Cheat sheet de Git y GitHub Actions

Para trabajar en el repo del equipo sin miedo, escribir un workflow con la documentación abierta y diagnosticar un CI en rojo rápido. Git guarda tu trabajo en commits y ramas; GitHub lo revisa en pull requests; GitHub Actions lo prueba, construye la imagen y la despliega. Cada sección trae la sintaxis como plantilla con variables, las piezas, un ejemplo real del curso, una figura, un truco para recordarla, la trampa típica y un laboratorio opcional. En la versión web, al final hay un diagnosticador de síntomas, un revisor de workflows y un quiz de errores.

Cómo recordarla · Trampa típica · *variable* la cambias tú · Ejemplos en verde

EN LAS FIGURAS: tú y tu máquina · GitHub y el remoto · automático (Actions) · nuevo, avanza o seguro · espera o cambia · se pierde, reescribe o falla · solo en el reflog

0 Nombres y números de bolsillo

¿Qué estados, nombres y números de git y de GitHub Actions conviene saberse de memoria para leer un repo o una ejecución de CI (las pruebas automáticas de cada cambio) en segundos?

SINTAXIS CON VARIABLES

`git status -sb`

La foto rápida del repo. Cada archivo lleva dos letras: la **izquierda** es el staging (lo que entra en el próximo commit, la próxima foto del proyecto que guardes) y la **derecha**, el working tree (tus archivos en disco). La primera línea dice tu rama (la etiqueta que avanza con tus commits), la rama del remoto a la que está conectada (su copia en GitHub) y cuántos commits tienes de más o de menos que ella: `[ahead 1, behind 1]`.

`git log --oneline --graph --all -n` *n*

La historia, un commit por línea: el SHA corto (el nombre del commit), entre paréntesis las etiquetas que apuntan ahí y el mensaje. `HEAD -> main` es dónde estás; `origin/main` es lo último que viste del remoto (la copia del repo en GitHub).

`git rev-parse --short` *ref*

Traduce un nombre (`HEAD`, `main`, `origin/main`, `HEAD~1`) al SHA de su commit; sin `--short`, da los 40 caracteres. Con `--short` acepta un solo nombre por vez y da 7 caracteres en un repo chico, más cuando el repo crece. En el C01, el tag (el nombre de versión) de la imagen Docker del gateway son los primeros 7: `--short=7` los da aunque el repo crezca, como `${GITHUB_SHA::7}` en CI.

`git reflog -n` *n*

El diario de HEAD en tu máquina: cada vez que HEAD se mueve (un commit, un cambio de rama, un reset que mueve la rama a otro commit, un merge o un rebase que juntan ramas) queda una línea `HEAD@{n}`. No viaja con push (subir al remoto) ni con clone (copiar el repo). Si un commit "desapareció", búscalo aquí.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
n	cuántas líneas mostrar	`20`
ref	cualquier nombre de un commit: rama, HEAD, HEAD~1, origin/main o un SHA	`HEAD`
tamaño	bytes del contenido del commit; git lo escribe en un encabezado antes de calcular el SHA	`244: git cat-file -s HEAD`
árbol	la foto de los archivos (tree): nombre, permisos y SHA del contenido de cada uno	`tree d6cd4c0…`
padres	SHA del commit anterior: ninguno en el primero, uno casi siempre, dos en un merge (el commit que junta dos ramas)	`parent 9798182…`
ref ~ *n*	n pasos hacia atrás, siguiendo siempre el primer padre	`HEAD~1 = el commit anterior`
ref ^2	el segundo padre de un merge: la punta de la rama que entró	`HEAD^2 en el merge de un PR (pull request) = el último commit de la rama`
t_{reflog}	cuánto vive una línea del reflog (gc.reflogExpire)	`90 días`

VARIABLE	QUÉ ES	EJEMPLO
$t_{\text{inalcanzable}}$	cuánto vive si su commit ya no está en ninguna rama, como tras un reset --hard (gc.reflogExpireUnreachable); después, git gc (la limpieza que git corre sola de vez en cuando) puede borrar el commit	30 días

FÓRMULAS

$$\text{SHA del commit} = \text{SHA-1}\big(\text{commit } \langle\text{tamaño}\rangle\backslash 0 \| \text{árbol} \| \text{padres} \| \text{autor y committer (quién y cuándo)} \| \text{mensaje}\big)$$

$\text{HEAD}\sim 3 = \text{HEAD}\hat{}\hat{}\hat{}$ $\quad$ $\text{HEAD}\hat{}2 = \text{segundo padre (de un merge)}$

$40 \text{ hex} \times 4 \text{ bits} = 160 \text{ bits (SHA-1)}$ $\quad$ $64 \times 4 = 256 \text{ bits (SHA-256)}$

$t_{\text{reflog}} = 90 \text{ días}$ $\quad$ $t_{\text{inalcanzable}} = 30 \text{ días}$ $\quad$ (defaults de git gc)

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`??`	Untracked: un archivo nuevo que git todavía no sigue. No entra en ningún commit hasta que hagas git add.	`?? .env (va al .gitignore, no a un commit)`
`M a la derecha`	Cambió en disco, pero está fuera del staging: el próximo commit no lo lleva.	`M gateway/app/preprocess.py`
`M a la izquierda`	Cambió y está en staging: el próximo commit lo lleva.	`M gateway/app/main.py`
`MM`	Está en staging y lo volviste a cambiar después: el commit lleva la versión de tu git add, no la última.	`MM gateway/app/main.py`
`A · D · R`	Nuevo, borrado o renombrado, ya en staging.	`R gateway/app/preprocess.py -> gateway/app/preproceso.py`
`UU`	Conflicto sin resolver: las dos ramas cambiaron las mismas líneas. Se resuelve editando el archivo y con git add.	`UU gateway/app/main.py`
`HEAD`	El "estás aquí": casi siempre apunta a una rama, y la rama a un commit.	`cat .git/HEAD → ref: refs/heads/main`
`HEAD (no branch)`	Detached HEAD: HEAD apunta directo a un commit, sin rama. Lo que commitees ahí no lo guarda ninguna etiqueta.	`## HEAD (no branch)`
`main = refs/heads/main`	Una rama: un archivo con el SHA de su último commit. Avanza sola con cada commit que haces estando en ella.	`.git/refs/heads/main (41 bytes)`
`origin/main`	Rama de seguimiento (remote-tracking): tu última foto del main del remoto. Solo se mueve con fetch, pull o push.	`[behind 1] = el remoto tiene 1 commit que tú no`
`upstream`	La rama remota a la que está conectada la tuya: de ahí trae pull, ahí sube push y contra ella te compara status. En la sintaxis de git se escribe @{u}.	`git log --oneline @{u}.. (tus commits sin subir)`
`SHA`	El nombre de un commit: 40 caracteres hexadecimales. Los primeros 7 bastan en un repo chico; git usa más si hace falta.	`3f9c2ab = classifier-gateway:3f9c2ab`
`HEAD~1 · HEAD^2 · HEAD@{2}`	El commit anterior; el segundo padre de un merge; dónde estaba HEAD hace dos movimientos (reflog).	`git show HEAD~1`
`ORIG_HEAD · FETCH_HEAD`	Notas que deja git: dónde estaba HEAD antes de un reset, merge o rebase; y lo último que trajo fetch.	`git reset --hard ORIG_HEAD deshace un merge recién hecho`

QUÉ	DEFAULT O LÍMITE	QUÉ HACER
Rama inicial de git init	master (Git 3.0, todavía sin fecha, la cambiará a main). GitHub crea main, y el git de Apple (Command Line Tools de Xcode) ya trae main en su configuración	git init -b main: todo el equipo habla de la misma rama
SHA de un commit	40 caracteres hex (SHA-1); git muestra 7 en un repo chico, más si el repo crece	El tag de la imagen son los primeros 7: git rev-parse --short=7 HEAD (en CI, ${GITHUB_SHA::7})
Reflog	90 días; 30 si el commit ya no está en ninguna rama	Tu red tras un reset o un rebase. Es local: no viaja con push ni con clone
git pull con ramas divergentes (cada una tiene commits que la otra no)	Sin pull.rebase ni pull.ff configurados, falla (desde Git 2.33.1): Need to specify how to reconcile divergent branches.	Decide una vez: git config --global pull.ff only (solo avanza; si divergieron, eliges tú) o pull.rebase true (tus commits van encima)
Archivos grandes en GitHub	Aviso arriba de 50 MiB; bloqueo arriba de 100 MiB	Pesos (.pt, .onnx, .plan) fuera de git: registry de modelos (donde se guardan sus versiones), S3 o Git LFS (guarda el archivo aparte y deja en git solo un puntero)
Contraseña de GitHub en git	No se acepta desde el 13 de agosto de 2021	Una llave SSH, un token (una clave que creas en GitHub) o gh auth login (sección GitHub)
timeout-minutes de un job (un grupo de pasos que corre en una máquina)	360 min. En los runners de GitHub (las máquinas que presta), un job muere a las 6 h; en uno self-hosted (tuyo), a los 5 días	Pon el tuyo: un job colgado quema minutos hasta el tope
Retención en Actions	90 días para artifacts (archivos que guarda un job) y logs; desde el 1 de octubre de 2026, también para checks, runs y statuses (cada ejecución y sus resultados). Se ajusta: repos públicos, 1–90 días; privados, 1–400	Lo que deba durar más (reportes, evidencia), guárdalo fuera de Actions
Caché de dependencias	10 GB por repo (más, solo con un medio de pago); se borra lo que nadie usa en 7 días	Si la caché nunca se reusa, casi siempre su key (el nombre con que se guarda y se busca) está mal armada
GITHUB_TOKEN (la credencial que GitHub le da a cada job)	En un repo personal o una organización nuevos: solo lectura de contents y packages, y sin permiso para crear o aprobar PRs (pull requests). Vence al terminar el job: 6 h como máximo en runners de GitHub; en self-hosted se renueva hasta 24 h	Pide en permissions: solo lo que el job necesita
Eventos hechos con GITHUB_TOKEN	No disparan workflows (los procesos automáticos de Actions), salvo workflow_dispatch y repository_dispatch (los disparos a mano o por API). Si abre o actualiza un PR, los runs de ese PR esperan aprobación	Si un workflow abre el PR del manifest (el YAML de Kubernetes con el tag de la imagen) con GITHUB_TOKEN, su CI no arranca solo: alguien con permiso de escritura aprueba los runs (o el PR se abre con el token de una GitHub App)
Secrets (claves guardadas en GitHub) en PRs desde forks (copias del repo en otra cuenta)	No llegan al runner (salvo GITHUB_TOKEN)	Un fork no puede usar tus llaves: no pongas deploys en esos PRs

Verbos de bolsillo, en orden: `status` (qué hay) → `diff` (qué cambió) → `log` (de dónde viene) → `reflog` (dónde estuve). Y `git help {comando}` abre el manual de tu versión.

Los defaults y límites que más aparecen en el día a día. Verificados en octubre de 2026 contra git-scm.com (Git 2.56) y docs.github.com.

EJEMPLO REAL · LEER TU RAMA DE PR DEL GATEWAY EN DOS COMANDOS, A MITAD DE UN CAMBIO

TEXTO sesión real con git 2.50.1 en un repo de práctica con la estructura de reference/c01/gateway (los SHAs cambian en cada máquina)

```
$ git status -sb
## feat/gateway-timeout...origin/feat/gateway-timeout [ahead 1]
M  gateway/app/main.py
 M gateway/app/preprocess.py
?? .env

$ git log --oneline --graph --all
```

```
* 4f4fd71 (HEAD -> feat/gateway-timeout) gateway: deadline total de 250 ms
* 3fb9f47 (origin/feat/gateway-timeout) gateway: bajar INFER_TIMEOUT_S a 120 ms
* a469e77 (origin/main, main) gateway: primera versión
```

- `[ahead 1]`: tienes 1 commit (`4f4fd71`) que el remoto no tiene; `git push` lo sube. `[behind 1]` sería lo contrario: te falta 1, y `git pull` lo trae.
- main.py tiene la **M a la izquierda**: el cambio está en staging y entra en el próximo commit.
- preprocess.py tiene la **M a la derecha**: cambió en disco, pero el próximo commit no lo lleva hasta que hagas `git add`.
- `?? .env`: un archivo que git no sigue. Es un secreto local: va al `.gitignore` (la lista de lo que git ignora), nunca a un commit (sección Día a día).
- En el log, `HEAD -> feat/gateway-timeout` es dónde estás y `origin/…` es tu última foto del remoto. `main` y `origin/main` están en el mismo commit: al día, según tu último fetch.

CÓMO RECORDARLA

Lee `git status -sb` en dos columnas: **izquierda = staging, derecha = disco**. **HEAD** es "estás aquí", una **rama** es una etiqueta con un SHA y **origin/main** es tu última foto del remoto. Números: SHA de **40** hex, reflog **90 / 30** días, job de Actions **360** min.

TRAMPA

Leer `origin/main` como el remoto en vivo: es tu foto del último fetch y puede tener días. `git status` no le pregunta nada al remoto, así que "up to date" puede ser falso: corre `git fetch` antes de mirar. Y creer que push sube el reflog o el stash (los cambios que guardaste aparte con git stash): los dos viven solo en tu máquina.

En el curso: M10 · Reproducibilidad: los cinco ingredientes · M5 · Registries y tags inmutables · M10 · CI/CD para modelos: la pirámide de tests · Herramientas · Infraestructura como código, CI y seguridad

La cadena: de tu cambio al deploy

¿Qué pasa, en orden, desde que editas un archivo hasta que esa versión recibe tráfico en producción, y quién hace cada paso?

SINTAXIS CON VARIABLES

`git switch main && git pull`

`git switch -c` *rama*

Cada cambio va en su propia **rama** (una etiqueta que avanza con cada commit, cada foto del proyecto que guardas), creada desde la versión más nueva de main, la rama principal. Los nombres dicen qué es: `feat/…` para algo nuevo, `fix/…` para un arreglo.

`git add` *ruta*

`git commit -m "` *mensaje* **"**

Eliges qué entra (`add`) y guardas la foto con un mensaje (`commit`). El commit vive solo en tu máquina hasta el push.

`git push -u origin` *rama*

Sube la rama a GitHub (`origin` es el nombre corto del repo del equipo). `-u` la conecta con `origin/` *rama* , su copia en GitHub, la primera vez; después basta `git push` . GitHub responde con líneas `remote:` que traen el link para abrir el **pull request** (PR: la propuesta de juntar tu rama con main).

`git rev-parse --short=7 HEAD`

Después del merge (cuando el PR entra a main), parado en main: los primeros 7 caracteres del SHA (el nombre único) del commit que construye CD (la entrega continua). Son el tag de la imagen, `classifier-gateway:` *sha* , igual que `${GITHUB_SHA::7}` en CI (integración continua): de un pod (una copia del servicio corriendo en Kubernetes) llegas al código exacto. Sin `=7` , git da más caracteres cuando el repo crece.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
rama	nombre de la rama: tipo/qué-cambia	`feat/gateway-timeout`, `fix/preprocess-resize`
ruta	archivo que cambiaste	`gateway/app/main.py`
mensaje	qué cambió y dónde, en una línea	`gateway: bajar INFER_TIMEOUT_S a 120 ms`
sha	los primeros 7 caracteres del SHA del commit de main	`3f9c2ab, el tag del gateway en k8s/gateway.yaml del C01`
t_{total}	desde el git push hasta el primer request atendido por la versión nueva	`la suma de abajo; con GitOps (el cluster copia lo que dicen los manifests, los YAML de Kubernetes guardados en git), más lo que tarde en sincronizar`
t_{CI}	CI (integración continua): instalar dependencias y correr los tests en el runner, la máquina limpia donde corre CI	`los 24 tests del gateway, sin GPU`
t_{review}	lo que tarda una persona en leer y aprobar	`depende del equipo, no de la máquina`
$t_{\text{build+push}}$	construir la imagen (el paquete con el código y sus dependencias) para linux/amd64, los nodos x86 del cluster, y subirla a ECR, el almacén de imágenes (registry) de AWS	`con la caché de capas solo se rehace la del código (cheat sheet de Docker)`
t_{rollout}	Kubernetes cambia los pods (las copias del servicio que corren) de a poco hasta que todos pasan la readiness (el chequeo de "listo para recibir tráfico")	`el cheat sheet de Kubernetes le da 5m a rollout status del gateway`

FÓRMULAS

$$t_{\text{total}} \approx t_{\text{CI}} + t_{\text{review}} + t_{\text{build+push}} + t_{\text{rollout}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
working tree	Tus archivos en disco: donde editas.	`gateway/app/main.py abierto en tu editor`
staging (index)	La lista de lo que entra en el próximo commit.	`git add gateway/app/main.py`
commit	Una foto de todo el proyecto, con autor, fecha, mensaje y el commit anterior (su padre).	`3387071 gateway: bajar INFER_TIMEOUT_S a 120 ms`
rama de feature	Etiqueta propia para un cambio: sale de main y vuelve por un PR.	`feat/gateway-timeout`
git push	Sube tus commits al remoto (origin, el repo del equipo en GitHub).	`git push -u origin feat/gateway-timeout`
pull request (PR)	Propuesta de juntar tu rama con main: el diff (las líneas que cambian), la conversación, los checks y la review en un solo lugar.	`PR #42: gateway: bajar INFER_TIMEOUT_S a 120 ms`
CI (integración continua)	Un workflow de GitHub Actions (un archivo YAML que dice qué correr y cuándo) corre los tests en cada push y en cada PR, en una máquina limpia.	`pytest con los 24 tests del gateway`
check	El resultado de cada job de CI (un grupo de pasos del workflow), pegado al PR: verde (pasó), rojo (falló) o pendiente.	`el job de tests del gateway en verde`
review	Otra persona lee el diff y aprueba (Approve) o pide cambios (Request changes).	`un comentario sobre la línea de INFER_TIMEOUT_S`
protección de rama / ruleset	Reglas sobre main: nada de push directo; el merge espera checks verdes y aprobación.	`Require status checks to pass before merging`
merge	Junta la rama con main. GitHub ofrece tres formas: merge commit (un commit que junta las dos), squash (todo en un solo commit nuevo) o rebase (tus commits copiados encima); más en la sección Ramas.	`Merge pull request #42 from equipo/feat/gateway-timeout`
CD (entrega continua)	Otro workflow, en el push a main: construye la imagen, la sube y actualiza lo que corre en producción (el deploy).	`buildx para linux/amd64 → ECR`
tag = SHA	La imagen se nombra con los primeros 7 caracteres del SHA del commit de main. El repo de ECR del C01 tiene tags inmutables: nadie puede reusar ese nombre para otra imagen.	`classifier-gateway:3f9c2ab`
rollout · revert	Kubernetes reemplaza los pods de a poco por los de la imagen nueva. Si sale mal, un revert (commit que deshace otro) vuelve atrás dejando rastro.	`kubectl rollout status deployment/gateway`

TÚ, EN TU MÁQUINA

1 Editar

`git switch -c feat/gateway-timeout`

Rama nueva desde el main al día. Editas gateway/app/main.py: el cambio vive en tu working tree (tus archivos en disco).

2 Staging

`git add gateway/app/main.py`

Eliges qué entra en la foto. git diff --staged muestra exactamente eso.

3 Commit

`git commit -m "gateway: bajar INFER_TIMEOUT_S a 120 ms"`

La foto queda en tu repo local, con un SHA nuevo. Nadie más la ve todavía.

4 Push

`git push -u origin feat/gateway-timeout`

La rama sube a GitHub y queda conectada con origin/feat/gateway-timeout.

EN GITHUB, ENTRE PERSONAS

5 Pull request

`gh pr create --fill`

Con gh, la CLI de GitHub (su programa de terminal), o con el botón Compare & pull request en la web. El PR muestra el diff contra el punto donde nació la rama.

AUTOMÁTICO: CI CON GITHUB ACTIONS

6 CI

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

Un workflow de GitHub Actions corre los 24 tests del gateway, sin GPU, sobre un merge de prueba de tu rama con main. El resultado es un check del PR.

EN GITHUB, ENTRE PERSONAS

7 Review

`git diff main...feat/gateway-timeout`

Otra persona lee el mismo diff (con tres puntos, contra el punto donde nació la rama) y aprueba o pide cambios. Con protección de rama, el merge espera checks verdes y la aprobación.

8 Merge

El botón del PR: merge commit (como git merge --no-ff), squash o rebase. main recibe commits nuevos (uno solo con merge commit o squash; con rebase, uno por cada commit de la rama), y el SHA de su punta es el que viaja a producción.

AUTOMÁTICO: CD CON GITHUB ACTIONS

9 CD: imagen

`docker buildx build --platform linux/amd64 -t $REG/classifier-gateway:$SHA --push gateway/`

Otro workflow, en el push a main: entra a AWS por OIDC (GitHub le da al job un token de corta vida y AWS lo cambia por credenciales temporales: sin llaves guardadas), construye para los nodos amd64 y sube a ECR. El tag son los primeros 7 caracteres del commit de main y, con los tags inmutables del C01, no se puede pisar. $REG es el registry de ECR del C01 (123456789012.dkr.ecr.us-east-1.amazonaws.com). $SHA: en CI, ${GITHUB_SHA::7}; el release.sh del curso usa git rev-parse --short=7 HEAD, que da esos mismos 7 (o más, si otro objeto empieza igual).

10 Manifest

`kubectl -n classifier set image deployment/gateway gateway=$REG/classifier-gateway:$SHA`

El Deployment (el objeto de Kubernetes que mantiene corriendo el gateway) pasa al tag nuevo. Con GitOps, en vez de este comando, un PR cambia la línea image: de k8s/gateway.yaml y Argo CD (el programa que aplica al cluster lo que dice git) la aplica.

