevo queda en Pending: «0/5 nodes are available: 2 node(s) had untolerated taint {nvidia.com/gpu: present}, 3 node(s) didn't match Pod's node affinity/selector». El autoscaler de nodos no agrega ninguno y los pods viejos siguen sirviendo.	**9 · Kubernetes**	`kubectl describe pod` *pod_triton* `-n classifier (evento FailedScheduling) y el diff del PR en tolerations y nodeSelector, contra kubectl describe node` *nodo_gpu* `(bloques Taints y Labels).`
Negocio agregó una categoría nueva al árbol hace dos semanas y el ticket de reentrenamiento se abrió ese mismo día. El modelo nunca la sugiere (siempre se elige a mano en el selector) y una categoría vecina empeora en las métricas sin que nadie haya desplegado nada.	**1 · Datos**	Cuenta los ejemplos revisados de la categoría nueva en la cola de etiquetado y en el manifest del último snapshot (meta: ≈ 200–500 antes de reentrenar).
La imagen del gateway que construiste en tu Mac con Apple Silicon corre bien con docker run. En EKS, el pod queda en CrashLoopBackOff y kubectl logs --previous solo dice «exec format error».	**7 · Empaquetado**	`docker image inspect --format '{{.Os}}/{{.Architecture}}'` *imagen* `en tu Mac (o docker buildx imagetools inspect --format '{{json .Image}}'` *imagen* `en el registro): si dice arm64, reconstruye con --platform linux/amd64.`
Sin ningún deploy en tres semanas, el job diario reporta PSI > 0.2 sostenido 3 días en las categorías predichas (suben mochilas y útiles escolares) y la aceptación cae más de 3 pp semana contra semana. Cero errores, p99 normal.	**12 · Monitoreo**	Confirma que no coincide con ningún deploy (modelo, gateway, app), mira en el job qué categorías se movieron y etiqueta una muestra reciente para medir el impacto antes de reentrenar.
Un día después de un deploy del gateway (el modelo es el mismo) salta ClassifierLowConfidenceSpike: la fracción de predicciones con top-1 < 0.3 supera el doble de su promedio de 7 días, en móvil y en web por igual. Cero 5xx, latencia normal.	**6 · Serving**	Corre el test de tensores de referencia (20 imágenes) contra el commit desplegado y revisa el diff de gateway/app/preprocess.py; si los tensores cambiaron, rollback del gateway.
Desde la última versión de la app, el histograma classifier_image_bytes se corrió de ≈ 200 KB a varios MB, classifier_preprocess_seconds subió y el HPA agregó pods al gateway. El modelo y el gateway no cambiaron.	**11 · Request**	En los logs del gateway, image_bytes por source (mobile vs web) antes y después del release de la app; después confirma con el equipo de la app si cambió el downscale.
Tras un refactor de train.py, el fine-tune mensual corre sus ≈ 3–4 h sin errores, pero el gate lo bloquea: top-1 muy por debajo del champion. En MLflow, la loss de entrenamiento casi no bajó en ninguna época, y relanzarlo con el snapshot del mes pasado da lo mismo.	**3 · Entrenamiento**	Corre el smoke test sobre el commit del refactor (sobreajustar un batch de 32 imágenes en CPU o una GPU chica) y compara en MLflow los params de esta corrida con los del mes anterior.
En el deploy de un modelo nuevo, los pods nuevos de Triton quedan en CrashLoopBackOff sin llegar a Ready. kubectl logs --previous: «The engine plan file is generated on an incompatible device... please rebuild» y Triton se cierra. Los pods viejos siguen sirviendo.	**5 · Export**	`kubectl logs` *pod_triton* `-n classifier --previous y, en los tags de la versión del registry, la GPU y la release de NGC con que se construyó el engine (deben ser T4 y 24.08, como triton/Dockerfile).`
El canary estuvo 3 horas al 5 % fuera del pico, todo verde, y alguien lo pasó directo a 100 %. En el pico diurno siguiente, la tasa de 5xx del modelo nuevo se duplica.	**10 · Rollout**	Cuenta los requests del canary por ventana con sum(increase(classifier_requests_total{model_version="*version_canary*"}[10m])) y verifica si alguna etapa cubrió el pico diurno.
Cada reentrenamiento mensual pasa el gate (no inferior al champion, margen −0.3 pp), pero al comparar el champion de hoy con el de hace un año sobre el mismo golden set, el top-1 bajó ≈ 2.4 pp.	**4 · Gate**	Evalúa el challenger contra el champion de hace 3 meses (y el de hace 12) en el mismo golden set, con test pareado; las versiones anteriores siguen en el registry de MLflow.