KUBERNETES: HACIA EL TRÁFICO

11 Rollout

`kubectl -n classifier rollout status deployment/gateway --timeout=5m`

Kubernetes cambia los pods de a poco; los nuevos reciben tráfico cuando pasan la readiness. Si algo sale mal: revert (sección Deshacer) o volver al tag anterior.

tú · personas en GitHub · automático (Actions) · Kubernetes

Once pasos de tu editor al tráfico. Los cuatro primeros los haces tú; el PR y la review son entre personas; desde el merge todo es automático, y cada paso deja rastro con el SHA del commit de main.

EJEMPLO REAL · TU LADO DE LA CADENA EN LA TERMINAL (GATEWAY DEL C01)

TEXTO — sesión real con git 2.50.1 sobre una copia de reference/c01, con un remoto local en lugar de GitHub (… marca líneas omitidas)

```
$ git switch main && git pull
Already on 'main'
Your branch is up to date with 'origin/main'.
Already up to date.
$ git switch -c feat/gateway-timeout
Switched to a new branch 'feat/gateway-timeout'
# … editas gateway/app/main.py …
$ git diff
…
-INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.15"))  # 150 ms para Triton dentro del p99 ≤ 200 ms
+INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.12"))  # 120 ms para Triton dentro del p99 ≤ 200 ms
…
$ git add gateway/app/main.py
$ git commit -m "gateway: bajar INFER_TIMEOUT_S a 120 ms"
[feat/gateway-timeout 3387071] gateway: bajar INFER_TIMEOUT_S a 120 ms
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push -u origin feat/gateway-timeout
…
 * [new branch]      feat/gateway-timeout -> feat/gateway-timeout
branch 'feat/gateway-timeout' set up to track 'origin/feat/gateway-timeout'.
# En GitHub: abres el PR, CI corre los tests, alguien aprueba y se hace el merge
$ git switch main && git pull
…
Updating 9798182..27bd258
Fast-forward
 gateway/app/main.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git rev-parse --short=7 HEAD
27bd258
```

- `git switch main && git pull` antes de empezar: tu rama nace del main más nuevo. Ojo con `Your branch is up to date`: es según tu último fetch (la última vez que bajaste novedades del remoto); el `Already up to date.` de pull sí le preguntó al remoto.
- El diff muestra la línea real de `gateway/app/main.py`: `-` lo que se va y `+` lo que entra. Es lo mismo que verá quien revise el PR.
- El commit vive solo en tu máquina hasta el push. `-u` conecta tu rama con `origin/feat/gateway-timeout`: desde ahí, `git push` y `git pull` saben a dónde ir. Contra GitHub, el push imprime además líneas `remote:` con el link del PR.
- El CI de un PR prueba un merge de prueba de tu rama con main (`refs/pull/N/merge`), no tu rama sola. Y si el PR tiene conflictos (cambios que chocan con los de main), los workflows de `pull_request` no corren.
- Después del merge, `27bd258` es el commit de main que construye CD: la imagen queda como `classifier-gateway:27bd258`. En los archivos del C01, ese lugar lo ocupa `3f9c2ab`.
- Esta cadena es la del gateway. La imagen de Triton (el servidor que corre el modelo en la GPU) llega por otro camino: el pipeline del modelo (los pasos que convierten el checkpoint en engine y lo empaquetan) la construye en un runner con GPU T4, prueba la paridad (que dé los mismos resultados que el modelo original) sobre esa misma imagen y la sube con la versión del modelo como tag, antes de registrar esa versión.

CÓMO RECORDARLA

Git guarda, GitHub revisa, Actions prueba y construye, Kubernetes despliega. Tu parte termina en el push y el PR; desde el merge, todo es automático y deja rastro: commit → check → imagen `:SHA` → rollout.

TRAMPA

Creer que el merge es el deploy. El merge solo cambia main: a producción llega la imagen con el SHA de ese commit, cuando CD la construye, la sube y el manifest la apunta. Y abrir el PR con conflictos: los workflows de `pull_request` no corren hasta que los resuelvas, así que no hay checks que mirar.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M10 · El validation gate: cuándo un modelo puede reemplazar al actual · M5 · Registries y tags inmutables · C01 · Archivos de referencia

1 Qué es Git: fotos, etiquetas y tres zonas

¿Qué guarda git en realidad, qué es una rama, qué es HEAD y por qué hay un paso entre editar y commitear?

SINTAXIS CON VARIABLES

`git init -b` *rama*

Crea la carpeta `.git` : ahí vive el **repositorio**, con todos los commits (las fotos del proyecto), las ramas (etiquetas que avanzan con cada commit) y los tags (etiquetas fijas). `-b main` le pone nombre a la primera rama; sin eso, git usa `init.defaultBranch` y, si no está configurada, `master` (Git 3.0, todavía sin fecha, la cambiará a `main`). GitHub llama `main` a la rama por defecto.

`git cat-file -p` *ref*

Muestra por dentro un objeto (cada cosa que git guarda: un commit, una carpeta o un archivo). Un commit tiene su `tree` (la foto de los archivos), su `parent` (el commit anterior), autor y committer (quién lo escribió y quién lo guardó, con sus fechas) y el mensaje.

`git branch` *rama*

`git rev-parse` *rama* `main`

Crear una rama escribe un archivo con un SHA (el nombre de 40 caracteres de un commit): no copia ningún archivo del proyecto. Recién creada, apunta al commit donde estás (HEAD, el "estás aquí" de git); si estás en main, al mismo que main.

`git switch --detach` *ref*

Pone en disco una foto vieja sin mover ninguna rama: quedas en **detached HEAD** (HEAD apunta a un commit, no a una rama). Sirve para mirar o probar; `git switch -` te devuelve a tu rama.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
rama	nombre de la rama	`main`, `feat/gateway-timeout`
ref	cualquier nombre de un commit: HEAD, HEAD~1, una rama o un SHA	`HEAD`
rama	archivo en .git/refs/heads con el SHA del commit al que apunta (formato files, el default)	`wc -c .git/refs/heads/main → 41`

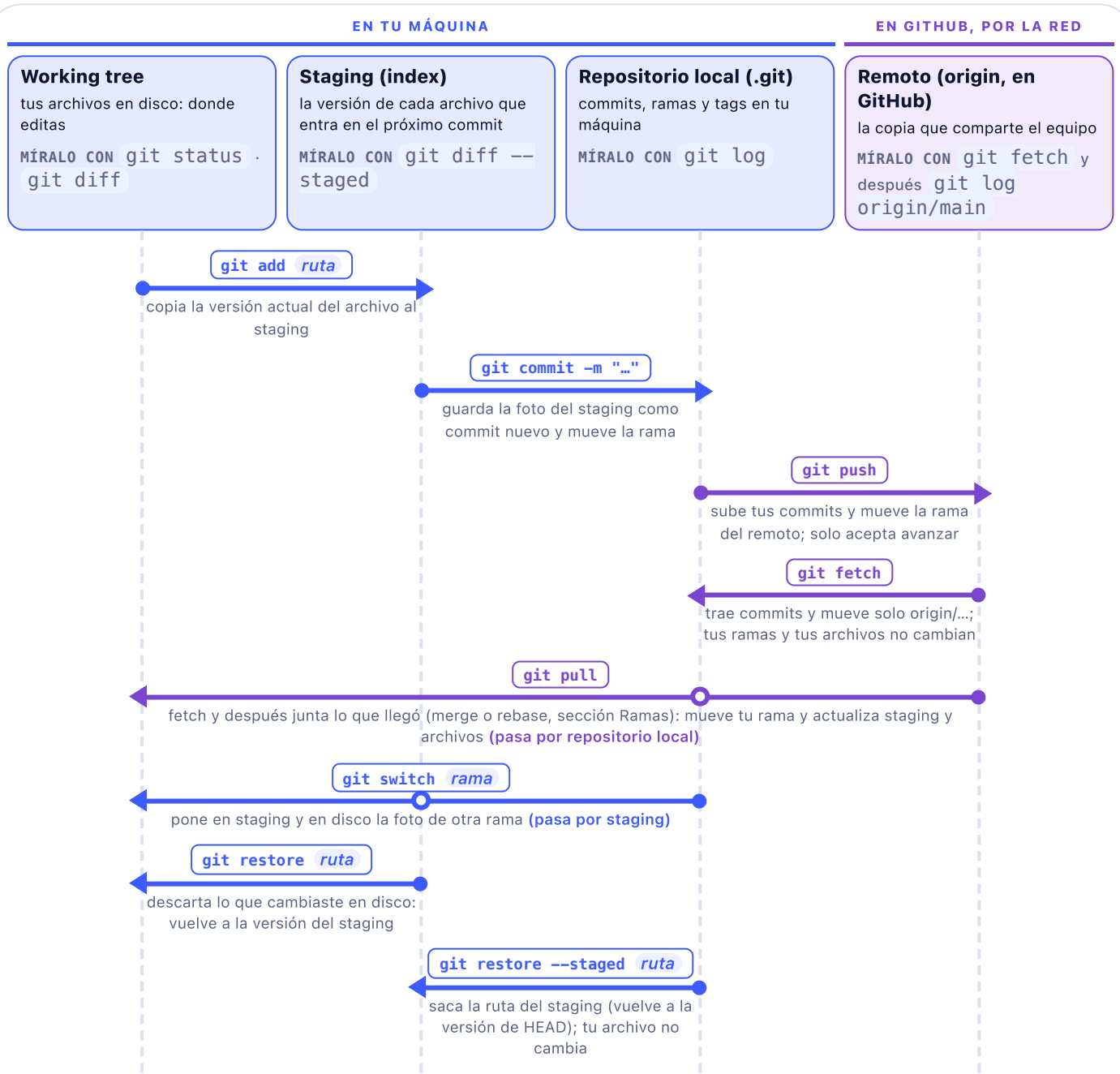
FÓRMULAS

$$\text{rama} = \text{SHA} + \backslash n = 40 + 1 = 41 \text{ bytes}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
repositorio (.git)	La carpeta oculta donde git guarda commits, ramas y configuración. Si la borras, borras la historia.	`~/lab-git/.git`
commit	Una foto de todos los archivos del proyecto en un momento, con un mensaje. Guarda la foto entera, no "el diff" (solo las líneas que cambiaron).	`gateway: bajar INFER_TIMEOUT_S a 120 ms`
tree (árbol)	La lista de carpetas y archivos de la foto, con el SHA del contenido de cada uno.	`040000 tree … gateway`
blob	El contenido de un archivo, guardado una sola vez. Si no cambia, el commit siguiente reusa el mismo.	`preprocess.py: el mismo blob 6d75488 en los dos commits`
parent (padre)	El commit anterior. Por eso la historia es una cadena que va hacia atrás.	`parent 9798182…`
SHA	El nombre del commit: un hash (una huella que git calcula) de su contenido. Si cambia cualquier cosa (un archivo, el mensaje, el padre), cambia el SHA.	`3387071`
rama (branch)	Una etiqueta con el SHA de un commit. Cada commit que haces estando en ella la mueve hacia adelante. No es una copia del proyecto.	`.git/refs/heads/feat/gateway-timeout`
HEAD	El "estás aquí": casi siempre apunta a una rama, y la rama a un commit.	`ref: refs/heads/main`
detached HEAD	HEAD apunta directo a un commit. Si commiteas ahí, ninguna rama guarda esos commits.	`HEAD detached at 315a4b2`

PIEZA	QUÉ HACE	EJEMPLO
working tree	Tus archivos en disco, los que editas. Cambiar de rama los reescribe.	`gateway/app/main.py`
staging (index)	La zona intermedia: la versión de cada archivo que entrará en el próximo commit. Te deja commitear solo una parte de lo que cambiaste.	`git add gateway/app/main.py`
remoto (origin)	Otra copia del repo, casi siempre en GitHub. origin es su nombre corto.	`git remote -v`
origin/main	Rama de seguimiento: tu copia local de dónde estaba main en el remoto la última vez que hablaste con él.	`git log origin/main`
git switch · git restore	Desde Git 2.23 se reparten los dos trabajos del viejo git checkout: switch cambia de rama, restore recupera archivos.	`git checkout main = git switch main`



Lo que no commiteaste no es de ninguna rama: `git switch` lo lleva contigo si no choca con la otra rama, y se niega si lo pisaría (`Your local changes to the following files would be overwritten by checkout`).

Las cuatro zonas y los comandos que mueven cambios entre ellas. add y commit van hacia la derecha; restore, hacia atrás; push, fetch y pull cruzan a la red. Las tres primeras viven en tu máquina.

EJEMPLO REAL · UN COMMIT DEL GATEWAY POR DENTRO: FOTO, PADRE Y ETIQUETA

TEXTO sesión real con git 2.50.1 sobre una copia de reference/c01/gateway, en la rama del cambio de timeout

```
$ cat .git/HEAD
ref: refs/heads/feat/gateway-timeout
$ cat .git/refs/heads/feat/gateway-timeout
338707141fa107b7131a8c16626d6cc4e9ff1305
$ git cat-file -p HEAD
tree d6cd4c098b8ee7ac214109095879486b26e545e8
parent 979818285678056fc3d1861d7d424c2a27aae823
author Ana Torres <ana@example.com> 1792645892 -0600
committer Ana Torres <ana@example.com> 1792645892 -0600

gateway: bajar INFER_TIMEOUT_S a 120 ms
$ git cat-file -p HEAD:gateway/app
100644 blob 5123940f72bde62371315600b516f9e0efcff01c    __init__.py
100644 blob 97f6b302a8f484d4a308fa1895a73a498bc9ed63    labels.json
100644 blob a3820d9939b821509addc20339c44b35dee89268    main.py
100644 blob 6d75488cb6c744ecde54335cae4b411181ebd5bc    preprocess.py
$ git cat-file -p HEAD~1:gateway/app
100644 blob 5123940f72bde62371315600b516f9e0efcff01c    __init__.py
100644 blob 97f6b302a8f484d4a308fa1895a73a498bc9ed63    labels.json
100644 blob e2709cc8a36ffc7b1ccd456f8fb2d857e646bbcb    main.py
100644 blob 6d75488cb6c744ecde54335cae4b411181ebd5bc    preprocess.py
```

- `.git/HEAD` no tiene un SHA: dice en qué rama estás. Y la rama es un archivo con un SHA (`3387071…`). Eso es todo lo que es una rama.
- El commit apunta a un `tree` (la foto del proyecto) y a su `parent` (9798182, el commit anterior). El primer commit de un repo no tiene parent; un merge tiene dos.
- Compara los dos listados: solo `main.py` tiene otro blob (a3820d9 contra e2709cc). `labels.json` y `preprocess.py` reusan el mismo: la foto es completa, pero lo que no cambió no se copia.
- El SHA es el hash de todo esto, con un encabezado `commit 244` (el tamaño). Cambia el mensaje, la fecha o el padre y el SHA cambia: por eso amend (rehacer el último commit) y rebase (copiar commits sobre otra base) crean commits nuevos.
- Si no encuentras el archivo de una rama, git la guardó en `.git/packed-refs` (pasa con las ramas remotas tras un clone, o tras un `git gc` , la limpieza que git corre de vez en cuando). `git rev-parse {rama}` funciona siempre.

CÓMO RECORDARLA

Un **commit es una foto** del proyecto, con su padre y un mensaje; una **rama es una etiqueta** que avanza sola con cada commit que haces en ella; **HEAD es el "estás aquí"**. Entre tu disco y la historia está el **staging**: ahí eliges qué entra en la foto.

TRAMPA

Pensar que una rama es una copia del proyecto: crear una escribe 41 bytes, y cambiar de rama reescribe tus archivos en disco. Lo que no commiteaste no es de ninguna rama: viaja contigo al cambiar, y git se niega si lo pisaría. Y creer que un commit guarda "el diff": guarda la foto entera (lo que no cambió se reusa, no se copia).

En el curso: M10 · Reproducibilidad: los cinco ingredientes · M9 · Versionado de datos y snapshots inmutables · M5 · Registries y tags inmutables

2 El día a día: clone, status, add, commit, diff, log, push, pull, fetch y .gitignore

¿Qué comandos usas todos los días, qué toca cada uno (tus archivos, tu repo o el remoto) y qué haces cuando push o pull se niegan?

SINTAXIS CON VARIABLES

`git clone` *url*
`cd` *carpeta*

Copia el repo con toda su historia a una carpeta nueva, registra el remoto (la copia en GitHub) como `origin` y te deja en la rama por defecto (casi siempre `main`), conectada con `origin/main`: tu copia local del main de GitHub. La URL puede ser SSH o HTTPS, dos formas de conectarte; cómo autenticarte está en la sección GitHub.

`git status`
`git diff`
`git add` *ruta*
`git diff --staged`
`git commit -m "` *mensaje* **"**

El ciclo de cada commit (cada foto que guardas): mira qué cambió, revisa el diff (las líneas que cambiaron), elige con `add` y revisa lo elegido antes de guardar. `git add -p` te deja elegir pedazos de un archivo.

`git fetch`
`git status -sb`
`git pull --rebase`
`git push`

Sincronizar. `fetch` baja lo nuevo y solo mueve `origin/…`: tus archivos no cambian. `pull` = fetch + juntar con tu rama; `--rebase` pone tus commits locales encima de los del remoto, sin un commit de merge (uno extra que junta las dos historias). `push` sube lo tuyo; la primera vez de una rama, `git push -u origin` *rama*.

`git config --global pull.ff only`

Decide una vez qué hace `git pull` cuando las ramas **divergieron** (cada una tiene commits que la otra no). Con `pull.ff only` solo avanza si no hay nada que mezclar y, si divergieron, se niega y eliges tú con `git pull --rebase`. Sin ninguna config también se niega (desde Git 2.33.1): `fatal: Need to specify how to reconcile divergent branches.`

`git check-ignore -v` *ruta*
`git rm --cached` *ruta*

`check-ignore -v` dice qué línea del `.gitignore` (la lista de patrones que git ignora) ignora un archivo. El `.gitignore` no afecta archivos que git ya sigue: para dejar de seguir uno sin borrarlo del disco, `git rm --cached` y un commit.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
url	dirección del repo	`git@github.com:equipo/classifier.git (ilustrativo)`
carpeta	la carpeta que crea clone: el nombre del repo sin .git	`classifier`
ruta	archivo o carpeta	`gateway/app/main.py`
mensaje	qué cambió, en una línea	`preprocess: comentar RESIZE_SHORT`
rama	tu rama, la primera vez que la subes	`feat/gateway-timeout`
A..B	los commits que están en B y no en A	`HEAD..origin/main = lo del remoto que todavía no tienes (HEAD es donde estás)`
A...B	en diff: lo que cambió B desde su merge-base con A (el último commit que tienen en común)	`git diff main...feat/gateway-timeout = el diff del PR (pull request)`

FÓRMULAS

$$\text{git log A..B} = \{\text{commits de B}\} \setminus \{\text{commits de A}\} \qquad \text{git diff A...B} = \text{diff}\big(\text{merge-base}(A, B),\ B\big)$$

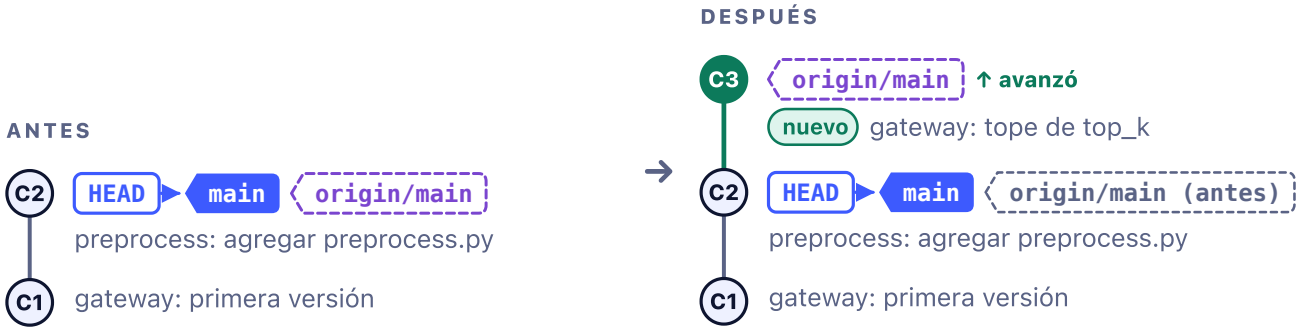
LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`git clone`	Copia el repo entero, con toda la historia, y crea origin.	`git clone git@github.com:equipo/classifier.git`
`git status`	Qué cambió, qué está en staging y cómo vas contra tu foto del remoto. No le pregunta nada al remoto.	`git status -sb → ## main...origin/main [behind 1]`

PIEZA	QUÉ HACE	EJEMPLO
git add	Copia la versión actual de un archivo al staging. -p elige pedazos; un punto agrega todo lo de la carpeta (mira antes).	`git add -p gateway/app/main.py`
git commit -m	Guarda la foto del staging con un mensaje. -a suma los cambios de archivos que git ya seguía, no los nuevos.	`git commit -m "gateway: tope de top_k"`
git diff	Lo que cambiaste en disco y todavía no está en staging.	`-INFER_TIMEOUT_S = 0.15 / +INFER_TIMEOUT_S = 0.12`
git diff --staged	Lo que está en staging contra el último commit: exactamente lo que vas a commitear.	`git diff --staged --stat`
git log	La historia. --oneline: una línea por commit; --graph: dibuja las ramas; -p: con el diff de cada uno.	`git log --oneline --graph -n 10`
git fetch	Baja commits y ramas del remoto y actualiza origin/...; no toca tus ramas ni tus archivos.	`main -> origin/main`
git pull	fetch + juntar con tu rama lo que llegó a su rama remota (upstream): merge, rebase o solo avanzar, según tu config.	`Updating 9798182..27bd258 · Fast-forward`
fast-forward	Avanzar la etiqueta de tu rama hasta el commit del remoto, sin crear nada: no había nada que mezclar.	`lo normal en main, si nunca commiteas ahí`
git push	Sube tus commits y mueve la rama del remoto. Solo acepta avanzar: si el remoto tiene algo que tú no, lo rechaza.	`! [rejected] … (non-fast-forward)`
upstream (-u)	La rama remota a la que está conectada la tuya. La usan pull, push y status.	`git push -u origin feat/gateway-timeout`
.gitignore	Patrones de archivos que git no ofrece agregar: venvs, caches, secretos, pesos.	`*.onnx · .env · __pycache__/`
git rm --cached	Deja de seguir un archivo sin borrarlo del disco; el .gitignore evita que vuelva a entrar.	`rm 'notas.log'`

1 fetch: solo se mueve origin/main

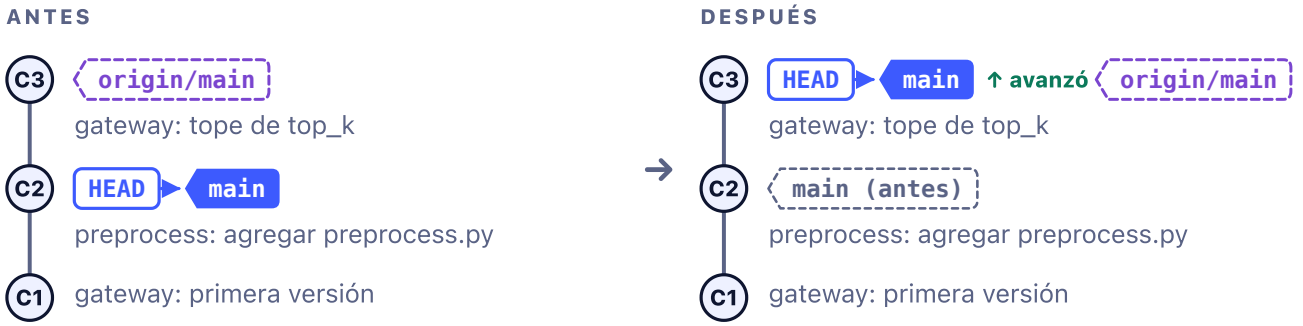
`git fetch`



C3 lo subió otra persona. fetch lo trae y mueve origin/main; main y tus archivos siguen en C2. git status -sb ahora dice [behind 1].

2 pull sin divergencia: fast-forward

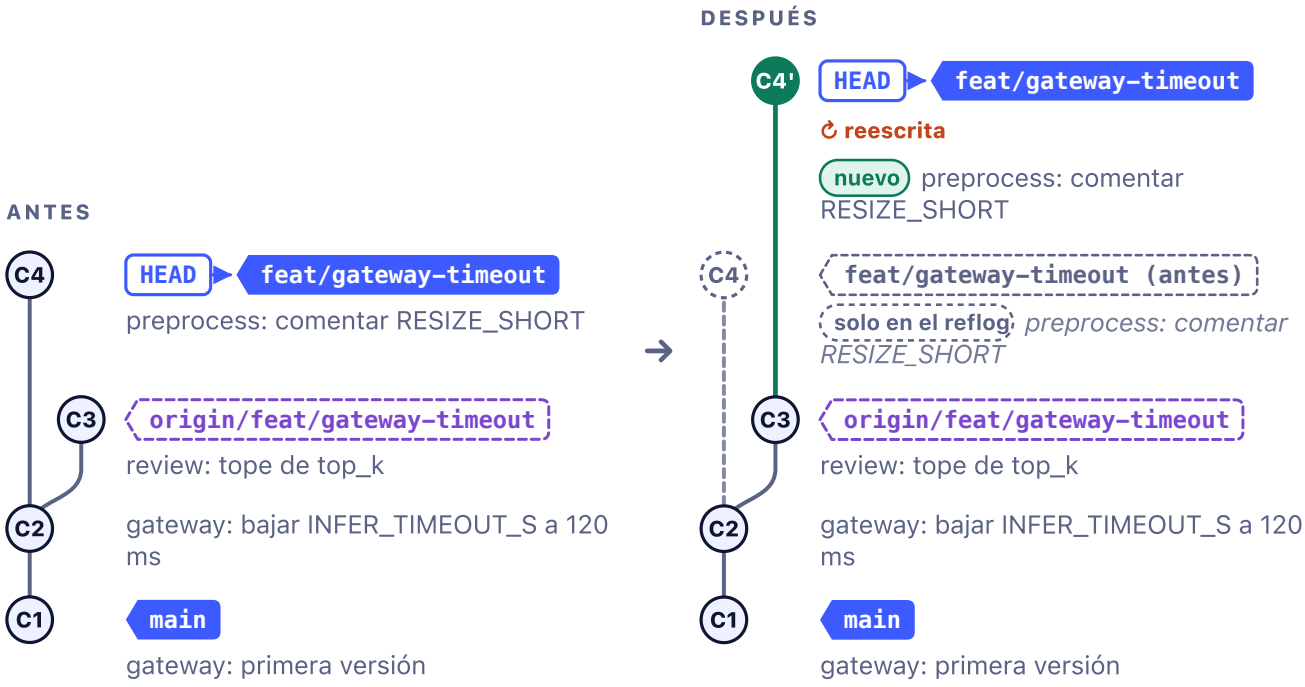
`git pull`



No había nada que mezclar: la etiqueta main avanza hasta C3 y tus archivos se actualizan. Ningún commit nuevo.

3 ramas divergentes en tu rama de PR

`git pull --rebase`



C3 (la sugerencia de la review) y tu C4 nacieron de C2: divergieron, y push se rechaza (non-fast-forward). --rebase copia C4 encima de C3 como C4', con otro SHA (otro nombre de commit); C4 queda solo en el reflog (el historial local de HEAD).



Arriba, lo más nuevo; cada línea une un commit con su padre.

commit · commit nuevo · solo en el reflog · rama local · rama remota (origin/...) · HEAD estás aquí · dónde estaba antes

Tu repo antes y después de cada comando. Las etiquetas origin/... son tu foto del remoto: solo se mueven con fetch, pull o push. Los dos primeros pasos ocurren en main; los dos últimos, en tu rama de PR cuando alguien empujó una sugerencia de la review.

EJEMPLO REAL · EL .GITIGNORE DE UN REPO CON LA ESTRUCTURA DEL C01

TEXTO — ilustrativo: armado con las reglas de reference/c01/gateway/.dockerignore y del README del C01 (model.plan y classifier.onnx no se versionan)

```
# .gitignore en la raíz del repo

# Python: se regenera solo, y un venv de macOS no sirve en la máquina Linux de CI
__pycache__/
*.py[cod]
.pytest_cache/
.venv/

# Secretos locales: nunca en git (el historial es para siempre)
.env
*.pem

# Pesos y engines: los generan los scripts de model/ y viven fuera de git
*.pt
*.onnx
*.plan
```

- Una barra al final (`__pycache__/`) solo ignora carpetas. Un patrón sin barra en medio vale en cualquier subcarpeta: `*.onnx` ignora también `model/classifier.onnx` .
- `.env` y `*.pem` : un secreto commiteado queda en la historia aunque lo borres en el commit siguiente. Si pasó, rótalo primero: crea uno nuevo e invalida el viejo (M15).
- Pesos y engines no van a git: GitHub avisa desde 50 MiB y bloquea archivos de más de 100 MiB. En el C01, el engine (el modelo compilado para la GPU) se construye en la T4 y viaja dentro de la imagen de Triton (el servidor del modelo); los datos se versionan con snapshots (M9) y los modelos, en el registry de modelos (M10).
- El `.gitignore` no saca archivos que git ya sigue: para eso, `git rm --cached {ruta}` y un commit. Lo de tu máquina (`.DS_Store` , `.vscode/`) va mejor en tu ignore global, `~/.config/git/ignore` .
- `git status -s --ignored` marca con `!!` lo que el .gitignore esconde (sin `-s` , lo lista bajo `Ignored files`), y `git check-ignore -v {ruta}` dice qué línea lo ignora.

CÓMO RECORDARLA

fetch mira, pull trae, push entrega. fetch solo mueve `origin/…` ; pull además mueve tu rama y tus archivos; push solo acepta avanzar. Antes de cada commit: `git status` y `git diff --staged` .

TRAMPA

`git add .` sin mirar: entran el `.env` , un `classifier.onnx` de cientos de MB o un `__pycache__` . Revisa `git status` y `git diff --staged` antes de cada commit. Y cuando un push se rechaza, no fuerces: `git fetch` , mira qué llegó y junta con `git pull --rebase` .

En el curso: M9 · Versionado de datos y snapshots inmutables | M15 · Secretos y credenciales | M10 · CI/CD para modelos: la pirámide de tests

3 Ramas, merge y rebase; conflictos y cómo resolverlos

¿Cómo trabajas en paralelo sin pisarte con el equipo, cómo juntas las ramas y qué haces cuando git marca un conflicto?

SINTAXIS CON VARIABLES

`git switch -c` *rama*
`git switch` *rama*
`git branch -d` *rama*

Crear y cambiarte (`-c`), solo cambiarte, y borrar una rama ya mergeada (unida a otra). `-d` se niega si la rama tiene commits que no están ni en la rama donde estás ni en su rama remota (`is not fully merged`); `-D` la borra igual.

`git merge` *rama*
`git merge --no-ff --no-edit` *rama*

Junta otra rama con la tuya. Si tu rama no avanzó, git solo mueve la etiqueta (**fast-forward**); si las dos avanzaron, crea un **merge commit** con dos padres. `--no-ff` fuerza el merge commit y `--no-edit` acepta el mensaje sin abrir el editor.

`git rebase` *base*
`git push --force-with-lease`

Copia tus commits encima de la punta de *base*, con SHAs nuevos (otros nombres de commit): la historia queda en línea recta. Si la rama ya estaba en el remoto, el push normal se rechaza; `--force-with-lease` la reemplaza solo si nadie empujó desde tu último fetch. Nunca en main.

`git status`
`git add` *ruta*
`git commit --no-edit`
`git merge --abort`

Un **conflicto**: las dos ramas cambiaron distinto las mismas líneas (o líneas vecinas) y git no elige. status los marca `both modified`; editas el archivo, quitas las marcas, `git add` y commit. `--abort` te deja como antes del merge.

`git rebase --continue`
`git rebase --abort`

En un rebase, el conflicto se resuelve igual (editar y `git add`), pero sigues con `--continue`, que abre el editor con el mensaje del commit. `GIT_EDITOR=true git rebase --continue` lo acepta tal cual.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
rama	rama a crear, cambiar, juntar o borrar	`feat/gateway-timeout`
base	la rama sobre la que pones tus commits	`main (u origin/main, recién traída con fetch)`
ruta	archivo en conflicto	`gateway/app/main.py`
base	el último commit que las dos ramas tienen en común (merge-base)	`git merge-base main feat/gateway-timeout`
$\text{cambio}_{\text{tuyo}}$	lo que cambió tu rama desde la base	`0.15 → 0.12 en INFER_TIMEOUT_S`
$\text{cambio}_{\text{suyo}}$	lo que cambió la otra rama desde la base	`0.15 → 0.18: un hotfix (arreglo urgente) que ya entró a main`
resultado	el archivo combinado; si los cambios no chocan, git lo arma solo	`el main.py del merge commit`

FÓRMULAS

$$\text{resultado} = \text{base} + \text{cambio}_{\text{tuyo}} + \text{cambio}_{\text{suyo}}$$

conflicto: los dos cambian distinto las mismas líneas (o vecinas)

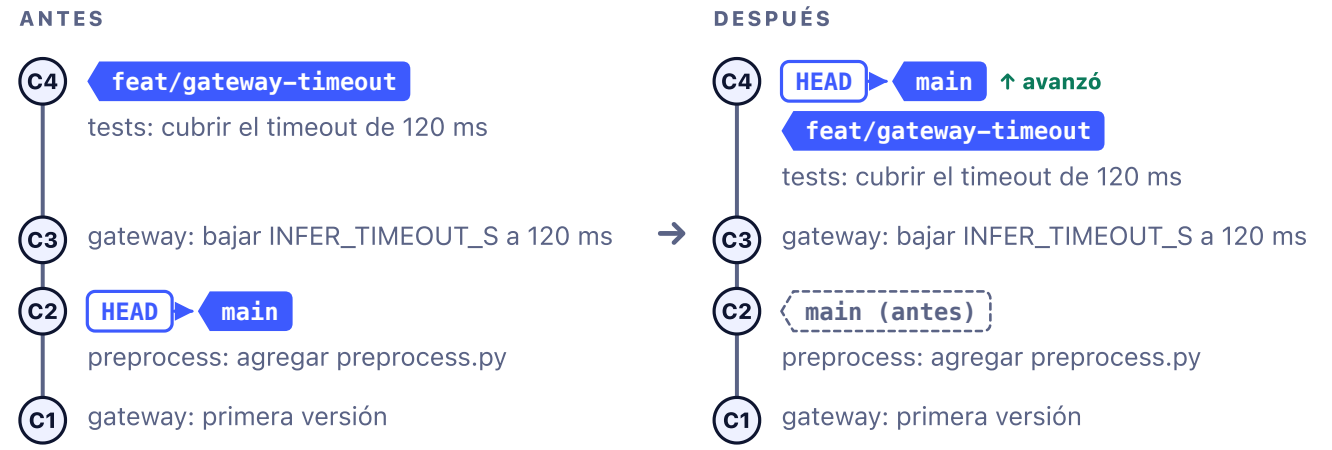
LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`rama (branch)`	Etiqueta que avanza con cada commit que haces en ella. Trabajar en una rama no toca a las demás.	`feat/gateway-timeout`
`fast-forward`	Juntar sin commit nuevo: la etiqueta solo avanza, porque la otra rama ya contenía todo lo tuyo.	`Updating be403cc..d73cbdd · Fast-forward`
`merge commit`	Commit con dos padres que junta dos historias que avanzaron por separado.	`Merge made by the 'ort' strategy.`
`merge-base`	El último commit en común. Git compara cada lado contra él (three-way merge).	`git merge-base main HEAD`
`rebase`	Rehace tus commits, uno por uno, encima de otra base. Cada uno sale con un SHA nuevo.	`Successfully rebased and updated refs/heads/feat/gateway-timeout.`

PIEZA	QUÉ HACE	EJEMPLO
conflicto (conflict)	Las dos ramas cambiaron distinto las mismas líneas, o líneas vecinas. Git para y te pide decidir.	`CONFLICT (content): Merge conflict in gateway/app/main.py`
marcas de conflicto	<<<<<<< abre tu lado, ======= separa, >>>>>>> cierra el otro lado. Al resolver, las borras.	`<<<<<<< HEAD … >>>>>>> main`
UU · both modified	Archivo en conflicto sin resolver. Pasa a resuelto cuando haces git add.	`UU gateway/app/main.py`
git merge --abort · git rebase --abort	Cancelan y te dejan como antes de empezar.	`cuando el conflicto es más grande de lo que esperabas`
ours / theirs	En un merge, ours es tu rama y theirs la que entra. En un rebase se invierten: ours es la base y theirs, tu commit.	`git restore --theirs gateway/app/main.py`
git diff --check	Avisa si quedaron marcas de conflicto o espacios sobrantes. Pásalo antes del git add.	`gateway/app/main.py:38: leftover conflict marker`
merge.conflictStyle zdiff3	Agrega a las marcas la versión de la base, tras |||||||: ves qué había antes de los dos cambios. Existe desde Git 2.35.	`git config --global merge.conflictStyle zdiff3`
git push --force-with-lease	Reemplaza la rama remota solo si sigue donde la viste en tu último fetch. Para tu rama de PR (pull request), nunca para main.	`+ 941be23...8707e88 feat/gateway-timeout -> feat/gateway-timeout (forced update)`
git branch -d · -D	Borra una rama local. -d solo si sus commits ya están en la rama donde estás o en su rama remota; -D, aunque no lo estén.	`error: the branch 'feat/x' is not fully merged`

1 Fast-forward: main no tenía commits nuevos

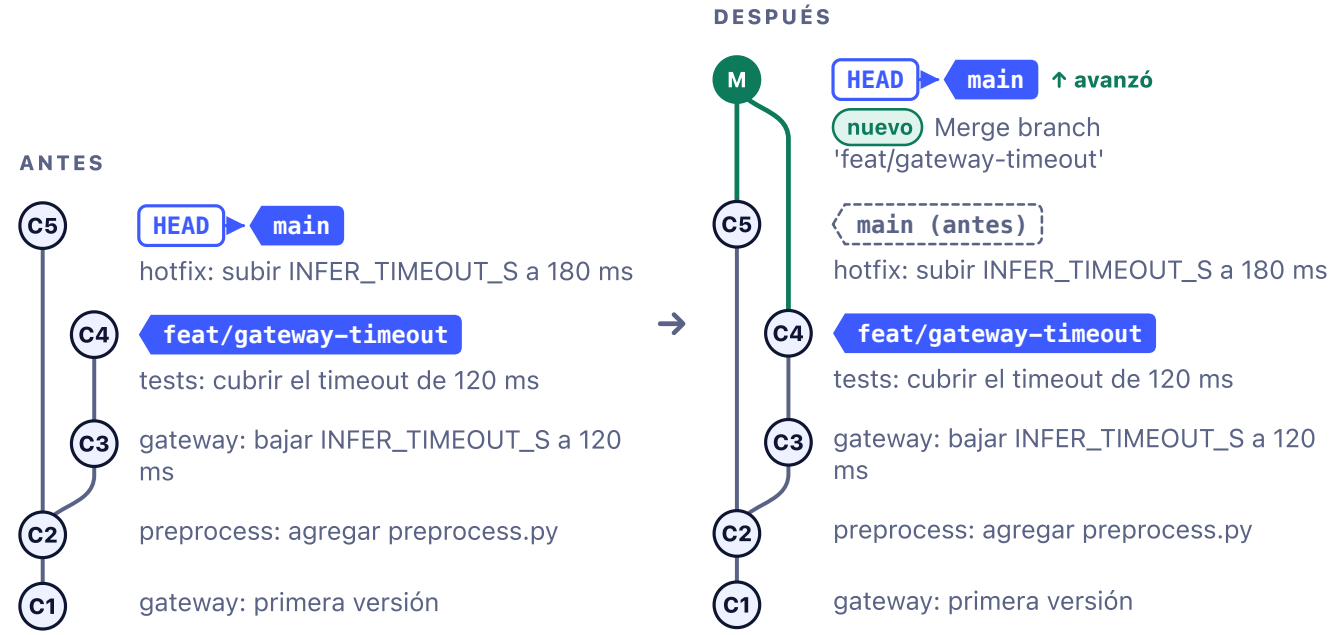
`git merge feat/gateway-timeout`



main es ancestro de la rama: git solo avanza la etiqueta. No hay commit nuevo y la historia queda en línea recta.

2 Merge commit: las dos ramas avanzaron

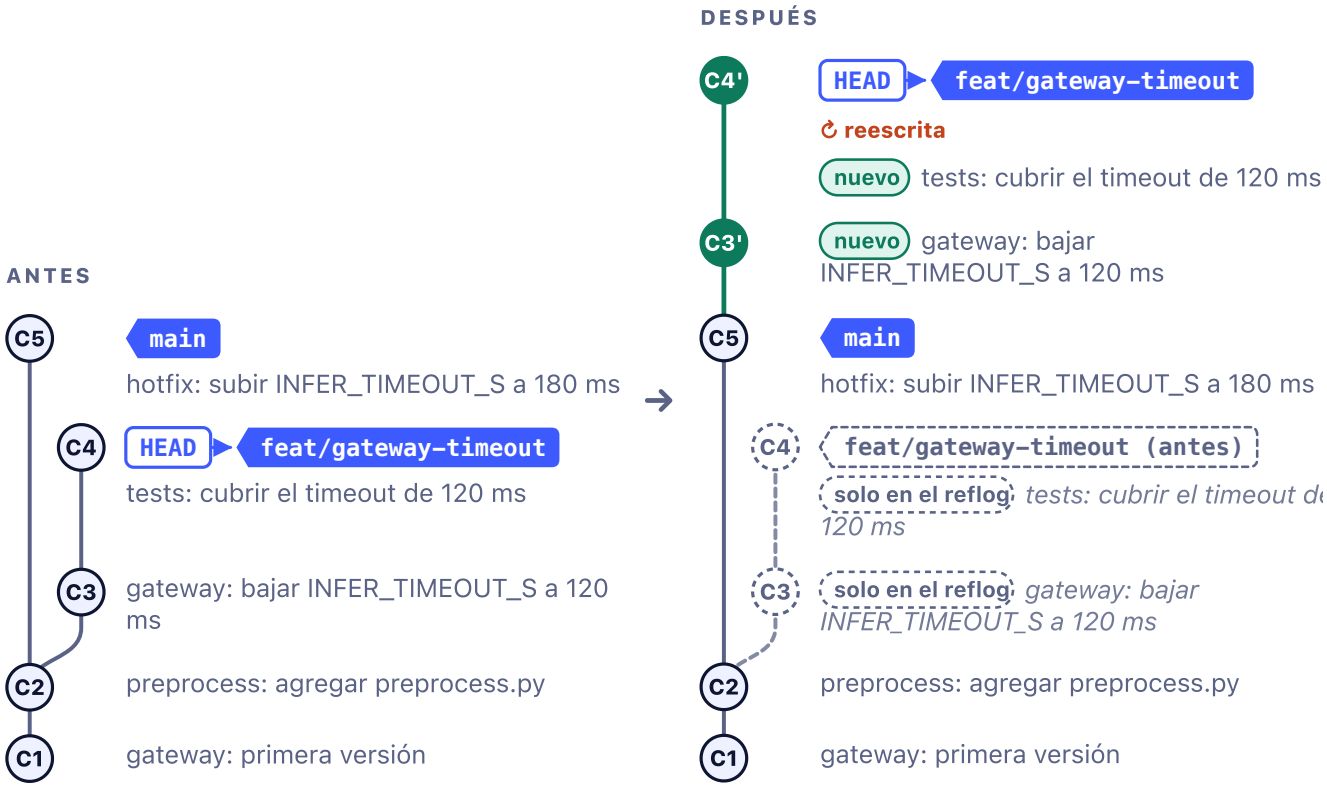
`git merge feat/gateway-timeout`



M tiene dos padres: C5 (el primero, el main de antes) y C4 (la rama). Si C3 o C4 tocaron las mismas líneas que C5, aquí aparece el conflicto.