SÍNTOMA	ESTACIÓN	PRIMER CHEQUEO
Quieres reproducir el modelo de septiembre con ds-2026-09-01, pero el sha256 de su rows.parquet ya no coincide con el rows_sha256 de la tarjeta del snapshot, y el reentrenamiento da métricas que no se parecen a las registradas.	**2 · Versionado**	`aws s3api list-object-versions --bucket` *bucket* `--prefix datasets/ds-2026-09-01/`: más de una versión de rows.parquet, o un LastModified posterior al created_at de la tarjeta, confirma la reescritura; después, `aws s3api get-bucket-policy --bucket` *bucket* para ver qué la permitió.
En el pico salta TritonQueueTimeHigh y KEDA pide una tercera réplica de Triton, pero el pod nuevo lleva 20 minutos en Pending y no aparece ningún nodo GPU nuevo.	**9 · Kubernetes**	`kubectl describe pod` *pod_triton* `-n classifier` (eventos FailedScheduling) y el estado del autoscaler de nodos: `kubectl -n kube-system get configmap cluster-autoscaler-status -o yaml`, o `kubectl get nodepools,nodeclaims` con Karpenter.
Dos pods del gateway del mismo Deployment, ambos con la imagen classifier-gateway:latest, se comportan distinto: solo uno escribe un campo nuevo en su línea de log. kubectl rollout undo no devuelve la versión anterior.	**8 · CI/CD**	`kubectl get pods -n classifier -l app.kubernetes.io/name=gateway -o custom-columns=POD:.metadata.name,DIGEST:.status.containerStatuses[0].imageID`: dos digests distintos confirman que el tag se movió.
Para acelerar el fine-tune subiste el batch: el paso falla a los pocos segundos y su log termina en «CUDA out of memory. Tried to allocate …». Alguien propone subir limits.memory del pod.	**3 · Entrenamiento**	`kubectl logs` *pod_finetune*: ¿termina en «CUDA out of memory»? → VRAM. `kubectl describe pod` *pod_finetune*: ¿Reason OOMKilled y Exit Code 137? → RAM del pod.
Un martes a media mañana, sin deploys, suben el tiempo en cola de Triton y el p99; KEDA ya está en 4 réplicas. En el panel de tráfico por caller, source=internal se multiplicó y el resto sigue igual.	**11 · Request**	`sum by (source) (rate(classifier_requests_total[5m]))` junto al tiempo en cola de Triton: si internal sube a la par de la cola, es el vecino; después, averigua qué equipo interno (moderación, búsqueda) lanzó un batch por la API.
En el reporte del pipeline, el top-1 sobre la validación del snapshot (casi toda con labels débiles de la categoría final de cada publicación) sube mes a mes; en el golden set el challenger apenas empata con el champion y la re-categorización por moderación no baja.	**1 · Datos**	Parte la métrica por label_source (humano, moderador, vendedor_filtrado) y compárala con el golden set y con el holdout de ≈ 1 % sin sugerencias.
En el pipeline, parity_check.py --backend triton termina con «[parity] NO PASA: no promuevas este artefacto»: el acuerdo top-1 entre el engine FP16 y PyTorch FP32 quedó bajo 99.5 %. El gate offline del modelo había pasado.	**5 · Export**	`python model/parity_check.py --checkpoint` *checkpoint* `--onnx model/classifier.onnx --images` *muestra_eval* `--n 2000` (backend ort, en CPU) y compara con el resultado del backend triton.
El challenger le gana al champion en el golden set por mucho más que cualquier mes anterior. En shadow, su acuerdo con el champion y su distribución de predicciones son los de siempre, y en el A/B la aceptación no se mueve.	**2 · Versionado**	Cruza los sha256 (y pHash) del golden set con el manifest de entrenamiento del snapshot y revisa el split por seller_id: debe haber cero coincidencias.
Tras un release que solo cambió gateway/app/main.py (gateway.yaml sigue igual, sin limits.cpu), el p99 sube a medida que crece la concurrencia aunque la CPU de los pods no está llena; hasta /healthz tarda.	**6 · Serving**	`git diff` *commit_anterior* *commit_nuevo* `--gateway/app/main.py`: busca un decode o resize llamado directo en la ruta async def, fuera de run_in_threadpool.
En cada rolling update del gateway, cada pod viejo se queda en Terminating los 45 s completos de terminationGracePeriodSeconds, aunque uvicorn tiene --timeout-graceful-shutdown 20. En local, docker stop gw tarda 10 s y docker ps -a muestra Exited (137).	**7 · Empaquetado**	`docker image inspect --format '{{json .Config.Cmd}}'` *imagen*: si empieza con "/bin/sh","-c", pásalo a forma exec (lista JSON), como el CMD de gateway/Dockerfile.
La corrida mensual quedó en verde en el orquestador y nadie recibió aviso, pero no apareció ninguna versión challenger en MLflow. En el historial, evaluación y gate figuran OK; engine, imagen de Triton, paridad y registro, como omitidos.	**4 · Gate**	Abre el log y la salida del paso de gate de esa corrida: si el veredicto es «no pasa» y el paso terminó con exit 0, ese es el hueco; después mira en el reporte qué criterio falló.
El análisis del canary revierte la release casi todas las madrugadas por el criterio «PSI de categorías contra el estable > 0.1»; de día nunca falla y en shadow el acuerdo con el champion era el de siempre.	**10 · Rollout**	Cuenta las predicciones del canary en la hora de cada rollback: sum(increase(classifier_top1_score_count{model_version="*version_canary*"}[1h])); si son unos pocos miles, el criterio necesita un n mínimo y agrupar categorías.
En el Hot Sale, sin deploys recientes, salta ClassifierHighP99Latency y el tramo que crece es classifier_inference_seconds. Triton sigue en 2 réplicas, TritonQueueTimeHigh nunca se disparó y los paneles de Triton en Grafana están vacíos.	**12 · Monitoreo**	`kubectl describe scaledobject triton -n classifier` (eventos KEDAScalerFailed) y, en Prometheus, busca nv_inference_queue_duration_us{model="classifier"}; si no hay serie, `kubectl get servicemonitor -n classifier`.

El recorrido es el del C01 del curso Ronda de Diseño ML. Los «archivos reales» son extractos de `reference/c01/`; los «descritos en el curso» son piezas que el curso explica pero que no están en el repo.

Las afirmaciones sobre cada herramienta se revisaron contra su documentación oficial (Kubernetes, KEDA, NVIDIA Triton y TensorRT, MLflow, Argo, Prometheus, GitHub Actions, AWS, DVC, lakeFS, Label Studio). Si tu terminal dice otra cosa, créele a tu terminal.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error