3 Rebase, parado en la rama: tus commits, copiados encima

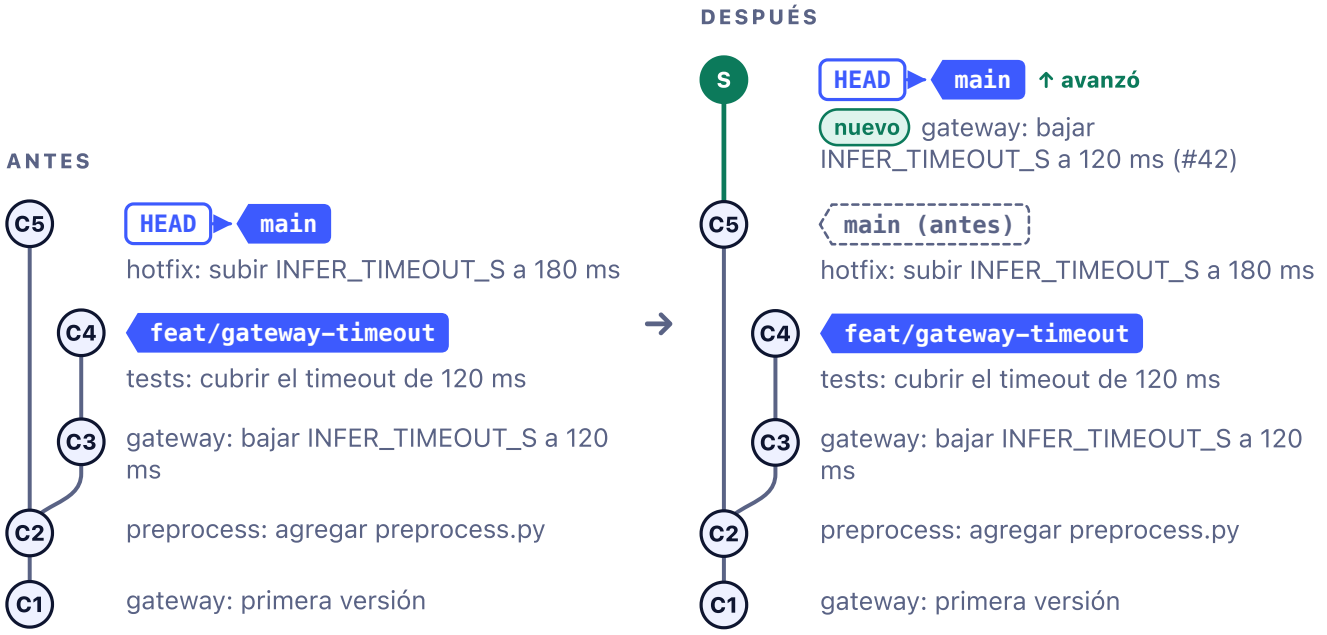
`git rebase main`



C3' y C4' traen los mismos cambios con SHAs nuevos; C3 y C4 quedan solo en el reflog. Ahora juntar la rama con main es un fast-forward. Si la rama ya estaba en el remoto: git push --force-with-lease.

4 Squash: un solo commit nuevo en main

Squash and merge en GitHub (≈ git merge --squash y git commit)



S trae juntos los cambios de C3 y C4, con un solo padre. Los commits de la rama no entran a main: bórrala después del merge. El tag de la imagen sale del SHA de S.

Arriba, lo más nuevo; cada línea une un commit con su padre.

commit · commit nuevo · solo en el reflog · rama local · rama remota (origin/...) · HEAD estás aquí · dónde estaba antes

Cuatro formas de juntar feat/gateway-timeout con main. Cada nodo es un commit y cada etiqueta, una rama; los commits que quedan fuera de toda rama solo siguen en el reflog (el historial local de HEAD).

EJEMPLO REAL · UN CONFLICTO REAL EN GATEWAY/APP/MAIN.PY, CON LAS MARCAS ZDIFF3

TEXTO sesión real con git 2.50.1 sobre una copia de reference/c01/gateway: tu rama baja el timeout y un hotfix ya mergeado en main lo sube

```
$ git config --global merge.conflictStyle zdiff3
$ git merge main
Auto-merging gateway/app/main.py
CONFLICT (content): Merge conflict in gateway/app/main.py
Automatic merge failed; fix conflicts and then commit the result.
$ git status -s
UU gateway/app/main.py
$ sed -n 38,44p gateway/app/main.py
<<<<<<< HEAD
INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.12"))  # 120 ms: el resto del p99 ≤ 200 ms queda para el preprocess
||||||| 0c52e4b
INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.15"))  # 150 ms para Triton dentro del p99 ≤ 200 ms
=======
INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.18"))  # hotfix: 504 en el pico con 150 ms
>>>>>>> main
# … dejas una sola línea y borras las marcas …
$ git diff --check
$ git add gateway/app/main.py
$ git commit --no-edit
[feat/gateway-timeout d7e8c0c] Merge branch 'main' into feat/gateway-timeout
```

- Estás en `feat/gateway-timeout` y traes main a tu rama para resolver el conflicto antes de que lo vea la review. Git juntó solo todo lo demás; paró en la única línea que tocaron los dos.
- Entre `<<<<<<< HEAD` y `|||||||` está tu lado; después de `|||||||`, la **base** (0c52e4b, el último commit en común); entre `=======` y `>>>>>>> main`, el otro lado. Sin zdiff3, la parte de la base no aparece.
- Resolver no es elegir "el mío": aquí los dos cambios responden a problemas distintos (el p99 y los 504 del pico). Decide con quien hizo el hotfix y escribe la línea final, que puede no ser ninguna de las dos.
- `git diff --check` sin salida significa que no quedaron marcas. Git acepta un `git add` con marcas adentro, así que este chequeo es tuyo.
- El merge commit nuevo tiene dos padres. Con `git push`, el PR se actualiza y el CI (las pruebas automáticas) vuelve a correr sobre el resultado.

CÓMO RECORDARLA

Merge junta, rebase reescribe. Merge agrega un commit con dos padres (o solo avanza la etiqueta: fast-forward); rebase copia tus commits encima de otra base, con SHAs nuevos. Regla de oro: no hagas rebase de commits que otras personas ya tienen.

TRAMPA

En un rebase, ours y theirs se invierten: `HEAD` (ours) es la base y el otro lado es **tu** commit, así que `git restore --theirs` en medio de un rebase se queda con lo tuyo. Y después de un rebase de tu rama ya subida, el push rechazado sugiere `git pull`: eso mezclaría la versión vieja. Lo que va es `git push --force-with-lease`.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M10 · El validation gate: cuándo un modelo puede reemplazar al actual · C01 · Archivos de referencia

4 Deshacer: restore, reset, revert, amend, stash y reflog

¿Cómo deshaces cada tipo de error, de un archivo sin guardar a un commit que ya está en main, y cómo recuperas un commit que creías perdido?

SINTAXIS CON VARIABLES

`git restore` *ruta*

`git restore --staged` *ruta*

Para archivos. Sin opciones, descarta lo que cambiaste en disco (vuelve a la versión del staging: la de tu último `git add` o, si no hiciste add, la del último commit): **no tiene deshacer**. Con `--staged`, saca el archivo del staging y tu disco no cambia. Reemplazan a `git checkout --` *ruta* y a `git reset` *ruta*.

`git commit --amend --no-edit`

Rehace el último commit con lo que hay en staging (con `-m "…"`, también cambia el mensaje). Sale un commit nuevo, con otro SHA: úsalo solo si no hiciste push.

`git reset` *modo* **`HEAD~`** *n*

Mueve la rama *n* commits atrás. `--soft`: los cambios quedan en staging; `--mixed` (el default): quedan en disco, fuera del staging; `--hard`: se borran de staging y disco, junto con lo que no habías commiteado. Los commits siguen en el reflog (el historial local de HEAD).

`git revert` *sha*

`git revert -m 1` *sha_merge*

Crea un commit nuevo que deshace otro. No reescribe nada: es el camino después de un push, y el rollback (volver a la versión anterior) por PR en main. Para revertir un merge commit, `-m 1` dice que la línea principal es su primer padre.

`git stash push -u -m "`*nota***`"`**

`git stash list`

`git stash pop`

Guarda tus cambios sin commit en una pila (el **stash**) y deja tu disco limpio. `-u` incluye también los archivos nuevos que git todavía no sigue (untracked); sin él, quedan afuera. `pop` los devuelve y borra la entrada; si hay conflicto, la entrada se queda.

`git reflog`

`git branch` *rescate* **`HEAD@{`***n***`}`**

El **reflog** lista dónde estuvo HEAD. Un commit "perdido" tras un reset o un rebase sigue ahí (al menos 30 días por defecto): ponle una rama encima y queda a salvo.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ruta	archivo a recuperar o a sacar del staging	`gateway/app/main.py`
modo	--soft, --mixed (el default) o --hard	`--soft para rehacer el último commit`
n	cuántos commits atrás (HEAD~n) o cuántos movimientos atrás en el reflog (HEAD@{n})	`1`
sha	SHA del commit a revertir; el corto alcanza	`c5aa999`
sha_merge	SHA de un merge commit, como el que deja el botón de un PR	`cd07ebf: Merge pull request #42`
nota	para qué guardaste esos cambios	`probar 100 ms`
rescate	nombre de la rama nueva que apunta al commit recuperado	`rescate`
t_{rescate}	tiempo mínimo para recuperar un commit que quedó fuera de toda rama: lo que dura su línea en el reflog (gc.reflogExpireUnreachable)	`30 días por defecto; después, git gc (la limpieza automática de git) puede borrarlo`

FÓRMULAS

$t_{\text{rescate}} \geq 30$ días (commit que quedó fuera de toda rama, con la config por defecto)

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`git restore` *ruta*	Descarta los cambios en disco de ese archivo. No queda copia en ningún lado.	`volver main.py a la versión del staging`
`git restore --staged` *ruta*	Saca el archivo del staging; en disco no cambia nada.	`deshacer un git add`
`git restore --source=`*ref* *ruta*	Trae un archivo como estaba en otro commit, sin mover la rama.	`git restore --source=HEAD~1 gateway/app/main.py`
`git commit --amend`	Reemplaza el último commit (mensaje o contenido) por uno nuevo, con otro SHA.	`corregir un typo en el mensaje antes del push`
`git reset --soft`	Mueve la rama atrás; staging y disco quedan igual: el cambio queda listo para recommitear.	`juntar los últimos 3 commits en uno`

PIEZA	QUÉ HACE	EJEMPLO
`git reset (--mixed)`	Mueve la rama atrás y saca los cambios del staging; tus archivos no cambian.	`Unstaged changes after reset:`
`git reset --hard`	Mueve la rama y pisa staging y disco. Borra también los archivos nuevos que estaban en staging.	`HEAD is now at b029a14 gateway: primera versión`
`ORIG_HEAD`	Dónde estaba HEAD antes del último reset, merge o rebase.	`git reset --hard ORIG_HEAD deshace un merge recién hecho`
`git revert`	Commit nuevo con el cambio inverso. La historia queda intacta: es seguro después de push.	`Revert "gateway: bajar INFER_TIMEOUT_S a 120 ms"`
`git revert -m 1`	Revierte un merge commit tomando como línea principal su primer padre (el main de antes del merge).	`This reverts commit cd07ebf…, reversing changes made to 1380e71…`
`git stash`	Guarda cambios sin commit y limpia el disco. Los archivos untracked (nuevos y sin git add), solo con -u.	`Saved working directory and index state On main: probar 100 ms`
`git reflog`	El diario local de HEAD: cada movimiento con su SHA. Ahí siguen los commits "perdidos".	`HEAD@{1}: commit: gateway: bajar INFER_TIMEOUT_S a 120 ms`
`git push --force-with-lease`	Sube una rama reescrita (amend, reset, rebase) solo si nadie la cambió desde tu último fetch. Solo para tu rama de PR.	`(forced update)`
`git clean -n · -f`	Lista (-n) o borra (-f) los archivos untracked. -f no tiene deshacer: mira primero con -n.	`Would remove notas.txt`

COMANDO	HEAD Y LA RAMA	STAGING	WORKING TREE	¿DESPUÉS DE PUSH?
`git restore` *ruta*	= no se mueve	= igual	✕ **la ruta vuelve a la versión del staging: lo que cambiaste sin git add se pierde**	✓ sí: no toca commits
`git restore --staged` *ruta*	= no se mueve	↻ la ruta vuelve a la versión de HEAD	= igual	✓ sí: no toca commits
`git commit --amend`	↻ la rama pasa a un commit nuevo, con otro SHA	↻ lo que había entra al commit nuevo	= igual	! no: reescribe un commit publicado
`git reset --soft HEAD~1`	↻ vuelve 1 commit	= igual: queda con los cambios del commit deshecho	= igual	! no: reescribe
`git reset HEAD~1 (--mixed)`	↻ vuelve 1 commit	↻ como el nuevo HEAD: los cambios salen del staging	= igual: los cambios quedan en disco	! no: reescribe
`git reset --hard HEAD~1`	↻ vuelve 1 commit	✕ **como el nuevo HEAD: lo que estaba en staging se pierde**	✕ **como el nuevo HEAD: lo que no era commit se pierde**	! no: reescribe
`git revert` *sha*	↻ avanza: commit nuevo con el cambio inverso	↻ como el nuevo HEAD (exige trabajo limpio antes)	↻ como el nuevo HEAD	✓ sí: es el camino en main
`git stash push -u`	= no se mueve	↻ se guarda en el stash y queda limpio	↻ se guarda en el stash y queda limpio	✓ sí: no toca commits
`git branch` *rescate* `HEAD@{` *n* `}`	↻ HEAD no se mueve; una rama nueva apunta al commit perdido	= igual	= igual	✓ sí: solo agrega una etiqueta

= no cambia

↻ cambia sin perder trabajo: lo anterior sigue en disco, en el reflog o en el stash

✕ se pierde: el reflog solo guarda commits

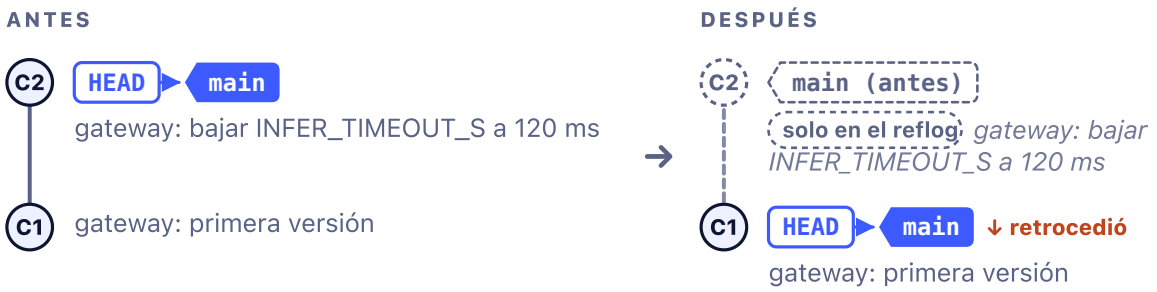
✓ seguro aunque ya hayas hecho push

! reescribe historia: solo en tu rama sin compartir, o en tu PR con push --force-with-lease

LOS MISMOS COMANDOS SOBRE EL GRAFO

1 reset: la rama vuelve atrás

`git reset {--soft | --mixed | --hard} HEAD~1`



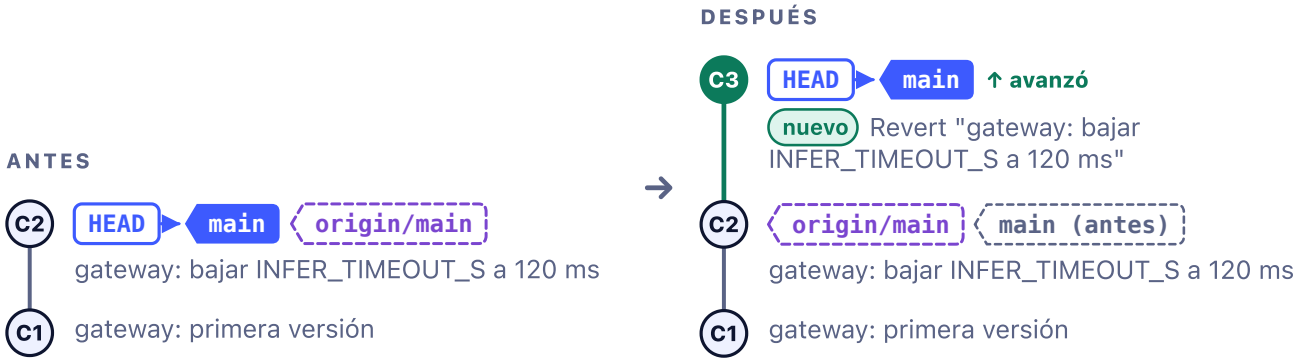
Los tres modos mueven la rama igual; cambia qué pasa con staging y con tus archivos

MODO	HEAD Y LA RAMA	STAGING	WORKING TREE
`--soft`	↺ vuelve 1 commit	= con los cambios de C2	= con los cambios de C2
`--mixed`	↺ vuelve 1 commit	↺ igual a C1	= con los cambios de C2
`--hard`	↺ vuelve 1 commit	✕ **igual a C1**	✕ **igual a C1: los cambios de C2 solo siguen en el commit C2**

El grafo es el mismo en los tres modos; cambia qué pasa con staging y disco. C2 no se borra: queda fuera de toda rama, en el reflog.

2 revert: un commit nuevo que deshace

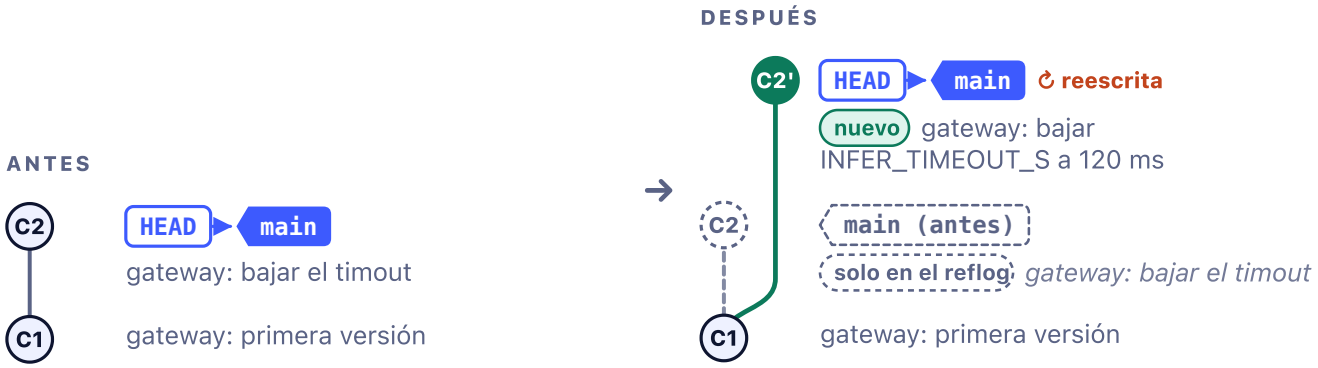
`git revert C2`



Nada se reescribe: para el remoto es avanzar, y acepta el push como el de cualquier commit. Es el rollback que deja rastro.

3 amend: reemplaza el último commit

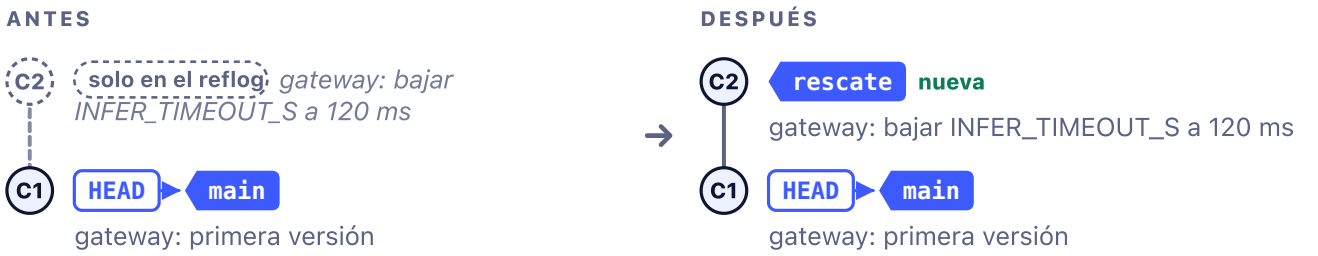
`git commit --amend`



C2' tiene el mismo padre y otro SHA. Si C2 ya estaba en el remoto, ahora tu rama y la remota divergen.

4 reflog: rescatar un commit

`git branch rescate HEAD@{1}`



Después de un reset --hard, C2 solo estaba en el reflog. Con una etiqueta encima vuelve a estar a salvo; luego git merge --ff-only rescate lo trae a main.

Arriba, lo más nuevo; cada línea une un commit con su padre.

commit · commit nuevo · solo en el reflog · rama local · rama remota (origin/...) · HEAD estás aquí · dónde estaba antes

Antes de deshacer, pregunta: **¿ya hice push?** (sí → `revert`) y **¿esto fue commit alguna vez?** (no → el reflog no lo tiene). Lo que llegó a estar en staging deja un blob suelto (una copia de su contenido) que `git fsck --lost-found` encuentra, sin nombre de archivo, hasta que `git gc` lo borre; lo que solo estuvo en disco no deja nada.

Qué mueve cada comando de deshacer y si sirve después de un push. Después de la tabla, los grafos de reset, revert, amend y el rescate con el reflog.

EJEMPLO REAL · REVERTIR EL PR #42 DEL GATEWAY, QUE YA ESTÁ EN MAIN

TEXTO sesión real con git 2.50.1 sobre una copia de reference/c01/gateway; el merge #42 imita el botón Create a merge commit de GitHub (… marca líneas omitidas)

```
$ git switch -c revert/gateway-timeout
Switched to a new branch 'revert/gateway-timeout'
$ git log --oneline --graph -3
*   cd07ebf Merge pull request #42 from equipo/feat/gateway-timeout
|\
| * c5aa999 gateway: bajar INFER_TIMEOUT_S a 120 ms
|/
* 1380e71 c01: gateway y manifests
$ git revert cd07ebf
error: commit cd07ebfa545a267cc125a8b5e05d7c53ec2910b9 is a merge but no -m option was given.
fatal: revert failed
$ git revert -m 1 --no-edit cd07ebf
[revert/gateway-timeout 3a36915] Revert "Merge pull request #42 from equipo/feat/gateway-timeout"
 Date: Wed Oct 21 23:12:58 2026 -0600
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git show -U0
…
    This reverts commit cd07ebfa545a267cc125a8b5e05d7c53ec2910b9, reversing
    changes made to 1380e71e91f3dbc3460c8ca2a0f6581533705ff6.
```

```
…
-INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.12"))  # 120 ms para Triton dentro del p99 ≤ 200 ms
+INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.15"))  # 150 ms para Triton dentro del p99 ≤ 200 ms
$ git push -u origin revert/gateway-timeout
```

- El revert también entra por una rama y un PR, con CI y review: en main no se commitea directo. En GitHub, el botón **Revert** de un PR mergeado abre ese PR por ti.
- Un merge commit tiene dos padres y git no sabe cuál es la línea principal: sin `-m` falla. `-m 1` elige el primer padre, el main de antes del merge (1380e71).
- El revert es un commit nuevo (3a36915) con el cambio inverso: `INFER_TIMEOUT_S` vuelve a 0.15. La historia queda intacta, así que no hace falta forzar nada.
- Cuando se mergee, CD (el workflow que construye y despliega) arma otra imagen, con otro tag y el código de antes. Si necesitas volver ya, el manifest (el YAML de Kubernetes) puede apuntar al tag anterior del gateway, que sigue en ECR, el registry de imágenes (sección CD).
- Ojo al volver a intentar: revertir un merge deshace sus cambios, pero sus commits siguen en la historia de main. Si arreglas la rama y la mergeas otra vez, git solo trae los commits nuevos y los viejos no vuelven a entrar: primero revierte el revert (`git revert 3a36915`).

CÓMO RECORDARLA

Antes de deshacer, pregunta: **¿ya hice push?** Si no: `restore` , `amend` , `reset` . Si sí: `revert` . Y si algo "se perdió": `git reflog` . Con la config por defecto, un commit no se pierde antes de 30 días; lo que nunca commiteaste, sí.

TRAMPA

`git reset --hard` y `git restore` no tienen red: lo que no estaba en un commit (ni en el stash) se pierde, y `--hard` borra también los archivos nuevos que estaban en staging. Y un revert no borra un secreto de la historia: el commit viejo sigue ahí. Rota el secreto primero: crea uno nuevo e invalida el viejo (M15).

En el curso: M11 · Rollback: criterios automáticos y qué guardar para volver · Docker y kubectl · Rollouts y escalado · M15 · Secretos y credenciales · M10 · CI/CD para modelos: la pirámide de tests

5 GitHub: remotos, acceso, pull requests y reglas de la rama

¿Cómo llega tu cambio a main en un repo de equipo, y qué lo frena si algo está mal?

SINTAXIS CON VARIABLES

`git remote add origin git@github.com:` *org* **`/`** *repo* **`.git`**
`git push -u origin` *rama*

Un **remoto** (remote) es otra copia del repo; aquí, la de GitHub. `origin` es solo su nombre por defecto. `-u` deja tu rama siguiendo a `origin/` *rama* : después alcanza con `git push` y `git pull` a secas.

`ssh-keygen -t ed25519 -C "` *email* **`"`**
`ssh -T git@github.com`

Crea una **llave SSH**: un par de archivos. El privado se queda en tu Mac; el `.pub` (público) lo agregas en GitHub → Settings → SSH and GPG keys. Si funciona, GitHub contesta `Hi` *usuario* `! You've successfully authenticated, but GitHub does not provide shell access.`

`gh pr create --base main --title "` *título* **`" --body "`** *descripción* **`"`**
`gh pr checks --watch`
`gh pr merge --squash --delete-branch`

`gh` es la CLI oficial de GitHub (su programa de terminal). Un **pull request** (PR) pide unir tu rama a `main`. `checks --watch` espera a que terminen los checks (los jobs de CI, las pruebas automáticas de cada cambio); `merge` (hacer la unión) falla si faltan aprobaciones o hay un check obligatorio en rojo.

`git tag -a` *versión* **`-m "`** *mensaje* **`"`**
`git push origin` *versión*

Un **tag** anotado es una etiqueta fija sobre un commit, con autor, fecha y mensaje. `git push` no sube tags si no los nombras: usa `git push origin` *versión* o `--follow-tags` (solo anotados).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
org	dueño del repo en GitHub: una persona o una organización	`acme`
repo	nombre del repo	`classifier`
rama	tu rama de trabajo	`feat/slot-wait-env`
email	comentario de la llave, para reconocerla en la lista	`ana@example.com`
usuario	tu usuario de GitHub (lo escribe GitHub en la respuesta)	`ana-ml`
título	qué cambia, en una línea	`gateway: SLOT_WAIT_S configurable por variable de entorno`
descripción	el cambio, el motivo y cómo lo probaste	`Permite ajustar la espera sin rebuild. 2 tests nuevos.`
versión	nombre del tag; casi siempre mayor.menor.parche	`v1.4.0`
mensaje	mensaje del tag anotado	`gateway con SLOT_WAIT_S configurable`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`origin`	Nombre por defecto del remoto: el repo de GitHub al que haces push y pull.	`git remote -v`
`origin/main`	Rama de seguimiento: tu copia de cómo estaba main en GitHub la última vez que hablaste con él (fetch, pull o push). No se mueve sola.	`git status -sb → ## main...origin/main`
`URL por SSH o por HTTPS`	git@github.com:... entra con tu llave SSH; https://github.com/... entra con un token personal (PAT) o con el que guarda gh.	`git remote set-url origin git@github.com:acme/classifier.git`
`contraseña de tu cuenta`	No sirve para que Git entre por HTTPS (push, o clone y pull de un repo privado): GitHub dejó de aceptarla el 13 de agosto de 2021.	`remote: Support for password authentication was removed on August 13, 2021.`
`token personal (PAT)`	Clave larga que creas en GitHub y que va donde Git pide contraseña por HTTPS. Mejor el fine-grained: solo ciertos repos y ciertos permisos.	`github_pat_… (fine-grained) · ghp_… (classic)`
`gh auth login`	Inicia sesión por el navegador y guarda el token en el llavero de macOS. Si eliges HTTPS y aceptas, también configura Git.	`gh auth status`
`pull request (PR)`	Pedido para unir tu rama (head) a otra (base), con el diff, los checks y la review en una página.	`base: main ← head: feat/slot-wait-env`
`draft PR`	Borrador: no se puede hacer merge y no pide review a los dueños del código hasta que lo marcas listo.	`gh pr create --draft`

PIEZA	QUÉ HACE	EJEMPLO
review	Comment (opinas), Approve (apruebas) o Request changes (pides cambios). Tu propio PR no lo puedes aprobar.	`1 aprobación para mergear a main`
CODEOWNERS	Archivo que dice quién es dueño de cada ruta; GitHub le pide review automáticamente. Va en .github/, en la raíz o en docs/.	`reference/c01/model/ @acme/ml-platform`
checks obligatorios	Jobs de CI que deben terminar en verde antes del merge (required status checks). Se eligen por el nombre del job.	`tests`
rama al día (strict)	Exige que tu rama tenga lo último de main antes del merge, para que los checks prueben la combinación real.	`botón Update branch del PR`
regla de protección o ruleset	Las dos protegen ramas. A una rama le aplica una sola regla de protección; rulesets, varios a la vez, y gana lo más estricto.	`hasta 75 rulesets por repo`
release	Página de GitHub colgada de un tag, con notas y archivos. gh la crea y, si el tag no existe, lo crea sobre lo último de la rama por defecto (main).	`gh release create v1.4.0 --generate-notes`

PULL REQUEST `feat/slot-wait-env` → `main` de tu rama (head) a la rama base

1. **Rama y commits** (tú)
   `git switch -c feat/slot-wait-env`
   Todo pasa en tu Mac; main no se toca.
2. **Push** (tú)
   `git push -u origin feat/slot-wait-env`
   La rama existe en GitHub.
3. **PR abierto** (tú)
   `gh pr create --base main`
   GitHub arma el diff y pide review a los CODEOWNERS de las rutas tocadas.
4. **Checks** (GitHub Actions)
   `gh pr checks --watch`
   El job tests corre sobre el merge de prueba con main.
5. **Review** (otra persona del equipo)
   Approve o Request changes. Un push nuevo puede descartar la aprobación, si la regla lo pide.
6. **Merge** (tú)
   `gh pr merge --squash --delete-branch`
   Solo se habilita con las reglas cumplidas.
7. **Push a main** (GitHub Actions)
   Dispara tests → imagen → deploy, que espera la aprobación del environment.

ANTES DEL MERGE: LAS REGLAS DE MAIN

Require a pull request before merging
1 aprobación
FRENA: push directo a main y merge sin review

Require status checks before merging
tests
FRENA: merge con el check en rojo o sin reportar

Require branches to be up to date before merging
activado
FRENA: merge de una rama atrasada respecto de main

Block force pushes
activado (default en ramas protegidas)
FRENA: reescribir la historia de main

tú · GitHub Actions · otra persona, en GitHub · reglas que hacen esperar al merge

La vida de un PR del gateway, de tu rama a main. Tres cosas pueden frenarlo: falta la aprobación, el check tests está en rojo o la rama quedó atrás de main.

LOS TRES BOTONES DE MERGE

MÉTODO	QUÉ QUEDA EN MAIN	BUENO PARA	OJO
Create a merge commit	Tus commits más un commit de merge que une las dos historias	Ver cuándo entró cada rama	Historia con bifurcaciones; git log --first-parent muestra solo la línea de main

MÉTODO	QUÉ QUEDA EN MAIN	BUENO PARA	OJO
Squash and merge	Un solo commit nuevo con todo el PR	Un commit por cambio: los primeros 7 caracteres de su SHA son el tag de la imagen del gateway	Se pierden los commits intermedios de la rama
Rebase and merge	Tus commits copiados uno por uno sobre main, sin commit de merge	Historia en línea recta y con detalle	GitHub crea commits nuevos, con otros SHA, aunque no haya conflictos

Con la regla Require linear history, main no acepta commits de merge. El tag de la imagen (3f9c2ab) son los primeros 7 caracteres del SHA del commit que quedó en main, no del de tu rama.

EJEMPLO REAL · UN CAMBIO AL GATEWAY, DE TU RAMA A MAIN

BASH · ZSH ilustrativo, armado con los pasos del curso (el gateway de reference/c01)

```bash
git switch -c feat/slot-wait-env               # rama nueva desde main
# … editas reference/c01/gateway/app/main.py y agregas 2 tests …
git add reference/c01/gateway
git commit -m "gateway: SLOT_WAIT_S configurable por variable de entorno"
git push -u origin feat/slot-wait-env          # crea la rama en GitHub

gh pr create --base main --title "gateway: SLOT_WAIT_S configurable" \
  --body "Permite ajustar la espera sin rebuild. 2 tests nuevos en test_api.py."
gh pr checks --watch                           # espera el check tests (pytest)

# Con 1 aprobación y tests en verde:
gh pr merge --squash --delete-branch
git switch main && git pull                    # tu main local, con el commit squash
git log --oneline -1                           # sus primeros 7 caracteres serán el tag de la imagen
```

- `git push -u` crea la rama en GitHub y la deja como upstream (la rama remota que sigue): los próximos `git push` no necesitan argumentos.
- Los checks de un PR no prueban tu rama sola: corren sobre un **merge de prueba** de tu rama con main (`refs/pull/N/merge`). Por eso un PR puede fallar en CI aunque tu rama sola pase: combinada con el main de ese momento, es otro código.
- `gh pr merge` respeta las reglas de la rama: sin aprobación o con `tests` en rojo, falla. Con `--auto` queda programado para cuando se cumplan, si el repo tiene activado auto-merge (Allow auto-merge).
- Con squash, main recibe **un** commit nuevo. Los primeros 7 caracteres de su SHA (el identificador del commit, por ejemplo `3f9c2ab`) son el tag de la imagen que construye el workflow del gateway. En tu Mac salen con `git rev-parse --short=7 HEAD` ; sin `=7` , git da más caracteres cuando el repo crece.
- `--delete-branch` borra la rama local y la remota después del merge: lo que importa ya está en main.

CÓMO RECORDARLA

Rama → push → PR → checks y review → merge. Quien dice que no es la regla de la rama: sin aprobación o con `tests` en rojo, el merge no se habilita.

TRAMPA

Hacer push directo a main "porque es un cambio chico": si main no está protegida, GitHub lo acepta y el CD (el despliegue automático) construye y despliega algo que nadie revisó. Protege main desde el día 1 (PR obligatorio y check `tests`); en un repo privado del plan Free no hay reglas de rama: hace falta Pro, Team o Enterprise.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M11 · Rollback: criterios automáticos y qué guardar para volver · Herramientas · Infraestructura como código, CI y seguridad · C01 · Archivos de referencia

6 Anatomía de un workflow de Actions

¿Qué partes tiene un workflow, en qué orden corre cada cosa y cómo se pasan datos entre sí?

SINTAXIS CON VARIABLES

```
name: nombre
on:
  evento:
    branches: [ rama ]
permissions:
  contents: read
jobs:
  job:
    runs-on: runner
    timeout-minutes: minutos
    steps:
      - uses: dueño / acción @{ versión }
        with:
          input: valor
      - run: comando
```

Un **workflow** es un archivo YAML en `.github/workflows/`. `on` dice cuándo corre. Cada **job** corre en su propio **runner** (la máquina que lo ejecuta; los de GitHub son una máquina nueva y limpia en cada job) y sus **steps** (pasos) corren en orden dentro de esa máquina.

```
  job_b:
    needs: [ job_a ]
    if: github.event_name == 'evento'
```

Sin `needs`, los jobs corren en paralelo. Si `job_a` falla o se salta, `job_b` también se salta, salvo que su `if` use `always()`.

```
    outputs:
      salida: ${{ steps.paso.outputs.salida }}
    steps:
      - id: paso
        run: echo "salida=valor" >>
"$GITHUB_OUTPUT"
# en un job con needs: [ job ] se lee ${{
needs.job.outputs.salida }}
```

Un paso deja un dato escribiendo en el archivo `$GITHUB_OUTPUT` (reemplaza al viejo `::set-output`). Otro job lo lee solo si nombra al primero en `needs`.

```
on:
  workflow_dispatch:
    inputs:
      input:
        type: tipo
        required: true
```

Agrega el botón **Run workflow**, que solo aparece si el archivo está en la rama por defecto. Hasta 25 inputs; se leen con `inputs.input`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
nombre	nombre que ves en la pestaña Actions	`gateway`
evento	qué lo dispara	`push, pull_request, workflow_dispatch, schedule`
rama	filtro de ramas; en pull_request, la rama base del PR	`main`
job	id del job: empieza con una letra o _ y sigue con letras, números, - o _	`tests`
runner	etiqueta de la máquina	`ubuntu-24.04 · [self-hosted, linux, x64, gpu-t4]`
minutos	tope del job; si falta, 360	`10`
dueño	usuario u organización dueña de la acción	`actions`
acción	repo de la acción	`checkout`
versión	versión mayor o SHA de 40 caracteres	`v7`
input	parámetro de la acción o del workflow	`python-version`
valor	su valor; los números de versión, entre comillas	`"3.13"`
comando	comando de shell	`pytest`
job_a	job del que depende	`tests`
job_b	job que espera	`imagen`
salida	nombre del dato que pasa	`digest`
paso	id del paso que lo escribe	`build`
tipo	tipo del input	`string, boolean, choice, number, environment`

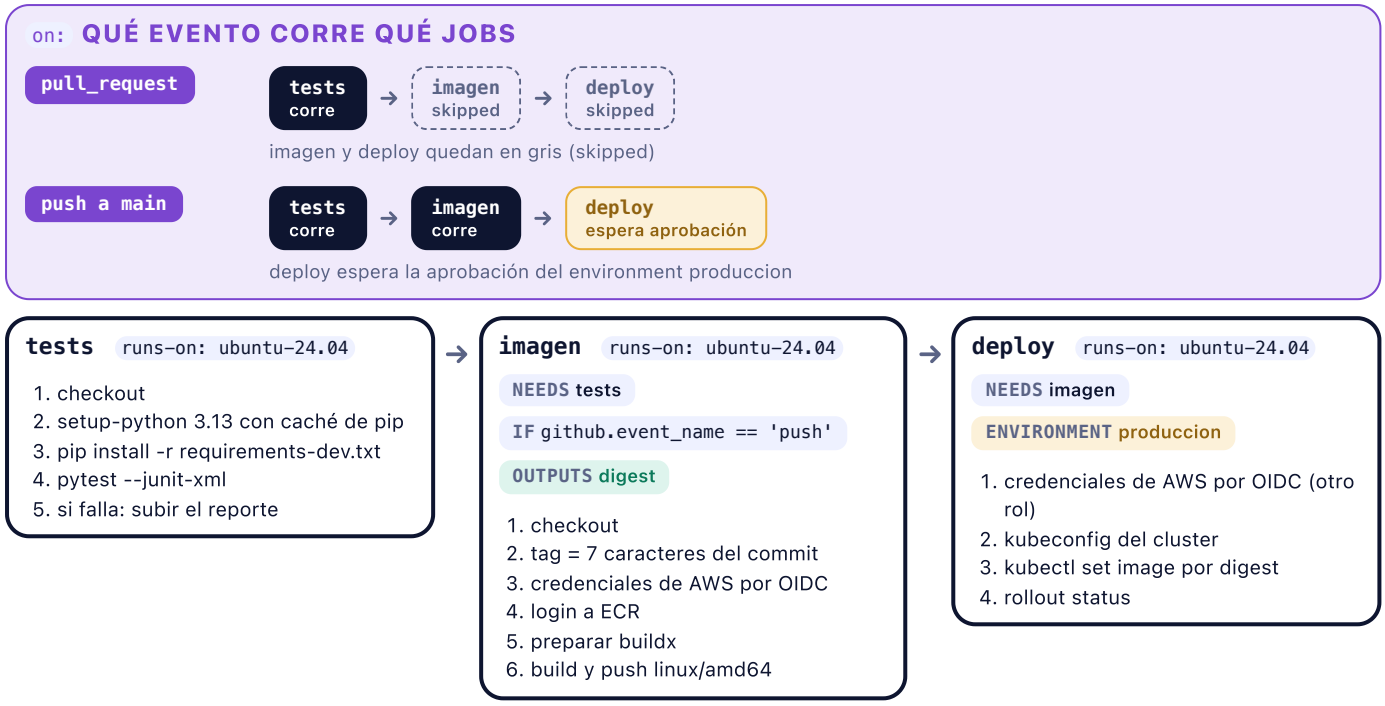
FÓRMULAS

$$t_{\text{workflow}} \approx \max_{\text{caminos de needs}} \sum_{\text{jobs del camino}} \left(t_{\text{esperar runner}} + t_{\text{job}} \right)$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
on	Los eventos que lo disparan: un push, un PR, el botón (workflow_dispatch) o un horario (schedule).	`on: [push, pull_request]`
branches / paths	Filtros: solo ciertas ramas, o solo si el commit toca ciertos archivos.	`paths: ['reference/c01/gateway/**']`
jobs	Cada clave bajo jobs es un job: un grupo de pasos que corre en su propio runner. No cuentes con los archivos de otro job: pásalos con outputs o artifacts.	`tests · imagen · deploy`
runs-on	En qué runner corre: una etiqueta de GitHub o las etiquetas de tu propia máquina.	`ubuntu-24.04`
steps	Los pasos del job, en orden; comparten carpeta y archivos.	`checkout → pip install → pytest`
uses + with	uses corre una acción (código ya hecho, de otro repo) en una versión fija; with le pasa sus inputs.	`actions/setup-python@v7 con python-version: "3.13"`
run	Comandos de terminal (shell). En Linux, si no pones la clave shell, corre con bash -e: para en el primer error, pero no ve un fallo en medio de un pipe (|). Con shell: bash suma pipefail, que sí lo ve.	`run: pytest`
env	Variables de entorno del workflow, del job o del paso; si un nombre se repite, gana la más cercana al paso.	`env: { IMAGEN: … }`
needs	Este job espera a otros. Si uno falla o se salta, este también se salta, salvo que su if use always().	`needs: tests`
if	Corre el job o el paso solo si se cumple. Sin if vale success(): que nada antes haya fallado.	`if: failure() · if: always()`
${{ }}	Expresión: GitHub calcula su valor y lo pega en el YAML antes de correr el paso.	`${{ github.sha }}`
contextos	Objetos con datos del run (esta ejecución del workflow): github, env, vars, secrets, inputs, steps, needs, matrix, runner.	`github.ref_name → main`
$GITHUB_OUTPUT · $GITHUB_ENV	Archivos donde un paso deja una salida o una variable para los pasos que siguen (no para sí mismo).	`echo "tag=3f9c2ab" >> "$GITHUB_OUTPUT"`
timeout-minutes	Corta el job a los N minutos. Si falta: 360 (6 h).	`timeout-minutes: 10`

ARCHIVO `.github/workflows/gateway.yml`



El workflow del gateway como grafo: cada evento enciende un camino distinto de jobs, y cada job es una máquina nueva con sus pasos en orden.

EJEMPLO REAL · EL ESQUELETO DEL WORKFLOW DEL GATEWAY: TRES JOBS Y QUIÉN ESPERA A QUIÉN

YAML `.github/workflows/gateway.yml (ilustrativo, armado con los pasos del curso; extracto)`

```yaml
name: gateway
on:
  pull_request:
  push: { branches: [main] }
permissions:
  contents: read
jobs:
  tests:
    runs-on: ubuntu-24.04
    timeout-minutes: 10
    defaults:
      run: { working-directory: reference/c01/gateway }
    steps:
      # … checkout, setup-python, pip install, pytest …

  imagen:
    needs: tests
    if: github.event_name == 'push'
    runs-on: ubuntu-24.04
    outputs:
      digest: ${{ steps.build.outputs.digest }}
    steps:
      - id: tag
        run: echo "tag=${GITHUB_SHA::7}" >> "$GITHUB_OUTPUT"
      # … credenciales de AWS, login a ECR y build con id: build …

  deploy:
    needs: imagen
    environment: produccion
    # … usa ${{ needs.imagen.outputs.digest }} …
```

- Un PR dispara solo `tests`: `imagen` tiene `if: github.event_name == 'push'`, así que se salta, y `deploy`, que necesita a `imagen`, también.
- Un push a main corre los tres en cadena porque `needs` los ordena. Sin `needs` correrían a la vez, cada uno en su runner.
- `defaults.run.working-directory` vale para los `run` del job. Los `with` de las acciones siguen siendo rutas desde la raíz del repo.
- El digest viaja en dos saltos: el paso `build` lo devuelve, el job lo publica en `outputs` y `deploy` lo lee con `needs.imagen.outputs.digest`.

- Arriba, `permissions` deja el `GITHUB_TOKEN` (la llave que GitHub le da a cada job) con solo `contents: read` en todos los jobs; los que hablan con AWS piden más en su propio bloque (sección 8).

CÓMO RECORDARLA

on (cuándo) → **jobs** (qué, cada uno en su propio runner) → **steps** (en orden, en la misma máquina). Entre pasos, los datos viajan por archivos y `$GITHUB_OUTPUT`; entre jobs, solo por `outputs` o por artifacts (archivos que guarda el run).

TRAMPA

Pegar `${{ github.event.pull_request.title }}`, o cualquier texto que escribe otra persona, dentro de `run`: GitHub lo copia tal cual al script antes de correrlo, y un título como `a"; curl … | sh; "` se ejecuta en tu runner. Pásalo por `env` y úsalo como `"$TITULO"`. Y un `if: ${{ x }} == 'y'` siempre es verdadero: es un texto, no una comparación.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · Herramientas · Infraestructura como código, CI y seguridad · M15 · Supply chain: imágenes, dependencias y pesos verificados · C01 · Archivos de referencia

7 Tests en CI: Python, caché, matrix y checks obligatorios

¿Cómo corres pytest en cada PR, rápido y sin que un test rojo llegue a main?

SINTAXIS CON VARIABLES

```
- uses: actions/setup-python@v7
  with:
    python-version: "versión"
    cache: pip
    cache-dependency-path:
carpeta/requirements*.txt
- run: pip install -r carpeta/requirements-dev.txt
- run: pytest --junit-xml=reporte.xml
```

En CI (integración continua: las pruebas automáticas que corren con cada push y cada PR), la caché de pip guarda los paquetes que pip **bajó**, no el entorno instalado: el `pip install` corre igual, pero saca los paquetes de la caché en vez de bajarlos. La versión va entre comillas: en YAML, `3.10` sin comillas es el número 3.1.

```
- uses: actions/cache@v6
  with:
    path: carpeta_cache
    key: so-herramienta-${{
hashFiles('archivo_lock') }}
    restore-keys: |
      so-herramienta-
```

Una **caché** (archivos que se guardan entre runs para no repetir trabajo) se busca por su `key`. Si coincide exacta es un **hit**; si no, `restore-keys` trae la más reciente que empiece igual. Una caché no se reescribe: si cambia el archivo, cambia la key y se guarda otra.

```
strategy:
  fail-fast: false
  matrix:
    python-version: ["v1", "v2"]
steps:
  - uses: actions/setup-python@v7
    with:
      python-version: ${{ matrix.python-version }}
```

Una **matrix** repite el job por cada combinación de valores, y cada copia es un check aparte (su propio resultado en el PR), como `tests (3.13)`. Con `fail-fast: true`, el default, el primer rojo cancela a los demás.

```
- if: failure()
  uses: actions/upload-artifact@v7
  with:
    name: nombre
    path: ruta
    retention-days: días
```

Un **artifact** es un archivo del run que bajas desde su página (en un zip). Sin `retention-days`, dura lo que diga el repo: 90 días por defecto. `if: failure()` lo sube solo cuando algo falló.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
versión	versión de Python, entre comillas	`"3.13"`
carpeta	carpeta del proyecto dentro del repo	`reference/c01/gateway`
reporte	nombre del reporte JUnit XML (un formato estándar de resultados de tests)	`reporte-tests`
carpeta_cache	carpeta que quieres guardar entre runs	`~/.cache/torch/hub/checkpoints`
so	sistema operativo del runner (runner.os)	`Linux`
herramienta	prefijo que nombra la caché	`pesos-torchvision`
archivo_lock	archivo cuyo contenido cambia cuando cambia lo que guardas	`model/requirements.txt`
v1	primera versión a probar	`3.13`
v2	segunda versión	`3.14`
nombre	nombre del artifact, único en el run	`reporte-tests`
ruta	archivo o carpeta, desde la raíz del repo	`reference/c01/gateway/reporte-tests.xml`
días	días que se guarda: de 1 al máximo del repo (90 por defecto; un repo privado puede subirlo hasta 400)	`14`

FÓRMULAS

$$N_{\text{jobs}} = \prod_{k} |\text{valores}_k| - N_{\text{exclude}} + N_{\text{include nuevos}} \leq 256$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
actions/setup-python	Instala la versión de Python pedida (o la toma de las que ya trae el runner) y la pone primera en el PATH: los pasos siguientes usan ese python.	`python-version: "3.13"`
cache: pip	Guarda la carpeta de descargas de pip; la key incluye el hash de tus requirements.	`cache-dependency-path: reference/c01/gateway/requirements*.txt`
actions/cache	Caché a medida: tú eliges la carpeta y la key.	`~/.cache/torch/hub/checkpoints`
hit / miss	Hit: la key coincide y se restaura al empezar. Miss: el job hace todo y, si termina bien, guarda la caché al final.	`salida cache-hit: true`
límite y limpieza	10 GB por repo por defecto. Se borra lo que no se usa en 7 días y, si te pasas del límite, lo que hace más tiempo que nadie usa.	`un admin puede subir el límite, con pago`
alcance	Un run usa cachés de su rama y de la rama por defecto (un PR, también las de su rama base); la que crea un PR solo la ve ese PR.	`un push a main calienta la caché de todos los PRs`
cache-mode	Cuánto puede usar el job la caché: read (solo restaurar), write (restaurar y guardar), write-only (solo guardar) o none. En eventos que puede disparar cualquiera, como pull_request_target, el default es read.	`cache-mode: read`
pytest	Sale con 0 si todo pasó, 1 si algún test falló y 5 si no encontró tests. --junit-xml deja el resultado en un XML.	`pytest --junit-xml=reporte-tests.xml`
strategy.matrix	Repite el job por cada combinación de valores. Máximo 256 jobs por run.	`python-version: ["3.13", "3.14"]`
fail-fast	true (default): el primer job rojo de la matrix cancela a los otros. false: ves todos los resultados.	`fail-fast: false`
actions/upload-artifact	Sube archivos del run en un zip. Ignora archivos ocultos y no acepta dos artifacts con el mismo nombre en un run, salvo con overwrite: true (reemplaza al anterior).	`name: reporte-tests`
retención	Artifacts y logs duran 90 días por defecto; el repo puede elegir de 1 a 90 si es público y de 1 a 400 si es privado. Desde el 1 de octubre de 2026 el mismo plazo vale para checks y runs.	`retention-days: 14`
check obligatorio	Un job que debe quedar en verde para el merge. Para elegirlo en la regla, tiene que haber terminado en verde en este repo en los últimos 7 días.	`tests`
continue-on-error	Deja seguir al job aunque el paso falle. En un paso de tests, esconde los rojos.	`no lo uses con pytest`

	PARA QUÉ	QUIÉN LO LEE	CUÁNTO VIVE	LÍMITE	EN EL GATEWAY
Caché (actions/cache, cache: pip)	Ir más rápido: no volver a bajar ni a calcular	Los runs siguientes de la misma rama o de la rama por defecto	Hasta 7 días sin uso (configurable)	10 GB por repo por defecto	Descargas de pip; capas de Docker (type=gha)
Artifact (upload-artifact)	Guardar un resultado para mirarlo o pasarlo a otro job	Personas, desde la página del run, u otro job	90 días por defecto	Cuota de almacenamiento de la cuenta; 500 por job	reporte-tests.xml cuando algo falla
Output de job (outputs)	Pasar un dato chico al job siguiente	Jobs que lo nombran en needs	Lo que dura el run	1 MB por job, 50 MB por run	El digest de la imagen

Si borrar algo rompe el pipeline, no era caché. Y una salida que contiene un secret no viaja al otro job: GitHub la descarta.

Tres lugares donde CI guarda cosas. Confundirlos es la causa más común de "en el run anterior estaba".

EJEMPLO REAL · EL JOB TESTS DEL GATEWAY: 24 TESTS SIN GPU EN CADA PR

YAML .github/workflows/gateway.yml (ilustrativo, armado con los pasos del curso; extracto)

```yaml
tests:
  runs-on: ubuntu-24.04
  timeout-minutes: 10
  defaults:
    run: { working-directory: reference/c01/gateway }
  steps:
    - uses: actions/checkout@v7
      with: { persist-credentials: false }
    - uses: actions/setup-python@v7
      with:
        python-version: "3.13"
        cache: pip
        cache-dependency-path: reference/c01/gateway/requirements*.txt
    - run: pip install -r requirements-dev.txt
    - run: pytest --junit-xml=reporte-tests.xml
    - if: failure()
      uses: actions/upload-artifact@v7
      with: { name: reporte-tests, path: reference/c01/gateway/reporte-tests.xml }
```

- Python 3.13, igual que `python:3.13-slim` del Dockerfile: los tests prueban las mismas versiones que corren en producción, porque `requirements-dev.txt` incluye a `requirements.txt`.
- `requirements*.txt` toma los dos archivos: si cambia cualquiera, cambia la key y pip vuelve a descargar lo que necesite.
- `pytest` lee `pytest.ini` del gateway (`testpaths = tests`, `-q`). Los 24 tests usan un Triton falso: no hace falta GPU.
- El reporte sube solo si algo falló. Su ruta va desde la raíz del repo: `working-directory` solo afecta a los `run`.
- El job se llama `tests`, y ese es el nombre que eliges como check obligatorio en la regla de main.

CÓMO RECORDARLA

Caché para ir rápido (si se pierde, solo tarda más), **artifact** para mirar después y **check obligatorio** para que nada rojo llegue a main.

TRAMPA

Filtrar con `paths` un workflow cuyo check es obligatorio: un PR que no toca esas rutas no lo corre, el check queda en `Waiting for status to be reported` y el PR no se puede mergear. Pon la condición en el `if` de un job: un job saltado por `if` cuenta como verde.

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

8 Secretos, permisos y OIDC: entrar a AWS sin llaves

¿Con qué permisos corre tu workflow, dónde guardas lo secreto y cómo entra a AWS sin guardar ninguna llave?

SINTAXIS CON VARIABLES

```
permissions:
  contents: read
jobs:
  job:
    permissions:
      contents: read
      permiso: write
```

El **GITHUB_TOKEN** es la llave temporal que GitHub le da a cada job para hablar con tu repo. `permissions` fija qué puede hacer: lo que no nombras queda en `none`, y un bloque en el job reemplaza al de arriba, así que ahí repites `contents: read`.

```
- name: paso
  env:
    VARIABLE: ${{ secrets.SECRETO }}
    OTRA: ${{ vars.VARIABLE_CONFIG }}
  run: comando
```

Un **secret** es un valor cifrado que guardas en Settings; en el log aparece como `***`. Una **variable** (`vars`) es configuración no secreta y se ve tal cual. Las dos entran por `env`, nunca pegadas en el comando.

```
permissions:
  id-token: write
  contents: read
steps:
  - uses: aws-actions/configure-aws-
credentials@{sha_acción} # v6.3.0
    with:
      role-to-assume: arn:aws:iam::cuenta:role/rol
      aws-region: región
```

OIDC (OpenID Connect): el job pide un token firmado por GitHub que dice quién es (repo, rama o environment, un entorno con reglas como `produccion`), y AWS lo cambia por credenciales de un **rol** (una identidad con permisos que el job toma prestada) que vencen solas, en 1 hora por defecto. `id-token: write` solo permite pedir ese token.

```
"Condition": {
  "StringEquals": {
    "token.actions.githubusercontent.com:aud":
"sts.amazonaws.com",
    "token.actions.githubusercontent.com:sub":
"repo:org/repo:ref:refs/heads/rama"
  }
}
```

La **trust policy** del rol dice qué tokens pueden asumirlo. El `sub` es el campo del token que dice quién lo pide; IAM (el servicio de permisos de AWS) rechaza la política si no lo revisa o si acepta cualquiera (`*`). Con environment, el sub pasa a ser `repo:org/repo:environment:entorno`. Hay dos formatos: un repo creado, renombrado o transferido después del 15 de julio de 2026 (o que eligió el formato nuevo) agrega los IDs numéricos del dueño y del repo, como `repo:acme@123/classifier@456:ref:refs/heads/main`. Copia el que usa tu repo: con el otro, AWS responde `Not authorized`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
job	id del job	`imagen`
permiso	el permiso extra que necesita ese job	`id-token (AWS), packages (GHCR), pull-requests (comentar el PR)`
paso	nombre del paso	`Bajar pesos del hub`
VARIABLE	variable de entorno que lee tu código	`HF_TOKEN`
SECRETO	nombre del secret: letras, números y _; no empieza con GITHUB_ ni con un número	`HF_TOKEN`
OTRA	otra variable de entorno	`MLFLOW_TRACKING_URI`
VARIABLE_CONFIG	nombre de la variable de configuración	`MLFLOW_TRACKING_URI`
comando	lo que lee esas variables	`python descargar_pesos.py`
sha_acción	commit de 40 caracteres de la acción; la versión va en el comentario	`e1253824e5c10ff9df46874f81ed3ec929e19cfd`
cuenta	id de la cuenta de AWS, 12 dígitos	`123456789012`
rol	rol de IAM que asume el job	`gha-classifier-gateway`
región	región de AWS	`us-east-1`
org	dueño del repo en GitHub	`acme`
repo	nombre del repo	`classifier`
rama	rama que puede asumir el rol	`main`
entorno	environment del job	`produccion`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
GITHUB_TOKEN	Token que GitHub crea al empezar cada job y que solo sirve para este repo. Vence al terminar el job: como mucho, 6 h en runners de GitHub y 24 h en runners propios.	`secrets.GITHUB_TOKEN · github.token`
permisos por defecto	Los fija el repo o la org. En repos personales y orgs creados desde febrero de 2023: solo lectura de contents y packages. En los nuevos, el workflow tampoco puede crear ni aprobar PRs. No dependas de eso: decláralo.	`Settings → Actions → Workflow permissions`
permissions:	Ajusta el GITHUB_TOKEN del workflow o del job. Lo no nombrado queda en none; write-all abre todo.	`contents: read`
id-token: write	Deja pedir un token OIDC. No da permiso de escribir en el repo.	`jobs imagen y deploy del gateway`
secrets	Valores cifrados de org, de repo o de environment; si un nombre se repite, gana el más específico. 100 por repo, 48 KB cada uno.	`secrets.HF_TOKEN`
vars	Configuración no secreta, visible en el log: URLs, nombres, regiones.	`vars.MLFLOW_TRACKING_URI`
***** en el log**	GitHub tapa el valor exacto de un secret. Un valor derivado (en base64, recortado, un token armado con él) no se tapa solo.	`::add-mask::VALOR tapa uno nuevo`
PRs desde forks	Con el evento pull_request, un PR desde un fork (la copia del repo en la cuenta de otra persona) no recibe secrets, y su GITHUB_TOKEN es de solo lectura.	`el job tests no necesita ninguno`
environment	Un entorno (como produccion) con reglas: quién aprueba, desde qué ramas se despliega y sus propios secrets.	`environment: produccion`
environment inexistente	Si el nombre no existe, el run lo crea vacío, sin reglas ni aprobación.	`produccion ≠ producción`
token OIDC (JWT)	Un texto firmado por GitHub para ese job (JWT es el formato estándar de ese texto), con datos (claims) como sub, aud, ref y environment.	`iss: https://token.actions.githubusercontent.com`
sub	Quién lo pide: repo y rama, PR o environment. La trust policy lo compara. Los repos creados o renombrados después del 15 de julio de 2026 (o que lo activan) lo llevan con IDs.	`repo:acme/classifier:ref:refs/heads/main · con IDs: repo:acme@123/classifier@456:ref:refs/heads/main`
aud	Para quién es el token. Para AWS es sts.amazonaws.com, el default de configure-aws-credentials.	`token.actions.githubusercontent.com:aud`
credenciales temporales	STS (el servicio de AWS que entrega credenciales) devuelve llave, secreto y token de sesión que vencen solos: 1 h por defecto.	`role-duration-seconds: 3600`

1 **Job imagen (GitHub Actions)**

Pide un token OIDC

Tiene permissions: id-token: write; configure-aws-credentials lo pide con aud = sts.amazonaws.com.

2 **GitHub (token.actions.githubusercontent.com)**

Firma un JWT con los datos del job

sub = repo:acme/classifier:ref:refs/heads/main

TOKEN OIDC (UN JWT) FIRMADO POR GITHUB

iss	https://token.actions.githubusercontent.com
aud	sts.amazonaws.com
sub	repo:acme/classifier:ref:refs/heads/main
ref	refs/heads/main
event_name	push

3 AWS STS

Cambia el token por el rol gha-classifier-gateway (AssumeRoleWithWebIdentity)

Verifica la firma con el proveedor OIDC registrado en IAM y compara aud y sub con la trust policy.

LA TRUST POLICY DEL ROL COMPARA

✓ `aud` del token: `sts.amazonaws.com`
LA POLICY PIDE: `sts.amazonaws.com`

✓ `sub` del token: `repo:acme/classifier:ref:refs/heads/main`
LA POLICY PIDE: `repo:acme/classifier:ref:refs/heads/main`

Coinciden: STS entrega credenciales.

▼ La trust policy entera (JSON)

JSON trust policy del rol

```
{
  "Effect": "Allow",
  "Principal": { "Federated":
"arn:aws:iam::123456789012:oidc-
provider/token.actions.githubusercontent.com" },
  "Action": "sts:AssumeRoleWithWebIdentity",
  "Condition": {
    "StringEquals": {
      "token.actions.githubusercontent.com:aud":
"sts.amazonaws.com",
      "token.actions.githubusercontent.com:sub":
"repo:acme/classifier:ref:refs/heads/main"
    }
  }
}
```

4 AWS STS

Devuelve credenciales temporales

Llave, secreto y token de sesión: 1 h por defecto.

5 Job imagen

Login a ECR y push

amazon-ecr-login canjea esas credenciales por un token de ECR que dura 12 h.

QUÉ SUB MANDA CADA JOB, CONTRA ESTA TRUST POLICY

CUÁNDO	SUB DEL TOKEN	¿ENTRA?
push a main, job sin environment	`repo:acme/classifier:ref:refs/heads/main`	✓ entra
pull_request, job sin environment	`repo:acme/classifier:pull_request`	✕ Not authorized
job con environment: produccion	`repo:acme/classifier:environment:produccion`	✕ Not authorized
push a main en un repo creado o renombrado después del 15 de julio de 2026 (formato con IDs)	`repo:acme@123/classifier@456:ref:refs/heads/main`	✕ Not authorized

el job (GitHub Actions) · GitHub · AWS · credenciales: vencen solas

Entrar a AWS sin llaves: GitHub firma quién es el job, AWS compara ese quién (el sub) con la trust policy del rol y STS entrega credenciales de una hora. El token y la trust policy usan el formato de siempre; la última fila muestra el sub con IDs de un repo creado o renombrado después del 15 de julio de 2026.

UN ROL POR TRABAJO EN EL C01: CADA UNO CON EL SUB DEL JOB QUE LO USA Y SOLO LOS PERMISOS QUE NECESITA

ROL DE IAM	LO USA	SUB EN LA TRUST POLICY	PUEDE
gha-classifier-gateway	job imagen (push a main)	repo:acme/classifier:ref:refs/heads/main	push a ECR, solo al repo classifier-gateway
gha-classifier-deploy	job deploy (environment produccion)	repo:acme/classifier:environment:produccion	eks:DescribeCluster (para armar el kubeconfig) y desplegar en el namespace classifier, con una access entry de EKS (el permiso del rol dentro del cluster)

ROL DE IAM	LO USA	SUB EN LA TRUST POLICY	PUEDE
gha-classifier-modelo	workflow triton-modelo (botón, desde main)	repo:acme/classifier:ref:refs/heads/main	leer el checkpoint y la muestra de eval en S3; push a classifier-triton

Los nombres son ilustrativos, y los sub tienen el formato de un repo de antes del 15 de julio de 2026; en uno más nuevo llevan IDs (`repo:acme@123/classifier@456:…`). Ojo: gateway y modelo aceptan el mismo sub, el de cualquier job en main; para separarlos, dale a triton-modelo su propio environment. Un solo rol para todo también funciona, pero entonces cualquier job que consiga un token puede desplegar.

EJEMPLO REAL · EL JOB DEPLOY DEL GATEWAY: ENVIRONMENT, UN ROL PROPIO Y NINGÚN SECRET

YAML .github/workflows/gateway.yml (ilustrativo, armado con los pasos del curso; extracto)

```yaml
deploy:
  needs: imagen
  runs-on: ubuntu-24.04
  timeout-minutes: 20
  environment: produccion
  concurrency: { group: deploy-gateway-produccion, cancel-in-progress: false }
  permissions:
    contents: read
    id-token: write
  steps:
    - uses: aws-actions/configure-aws-credentials@e1253824e5c10ff9df46874f81ed3ec929e19cfd # v6.3.0
      with:
        role-to-assume: arn:aws:iam::123456789012:role/gha-classifier-deploy
        aws-region: us-east-1
    - env:
        IMAGEN: 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-gateway@${{ needs.imagen.outputs.digest }}
      run: |
        aws eks update-kubeconfig --name classifier --region us-east-1
        kubectl -n classifier set image deployment/gateway gateway="$IMAGEN"
        kubectl -n classifier rollout status deployment/gateway --timeout=5m
```

- No hay ningún secret: las credenciales salen de OIDC y el nombre de la imagen no es secreto.
- Por `environment: produccion`, el sub del token es `repo:acme/classifier:environment:produccion` (en un repo con IDs, `repo:acme@123/classifier@456:environment:produccion`). La trust policy de `gha-classifier-deploy` pide ese valor: el job `imagen` no puede asumir este rol, y otro job (o un PR) solo podría si usa ese environment. Por eso `produccion` lleva reviewers y solo deja desplegar desde `main` .
- El job espera la aprobación del environment antes de recibir un runner; si tuviera secrets de environment, recién los recibiría después de aprobar.
- La aprobación manual (required reviewers) en un repo privado exige GitHub Enterprise; con Free, Pro o Team solo existe en repos públicos. Las ramas permitidas del environment sí funcionan en privados con Pro o Team.
- `permissions` repite `contents: read` : al declarar el bloque en el job, lo que no nombras queda en `none` .

CÓMO RECORDARLA

Sin llaves guardadas: `id-token: write` → token firmado por GitHub → la trust policy compara **sub** → credenciales que vencen solas. Lo secreto va en `secrets` , lo demás en `vars` , y nada de eso en el YAML.

TRAMPA

Agregar `environment: produccion` a un job que ya entraba a AWS: su sub cambia de `…:ref:refs/heads/main` a `…:environment:produccion` y STS lo rechaza con `Not authorized to perform sts:AssumeRoleWithWebIdentity` . Renombrar o transferir el repo hace lo mismo: desde el 15 de julio de 2026, eso lo pasa al sub con IDs. Y escribir mal el nombre del environment no da error: crea uno nuevo, sin aprobación.

En el curso: M15 · Secretos y credenciales · M15 · Supply chain: imágenes, dependencias y pesos verificados · M15 · Autenticación y autorización entre servicios · Herramientas · Infraestructura como código, CI y seguridad

9 Build y push de imágenes en CI

¿Cómo construye CI (el sistema que prueba y arma cada cambio solo) una imagen una sola vez, para la CPU de los nodos, con un tag que nunca se reusa?

SINTAXIS CON VARIABLES

```
- uses: docker/setup-buildx-action@{ sha_acción } # v4.4.1
- id: build
  uses: docker/build-push-action@{ sha_acción } # v7.4.0
  with:
    context: carpeta
    platforms: linux/amd64
    push: true
    tags: registry / repo : tag
    cache-from: type=gha
    cache-to: type=gha,mode=max
```

buildx es el build de Docker con BuildKit (el motor que arma las capas). `context` es la carpeta con el Dockerfile: si es una ruta, necesitas `actions/checkout` antes. `type=gha` guarda las capas en la caché de Actions.

```
- id: tag
  run: echo "tag=${GITHUB_SHA::7}" >> "$GITHUB_OUTPUT"
# después: ${{ steps.tag.outputs.tag }} → tag
```

Los primeros 7 caracteres del commit, como `3f9c2ab` en `k8s/gateway.yaml`. En un PR, `GITHUB_SHA` es un merge de prueba que nunca llega a main: por eso la imagen se publica solo en push a main.

```
- uses: aws-actions/configure-aws-credentials@{ sha_acción } # v6.3.0
  with:
    role-to-assume: arn:aws:iam:: cuenta :role/ rol
    aws-region: región
- id: ecr
  uses: aws-actions/amazon-ecr-login@{ sha_acción } # v2.1.7
# ${{ steps.ecr.outputs.registry }} → cuenta .dkr.ecr. región .amazonaws.com
```

`amazon-ecr-login` hace el `docker login` a ECR (el registry de AWS: el servicio que guarda tus imágenes) con las credenciales del job. El token de ECR dura 12 h, y la action cierra la sesión (logout) al terminar el job.

```
docker buildx imagetools inspect registry / repo : tag
```

Desde tu Mac, con el login a ECR hecho y sin bajar la imagen: lista las plataformas del índice (la lista de variantes por CPU que guarda el registry). Debe decir `linux/amd64`; la entrada `unknown/unknown` es la attestation (el certificado de cómo se construyó) que agrega el build.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
sha_acción	commit de 40 caracteres de la acción; la versión va en el comentario	`c3c9e263c25d99ce0380d002d59b67737d91b0dc`
carpeta	contexto del build: la carpeta con el Dockerfile	`reference/c01/gateway`
registry	host del registry	`123456789012.dkr.ecr.us-east-1.amazonaws.com`
repo	repositorio de la imagen en el registry	`classifier-gateway`
tag	los primeros 7 caracteres del commit: nunca se reusa	`3f9c2ab`
cuenta	id de la cuenta de AWS	`123456789012`
rol	rol de IAM con permiso de push a ese repositorio	`gha-classifier-gateway`
región	región del registry	`us-east-1`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
docker/setup-buildx-action	Arranca BuildKit en un contenedor aparte (driver docker-container, el default): así sí puede guardar la caché en Actions.	`driver: docker-container (default)`

PIEZA	QUÉ HACE	EJEMPLO
docker/build-push-action	Corre docker buildx build con lo que pongas en with; push: true sube al terminar.	**salidas: digest, imageid, metadata**
context	Sin context usa el contexto Git: BuildKit baja el commit solo e ignora cambios hechos en pasos anteriores. Con una ruta, usa tu checkout.	**context: reference/c01/gateway**
platforms	Para qué CPU es la imagen. El runner ubuntu-24.04 es amd64: construye nativo, sin emular.	**linux/amd64**
tags	Los nombres con que sube la imagen, uno por línea o separados por comas.	**…/classifier-gateway:3f9c2ab**
cache-from / cache-to: type=gha	Guarda y reusa capas en la caché de Actions; mode=max incluye las etapas intermedias. Cuenta para los 10 GB del repo.	**cache-to: type=gha,mode=max**
provenance	El build agrega por defecto una attestation de procedencia (de dónde y cómo se construyó): mode=min en repos privados, max en públicos.	**aparece como unknown/unknown en el índice**
digest	El sha256 de lo que subió: identifica el contenido. Es lo que firmas y despliegas.	**steps.build.outputs.digest**
aws-actions/amazon-ecr-login	docker login a ECR con las credenciales de AWS del job. Su salida registry es el host de ECR.	**steps.ecr.outputs.registry**
permisos de IAM para el push	ecr:GetAuthorizationToken (sobre "*": no se limita por repo), más BatchCheckLayerAvailability, InitiateLayerUpload, UploadLayerPart, CompleteLayerUpload, PutImage y BatchGetImage sobre ese repo.	**rol gha-classifier-gateway**
tags inmutables en ECR	Es una opción de cada repo de ECR (por defecto, los tags se pueden pisar); el del C01 la tiene activada. Con ella, subir otra imagen con un tag que ya existe falla: tag invalid: … cannot be overwritten because the repository is immutable (en la API, ImageTagAlreadyExistsException).	**un re-run que reconstruye un commit ya publicado**
${GITHUB_SHA::7}	Recorte de bash: los primeros 7 caracteres de la variable, siempre 7. En tu Mac, lo mismo es git rev-parse --short=7 HEAD (sin =7, git da más cuando el repo crece).	**3f9c2ab**
salida con un secret	Si una salida de job contiene un valor enmascarado, GitHub no la pasa al otro job: Skip output … since it may contain secret.	**el digest viaja sin problema: no es secreto**
docker/login-action	Login genérico a un registry. En GHCR (el registry de GitHub) alcanza el GITHUB_TOKEN con packages: write.	**registry: ghcr.io**

gateway · gateway.yml (cada PR y cada push a main)

1. **PR abierto**
   GitHub
   El job tests corre sobre el merge de prueba con main.
2. **tests: pytest, 24 tests** `≈ 1–2 min`
   runner ubuntu-24.04
   Con la caché de pip, el pip install no descarga.
3. **review y merge** `lo que tarde la review`
   GitHub
   1 aprobación y el check tests en verde.
4. **tests otra vez y build linux/amd64** `≈ 2–5 min`
   runner ubuntu-24.04
   Con type=gha solo se rehacen las capas que cambiaron.
5. **push a ECR, tag = 7 caracteres del SHA** `≈ 1 min`
   Amazon ECR
   Sube solo las capas nuevas.
6. **aprobación del environment produccion**
   `lo que tarde la persona`
   GitHub
   El job deploy espera sin ocupar un runner.
7. **kubectl set image y rollout status** `≈ 1–3 min`
   EKS
   Pods nuevos listos antes de bajar los viejos (maxUnavailable: 0). Con GitOps, este paso es un PR que cambia el digest en el manifest.

Triton · triton-modelo.yml (botón, solo con un modelo nuevo)

1. **export a ONNX y paridad en CPU** `minutos`
   runner T4 propio
   parity_check.py con ONNX Runtime.
2. **engine TensorRT e imagen de Triton** `minutos`
   runner T4 propio
   build_engine.sh aborta si la GPU no es una T4.
3. **paridad FP16 contra ese Triton** `minutos`
   runner T4 propio
   Un exit 1 corta todo antes del push.
4. **push a ECR y registro como challenger**
   `minutos (imagen de varios GB)`
   Amazon ECR y MLflow
   El canary y el paso a champion siguen en CD.

tú · GitHub · automático (runner, Actions, Argo CD) · espera a una persona · AWS · Kubernetes

`≈ min` duración estimada

El C01 en CI/CD: el gateway en cada commit a main y Triton solo con un modelo nuevo. Duraciones aproximadas: estimaciones, no medidas.

EJEMPLO REAL · EL JOB IMAGEN DEL GATEWAY: TESTS EN VERDE, UNA IMAGEN POR COMMIT

YAML .github/workflows/gateway.yml (ilustrativo, armado con los pasos del curso; extracto)

```yaml
  imagen:
    needs: tests
    if: github.event_name == 'push'
    runs-on: ubuntu-24.04
    timeout-minutes: 20
    # permissions: contents: read e id-token: write (sección 8)
    outputs:
      digest: ${{ steps.build.outputs.digest }}
    steps:
      - uses: actions/checkout@v7
        with: { persist-credentials: false }
      - id: tag
        run: echo "tag=${GITHUB_SHA::7}" >> "$GITHUB_OUTPUT"
      - uses: aws-actions/configure-aws-credentials@e1253824e5c10ff9df46874f81ed3ec929e19cfd # v6.3.0
        with:
          role-to-assume: arn:aws:iam::123456789012:role/gha-classifier-gateway
          aws-region: us-east-1
      - id: ecr
        uses: aws-actions/amazon-ecr-login@03f1aad4c6c7ffd436567f42f9384779290529bd # v2.1.7
      - uses: docker/setup-buildx-action@f87e5991a6d7451dcb8d9637bfbc97413f497069 # v4.4.1
      - id: build
        uses: docker/build-push-action@c3c9e263c25d99ce0380d002d59b67737d91b0dc # v7.4.0
        with:
          context: reference/c01/gateway
          platforms: linux/amd64
          push: true
          tags: ${{ steps.ecr.outputs.registry }}/classifier-gateway:${{ steps.tag.outputs.tag }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

- Solo corre en push a main y después de `tests`: cada commit de main produce una imagen, construida una sola vez.
- `${GITHUB_SHA::7}` da el mismo tag de 7 caracteres que usa el curso (`3f9c2ab`), aunque el repo crezca. El repo de ECR del C01 está configurado con tags inmutables (no es el default de ECR): nadie puede volver a subir ese tag con otro contenido.
- `platforms: linux/amd64`: el runner ya es amd64, así que no emula. En tu Mac (arm64) el mismo build sería emulado y lento: que lo haga CI.
- `type=gha` reusa capas entre runs: si solo cambió `app/`, la capa del `pip install` sale de la caché, porque el Dockerfile copia `requirements.txt` antes que el código.
- El job publica `digest` como salida: `deploy` despliega ese contenido exacto, no un tag.

CÓMO RECORDARLA

Una imagen por commit, construida una vez en CI: tests → build `linux/amd64` → push con tag = 7 caracteres del SHA (`${GITHUB_SHA::7}`) → el deploy usa su digest. Nunca `latest`, nunca desde tu Mac.

TRAMPA

Construir en tu Mac y subir desde ahí "porque es más rápido": sale `linux/arm64` (o amd64 emulado y lento) y con un tag que ningún run de CI puede explicar. Y un re-run que reconstruye un commit ya publicado, con otro digest, falla en el push con `tag invalid: … cannot be overwritten because the repository is immutable`: es la inmutabilidad haciendo su trabajo, no un bug.

En el curso: M5 · Registries y tags inmutables · M5 · Un Dockerfile de ML, decisión por decisión · M15 · Supply chain: imágenes, dependencias y pesos verificados · M10 · CI/CD para modelos: la pirámide de tests

10 CD: GitOps o kubectl, aprobación, concurrency y rollback con revert

La imagen ya está en ECR: ¿quién la lleva al cluster, quién aprueba, qué evita dos deploys a la vez y cómo vuelves atrás?

SINTAXIS CON VARIABLES

```
- id: app
  uses: actions/create-github-app-token@v3
  with: { client-id: "${{ vars.var_app }}",
private-key: "${{ secrets.secreto_app }}" }
- env: { IMG: "repo:tag@${{
needs.job_imagen.outputs.digest }}" }
  run: sed -i "s|repo[:@].*$|$IMG|" manifest
- uses: peter-evans/create-pull-
request@{sha_action}
  with: { token: "${{ steps.app.outputs.token }}",
branch: deploy/servicio, title: "deploy: servicio
tag" }
```

CD (continuous delivery, entrega continua) es llevar a producción, por etapas, lo que CI (las pruebas automáticas de cada cambio) ya probó. Hay dos caminos; este es el primero.

GitOps, lado de CI: GitOps quiere decir que el estado de producción vive en git: los manifests (los YAML de Kubernetes) dicen qué imagen corre. Después del push a ECR, el job cambia la línea `image:` del manifest: el tag dice de qué commit salió y el digest (la huella sha256 del contenido) fija los bytes exactos. Luego abre o actualiza un pull request con el token de una **GitHub App** (una identidad de máquina con permisos propios). Con ese token, los checks del PR corren solos. Desplegar es hacer merge de ese PR.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata: { name: app, namespace: argocd }
spec:
  project: default
  source: { repoURL: repo_git, targetRevision:
main, path: carpeta }
  destination: { server:
https://kubernetes.default.svc, namespace: ns }
  syncPolicy: { automated: { prune: true, selfHeal:
true } }
```

GitOps, lado del cluster. **Argo CD** corre dentro del cluster, compara lo que dice `main` con lo que corre y aplica la diferencia (un **sync**). `selfHeal` deshace los cambios hechos a mano en el cluster; `prune` borra lo que sacaste de git. Los dos vienen apagados si no los pones.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
var_app	variable del repo con el client ID de la GitHub App	`DEPLOY_APP_CLIENT_ID`
secreto_app	secret con la llave privada de la App (nunca en git)	`DEPLOY_APP_PRIVATE_KEY`
repo	el repositorio de ECR, sin el registry	`classifier-gateway`
tag	los 7 primeros caracteres del SHA del commit (el job los saca con ${GITHUB_SHA::7})	`9d41e7c`
registry	el registry de ECR de la cuenta	`123456789012.dkr.ecr.us-east-1.amazonaws.com`
digest	el digest que devolvió el push (output del job imagen)	`sha256:… (64 caracteres hex)`
contenedor	el contenedor del Deployment que cambia de imagen	`gateway`
job_imagen	el job que subió la imagen y publica su digest como output	`imagen`
manifest	el archivo con la línea image:	`reference/c01/k8s/gateway.yaml`
sha_action	el SHA completo de la versión de la action (fijada, sección 8)	`5f6978faf089d4d20b00c7766989d076bb2fc7f1 (v8.1.1)`
servicio	el Deployment que cambia	`gateway`
app	nombre de la Application de Argo CD	`classifier`
repo_git	URL del repo que lee Argo CD (si es privado, con una credencial de solo lectura)	`https://github.com/acme/classifier.git (ilustrativo)`
carpeta	carpeta de los manifests dentro del repo	`reference/c01/k8s`
ns	namespace de destino	`classifier`
entorno	environment de GitHub	`produccion`
plazo	cuánto espera rollout status antes de fallar	`10m (gateway) · 25m (Triton)`
sha_merge	el merge commit del PR del deploy	`el que muestra git log --merges -1`

```
deploy:
  needs: job_imagen
  environment: entorno
  concurrency: { group: deploy-servicio-entorno, cancel-in-progress: false }
  steps:  # antes: el rol de deploy por OIDC (sección 8), aws eks update-kubeconfig y un kubectl de la versión del cluster
    - run: kubectl -n ns set image deployment/servicio contenedor=registry/repo@{digest} && kubectl -n ns rollout status deployment/servicio --timeout=plazo
```

Estilo kubectl, el segundo camino y el de `gateway.yml`: el job empuja el cambio al cluster. Un **environment** (un entorno con nombre, con reglas y secretos propios) lo deja en **Waiting** hasta que apruebe alguien de la lista de **required reviewers**. `concurrency` deja un solo deploy a la vez en ese grupo. `set image` cambia solo la imagen, por digest; con `kubectl apply -f`, el manifest tiene que traer ya la imagen nueva. Si los pods nuevos no quedan listos, `rollout status` falla con `exceeded its progress deadline` (600 s sin avance, por defecto) y el job queda rojo; Kubernetes no vuelve atrás solo: los pods viejos siguen sirviendo.

```
git switch -c revert-deploy origin/main
git revert -m 1 sha_merge      # PR con merge commit; con squash: git revert sha
git push -u origin revert-deploy  # PR → merge → Argo CD aplica la versión anterior
```

Rollback = revert: un commit nuevo que deshace el del deploy. La historia queda completa: qué se desplegó, cuándo y cómo se volvió. Un merge commit tiene dos padres, así que `-m 1` dice que el lado bueno es `main`. El botón **Revert** de un PR mergeado hace lo mismo: abre un PR nuevo.

VARIABLE	QUÉ ES	EJEMPLO
sha	el commit único de un PR mergeado con squash	`el que muestra git log -1`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
GitOps	Los manifests de producción viven en git y un agente dentro del cluster aplica lo que dice main. Desplegar es un merge; volver, un revert.	`Argo CD o Flux (M10)`
Argo CD	Revisa el repo cada ≤ 3 min (120 s más hasta 60 s al azar), o al instante con un webhook (un aviso que GitHub le manda en cada push). Con automated, aplica las diferencias solo.	`Application → reference/c01/k8s`
automated · prune · selfHeal	automated: sincroniza cuando cambia git. prune: borra lo que sacaste de git. selfHeal: deshace los cambios a mano en el cluster. prune y selfHeal vienen apagados.	`un kubectl edit se revierte en segundos`
peter-evans/create-pull-request	Commitea lo que cambió el job en una rama y abre el PR. Si el PR ya existe, lo actualiza; si no hay cambios, no hace nada.	`branch: deploy/gateway`
PR abierto con GITHUB_TOKEN	El repo tiene que permitirlo (Allow GitHub Actions to create and approve pull requests, apagado por defecto en repos personales y en organizaciones nuevas). Desde junio de 2026, los workflows de ese PR esperan a que alguien con permiso de escritura (write) apriete Approve workflows to run.	`sin aprobación, el check obligatorio no corre`

PIEZA	QUÉ HACE	EJEMPLO
token de GitHub App · PAT	Con un token de GitHub App (o un personal access token, mejor de grano fino), el PR lo abre esa identidad y sus workflows corren solos. El token de la App dura 1 h y se revoca al terminar el job.	`actions/create-github-app-token@v3`
environment	Entorno con nombre, reglas y secretos propios. El job que lo nombra espera a que se cumplan las reglas; sus secretos llegan recién después de aprobar.	`environment: produccion`
required reviewers	Hasta 6 personas o equipos; con que una apruebe, el job sigue. Prevent self-review impide aprobar tu propio deploy. Sin respuesta en 30 días, el job falla.	`Review deployments → Approve and deploy`
environments según el plan	Required reviewers y wait timer (una espera fija antes de desplegar): en Free, Pro y Team solo funcionan en repos públicos; en privados hace falta Enterprise. Deployment branches (qué ramas pueden desplegar): también en privados con Pro o Team. En Free, los environments solo existen en repos públicos.	`tu repo de práctica: público`
concurrency	Grupo donde corre un solo job o workflow a la vez; el que llega después espera como pending. Es independiente del environment.	`group: deploy-gateway-produccion`
queue: single · queue: max	single (el default): espera uno solo; si llega otro, el que esperaba se cancela. max: hasta 100 en fila, en orden. max no se combina con cancel-in-progress: true.	`Canceling since a higher priority waiting request for '…' exists`
cancel-in-progress: true	Cancela también el que está corriendo. Bien para los tests de un PR; en un deploy puede dejarlo a medias: el job se corta entre pasos o antes de confirmar el rollout.	`group: pr-${{ github.head_ref || github.run_id }}`
kubectl desde el job	El runner necesita red hasta el API server de EKS (la puerta de entrada del cluster) y un rol de AWS con access entry (el permiso de EKS que deja entrar a ese rol). set image cambia solo la imagen (lo que hace gateway.yml); git ya no dice qué corre.	`kubectl -n classifier set image deployment/gateway gateway=… @sha256:…`
versión de kubectl	kubectl solo garantiza hablar con clusters de su versión menor ± 1 (la menor es el segundo número de la versión: el 37 de 1.37.1). El de ubuntu-24.04 es 1.37 (sirve para 1.36–1.38); con un cluster 1.35, el mínimo del C01, baja en el job el kubectl de esa versión.	`curl -LO https://dl.k8s.io/release/v1.35.9/bin/linux/amd64/kubectl`
git revert	Commit nuevo que deshace otro. Un merge commit necesita -m 1; un commit de squash, no. Con GitOps, ese revert es el rollback.	`Revert "Merge pull request #7 …"`

ESTILO GITOPS (EL DEL CURSO)

1 PR con el cambio
tú · GitHub
Rama, push y pull request a main.

2 Tests `≈ 1–2 min (est.)`
CI · ubuntu-24.04
pytest: los 24 tests del gateway, sin GPU. Es un check obligatorio.

3 Review y merge `minutos a horas`
el equipo · GitHub
Checks en verde y una aprobación; merge a main.

4 Build y push `≈ 2–5 min (est.)`
CI · ubuntu-24.04 → ECR
Rol por OIDC, buildx linux/amd64, push a ECR con tag = los 7 primeros caracteres del SHA, en un repo configurado con tags inmutables.

5 PR del manifest `≈ 10–30 s (est.)`
CI (GitHub App) · GitHub
Cambia la línea image: de reference/c01/k8s/gateway.yaml (tag y digest); sus checks corren solos.

6 Merge del manifest `lo que tarde la persona`
quien aprueba el deploy · GitHub
Aquí está la aprobación del estilo GitOps.

7 Sync de Argo CD
`≤ 3 min (polling) · segundos con webhook`
Argo CD · dentro del cluster
Ve el commit nuevo en main y aplica la diferencia.

8 Rollout `gateway ≈ 1–3 min (est.) · Triton ≈ 8–22 min`
Kubernetes · EKS
Pods nuevos antes de bajar los viejos (maxUnavailable: 0). Triton: 2 pods de a uno, cada uno con nodo GPU, imagen y engine (estimación del curso).

O, EN VEZ DE LOS PASOS 5–8: Estilo kubectl con environment

a Job deploy en Waiting
`hasta que alguien apruebe (falla a los 30 días)`
GitHub
environment: produccion con required reviewers; concurrency deja un deploy a la vez.

b kubectl set image y rollout status
`gateway ≈ 1–3 min (est.)`
CI
Por digest, como gateway.yml. El runner necesita red al API server de EKS, un rol con access entry y un kubectl de la versión del cluster (± 1 versión menor).

Rollback = revert

`gateway: minutos · Triton: ≈ 5–20 min si la release anterior ya no está viva (estimación del curso)`

1. git revert del merge del manifest (o el botón Revert del PR)
2. PR y merge
3. Argo CD sincroniza (≤ 3 min)
4. rollout a la imagen anterior

Con GitOps, un kubectl rollout undo dura hasta el próximo sync (o segundos, con selfHeal): el rollback de verdad es el revert.

tú · GitHub · automático (runner, Actions, Argo CD) · espera a una persona · AWS · Kubernetes

`≈ min` duración estimada

Un cambio del gateway del C01, de tu PR al pod nuevo, con duraciones estimadas. Arriba, el estilo GitOps del curso: la aprobación es el merge del PR del manifest y Argo CD despliega. Abajo, el estilo kubectl: el job espera en un environment hasta que alguien aprueba. Volver atrás recorre el mismo camino con un revert.

EJEMPLO REAL · C01: CI ABRE EL PR QUE DESPLIEGA EL GATEWAY (ESTILO GITOPS)

YAML — Ilustrativo, armado con los pasos del curso (M10 · GitOps para el despliegue; M05 · release.sh) sobre reference/c01/k8s/gateway.yaml. Validado con js-yaml contra la sintaxis de docs.github.com y los inputs del action.yml de cada action (octubre de 2026); el sed se probó sobre el manifest real.

```yaml
# .github/workflows/gateway.yml (extracto). Ilustrativo, armado con los pasos del curso: el C01 no trae workflows.
# … el job imagen (sección 9) sube classifier-gateway a ECR y publica outputs.digest
# Este job va en lugar del job deploy de gateway.yml (kubectl): mismo digest, camino GitOps.
  manifest-pr:                      # estilo GitOps: Argo CD aplica lo que llega a main
    needs: imagen
    if: github.event_name == 'push'  # solo después del merge a main, nunca en un PR
    runs-on: ubuntu-24.04
    timeout-minutes: 10
    permissions:
      contents: read                # el GITHUB_TOKEN solo lee: la rama y el PR los hace la GitHub App
```

```yaml
    concurrency: { group: deploy-gateway-pr, cancel-in-progress: false }
    steps:
      - uses: actions/checkout@v7
      - id: app                       # token de 1 h de la GitHub App de deploy, solo para este repo
        uses: actions/create-github-app-token@v3
        with:
          client-id: ${{ vars.DEPLOY_APP_CLIENT_ID }}
          private-key: ${{ secrets.DEPLOY_APP_PRIVATE_KEY }}
          permission-contents: write
          permission-pull-requests: write
      - id: tag
        run: echo "tag=${GITHUB_SHA::7}" >> "$GITHUB_OUTPUT"
      - name: Apuntar el manifest a la imagen nueva (tag para leer, digest para fijar)
        env:
          IMG: "classifier-gateway:${{ steps.tag.outputs.tag }}@${{ needs.imagen.outputs.digest }}"
        run: sed -i "s|classifier-gateway[:@].*$|${IMG}|" reference/c01/k8s/gateway.yaml
      - uses: peter-evans/create-pull-request@5f6978faf089d4d20b00c7766989d076bb2fc7f1 # v8.1.1
        with:
          token: ${{ steps.app.outputs.token }}  # con el token de la App, los checks del PR corren solos
          branch: deploy/gateway       # siempre la misma rama: si el PR ya existe, lo actualiza
          title: "deploy: gateway ${{ steps.tag.outputs.tag }}"
          commit-message: "deploy: gateway ${{ steps.tag.outputs.tag }}"
          body: "La imagen ya está en ECR. Al hacer merge, Argo CD la despliega; para volver atrás, revierte este PR."
```

- Cambia una línea: `…/classifier-gateway:3f9c2ab` pasa a `…/classifier-gateway:9d41e7c@sha256:…`. El tag dice de qué commit salió; el digest fija los bytes (con los dos, la imagen se baja por el digest).
- La aprobación del deploy es el review y el merge de este PR. Con un ruleset (las reglas de la rama main) que pide la aprobación de los code owners de `k8s/` (las personas o equipos que CODEOWNERS nombra como responsables), lo aprueba quien opera el servicio. Después, Argo CD aplica `main` en ≤ 3 min, o en segundos con un webhook.
- Por qué una GitHub App y no `GITHUB_TOKEN`: con `GITHUB_TOKEN`, el repo tiene que activar **Allow GitHub Actions to create and approve pull requests** (apagado por defecto en repos personales y en organizaciones nuevas) y, desde junio de 2026, los workflows de ese PR quedan esperando a que alguien con permiso de escritura (write) apriete **Approve workflows to run**. Si nadie lo hace, el check obligatorio nunca corre y el merge queda bloqueado. Con el token de la App, el PR lo abre la App y sus checks corren solos.
- `concurrency` con la cola por defecto (`queue: single`): si se juntan varios merges, solo espera el más nuevo. Los que esperaban antes se cancelan y el PR queda con la imagen más nueva. Aquí es justo lo que quieres.
- Cuidado con el bucle: si el workflow que construye la imagen corre con cualquier push a `main`, el merge de este PR dispara otro build y otro PR. Filtra el `push` con `paths` (solo el código del gateway) o pon los manifests en otro repo; la App necesita acceso a ese repo (`GITHUB_TOKEN` solo sirve en el repo del workflow).
- La action de terceros va fijada al SHA completo de v8.1.1, como recomienda la guía de seguridad de GitHub (sección 8); las de `actions/`, que son de GitHub, van a su major.

CÓMO RECORDARLA

CI construye, git decide, el cluster copia. Desplegar es un merge que cambia una línea; volver es un revert de ese merge. El environment pone a una persona en el medio y `concurrency` deja un solo deploy a la vez: si se juntan varios, solo espera el más nuevo y los demás se cancelan.

TRAMPA

arreglar producción con `kubectl set image` o `kubectl rollout undo` cuando hay GitOps. Dura poco: con `selfHeal`, Argo CD vuelve a lo que dice git en segundos, y sin él, en el próximo sync. El rollback de verdad es un `git revert` del commit del deploy. Y nunca `cancel-in-progress: true` en un job de deploy: el job se corta a mitad de camino y nadie confirma qué quedó corriendo.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M11 · Canary: porcentajes, tiempos y criterios para avanzar · M11 · Rollback: criterios automáticos y qué guardar para volver · M5 · Registries y tags inmutables · Docker y kubectl · Rollouts y escalado · C01 · Modos de falla y degradación

11 Runners: GitHub-hosted o self-hosted (la T4 del C01), labels, timeouts y costos

¿En qué máquina corre cada job, cuánto puede durar, cuánto cuesta y cuándo conviene una máquina propia, como la T4 del C01?

SINTAXIS CON VARIABLES

```
jobs:
  job:
    runs-on: imagen
    timeout-minutes: min
```

Un **runner** es la máquina que ejecuta un job. Con una imagen de GitHub (**GitHub-hosted**), cada job recibe una VM (máquina virtual) nueva. `timeout-minutes` corta el job si se pasa: el default es 360 min (6 h), que también es el máximo en GitHub-hosted.

```
runs-on: [self-hosted, so, arq, label]
```

Un runner **self-hosted** es una máquina tuya con el programa del runner instalado. Las **labels** son etiquetas: el job va a un runner que tenga **todas** las que pide. `self-hosted`, el sistema y la arquitectura las pone el runner solo; las demás son tuyas.

```
./config.sh --url https://github.com/org/repo
--token token_registro --labels label --ephemeral
--unattended
./run.sh
```

Registrar el runner, en la máquina. El token de registro sale de Settings → Actions → Runners → New self-hosted runner (o de la API) y vence en 1 h. Con `--ephemeral`, el runner toma **un** job y se desregistra: la base para crear una máquina por job y apagarla al final.

```
runs-on:
  group: grupo
  labels: [label]
```

En una organización, los runners se juntan en **runner groups** (grupos con su lista de repos permitidos). Con `group` y `labels`, el runner tiene que cumplir las dos cosas. Un **larger runner** de GitHub (más grande, o con GPU) se elige por su grupo o por su nombre, que es su label.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
job	nombre del job	`tests`
imagen	imagen de GitHub; fíjala en vez de -latest	`ubuntu-24.04`
min	minutos antes de que GitHub cancele el job	`10 (tests) · 120 (engine de Triton)`
so	sistema operativo: label que pone el runner	`linux`
arq	arquitectura: label que pone el runner	`x64`
label	label propia del runner	`gpu-t4`
org / *repo*	dónde se registra el runner	`acme/classifier (ilustrativo)`
token_registro	token de registro: lo genera GitHub, vence en 1 h; nunca va a git	`Settings → Actions → Runners`
grupo	runner group de la organización	`gpu`
t_{job}	minutos de cada job; se redondean hacia arriba	`2 min 10 s → 3`
incluidos	minutos del plan para repos privados, según el plan; en un repo público, los runners estándar no cobran	`2,000 Free · 3,000 Pro y Team · 50,000 Enterprise Cloud`
p	precio por minuto del runner (los larger runners no usan los incluidos)	`US$0.006 Linux 2-core · US$0.052 GPU T4`
$t_{\text{máquina}}$	tiempo que la máquina propia está prendida (job más arranque)	`≈ 45 min por corrida (est.)`
p_{EC2}	precio on-demand de la máquina	`g4dn.xlarge ≈ US$0.526/h (estimación del curso)`

FÓRMULAS

$$\text{minutos} = \sum_{\text{jobs}} \left\lceil t_{\text{job}} \right\rceil$$

$$\text{costo}_{\text{GitHub}} = \max(0,\ \text{minutos} - \text{incluidos}) \cdot p \qquad \text{costo}_{\text{self-hosted}} = t_{\text{máquina}} \cdot p_{\text{EC2}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
ubuntu-24.04 · ubuntu-latest	Imágenes de GitHub, con Docker, kubectl, AWS CLI, gh y varias versiones de Python ya instalados (la lista está en actions/runner-images y cambia con cada imagen). ubuntu-latest hoy es 24.04 y se mueve con aviso, en una migración de 1–2 meses: fija la versión.	`runs-on: ubuntu-24.04`
ubuntu-24.04-arm	Runner estándar arm64 (procesador ARM, como el de tu Mac). Gratis en repos públicos, como los x64; en privados también existe y gasta minutos.	`uname -m → aarch64`
runner estándar	Repo privado: 2 vCPU, 8 GB de RAM y 14 GB de disco. Repo público: 4 vCPU y 16 GB.	`nproc · free -h · df -h /`
larger runner	Más CPU, RAM y disco, o GPU: 1 T4 de 16 GB con 4 vCPU, 28 GB y 176 GB. Solo en Team o Enterprise Cloud; se cobra siempre, aunque el repo sea público.	`GPU Linux: US$0.052/min`
self-hosted	Tu máquina con el programa del runner. Sale por HTTPS 443 a GitHub y pregunta si hay trabajo: no abre puertos de entrada.	`la g4dn con T4 del C01`
labels por defecto	self-hosted, el sistema (linux, windows, macOS) y la arquitectura (x64, ARM, ARM64). No distinguen mayúsculas.	`[self-hosted, linux, x64, gpu-t4]`
runner group	Grupo de runners de una organización, con la lista de repos que pueden usarlo.	`runs-on: { group: gpu }`
--ephemeral · JIT	El runner corre un job y se desregistra. JIT (just-in-time) hace lo mismo con una configuración que pides a la API.	`./config.sh … --ephemeral`
ARC	Actions Runner Controller: un programa que corre dentro de Kubernetes y crea runners como pods según los jobs en cola.	`runners efímeros en EKS`
timeout-minutes	Corta el job (o el paso) a los N minutos. Default 360. Tope por job: 6 h en GitHub-hosted, 5 días en self-hosted.	`10 en tests, 120 en el engine`
en cola (queued)	Si ningún runner libre tiene esas labels: Waiting for a runner to pick up this job…; con self-hosted, a las 24 h en cola, el job falla.	`runner apagado o labels mal escritas`
minutos facturados	Cada job se redondea al minuto de arriba. Repos públicos con runners estándar: gratis. Privados: los incluidos del plan y después por minuto. Self-hosted: GitHub no cobra minutos.	`Linux 2-core: US$0.006/min`
jobs en paralelo	Según el plan: 20 en Free, 40 en Pro, 60 en Team, 500 en Enterprise (macOS: 5, o 50 en Enterprise). El resto espera en cola.	`una matrix grande hace fila`
self-hosted en un repo público	Casi nunca: cualquiera abre un PR y puede correr código en tu máquina, que no se limpia sola entre jobs.	`solo en repos privados`

GitHub-hosted

`runs-on: ubuntu-24.04`

`EN EL C01:` tests e imagen del gateway (gateway.yml)

Self-hosted

`runs-on: [self-hosted, linux, x64, gpu-t4]`

`EN EL C01:` engine e imagen de Triton en la g4dn con T4 (triton-modelo.yml)

	GITHUB-HOSTED	SELF-HOSTED
Quién pone y mantiene la máquina	GitHub	Tú: sistema, driver de NVIDIA, Docker, parches y el runner
Máquina limpia en cada job	Sí: una VM nueva por job	Solo si es efímera (--ephemeral o JIT) y la recreas

	GITHUB-HOSTED	SELF-HOSTED
Hardware	x64 o arm64; privado: 2 vCPU, 8 GB, 14 GB de disco · GPU T4 solo como larger runner (Team o Enterprise Cloud)	Lo que elijas: una g4dn.xlarge con 1 T4
Costo	Gratis en repos públicos (estándar); en privados, los minutos incluidos según el plan y después por minuto	GitHub no cobra minutos; pagas la máquina (≈ US$0.526/h on-demand)
Tope por job	6 h	5 días (y 24 h en cola)
Red	Internet, con IPs que cambian	La de tu VPC: llega a recursos privados
Riesgo con PRs de afuera	**Bajo**: la VM se descarta	**Alto**: un PR puede correr código en tu máquina; nunca en repos públicos

la máquina es de GitHub la máquina es tuya

Los dos tipos de runner, lado a lado, con lo que usa el C01 en cada uno. Precios y límites de docs.github.com, octubre de 2026; el precio de la g4dn es la estimación del curso.

EJEMPLO REAL · C01: QUÉ JOB CORRE EN QUÉ MÁQUINA

YAML — Ilustrativo, armado con los pasos del curso: reference/c01/README.md (engine e imagen en una g4dn.xlarge con T4), M10 · CI/CD (runner con T4 que se prende solo para el job) y C01 · costo. Specs, límites y precios de docs.github.com (octubre de 2026).

```yaml
# Extractos de gateway.yml y triton-modelo.yml. Ilustrativos, armados con los pasos del curso: el C01 no trae workflows.
# gateway.yml: máquinas de GitHub, una VM nueva por job
jobs:
  tests:
    runs-on: ubuntu-24.04          # x64: linux/amd64 sin emulación; fija la versión, no ubuntu-latest
    timeout-minutes: 10            # 24 tests tardan segundos: 10 min ya es una falla
    # …
  imagen:
    runs-on: ubuntu-24.04
    timeout-minutes: 20
    # …
---
# triton-modelo.yml: la T4 propia, a mano o desde el pipeline del modelo
on:
  workflow_dispatch:               # nunca pull_request: un PR no corre código en la T4
jobs:
  engine-e-imagen:
    runs-on: [self-hosted, linux, x64, gpu-t4]   # todas las labels: solo la g4dn las tiene
    timeout-minutes: 120           # sin esto, un trtexec colgado retiene la T4 los 360 min del default
    # … build_engine.sh, docker build triton/, paridad, push y registro
```

- `tests` e `imagen`: una VM nueva por job. En un repo privado tiene 2 vCPU, 8 GB de RAM y 14 GB de disco (4 vCPU y 16 GB si el repo es público). Es x64, así que `linux/amd64` se construye sin emulación.
- Lo decisivo: el engine de TensorRT solo se construye en una T4, y un runner estándar no tiene GPU. Además, la imagen de Triton pesa ≈ 9.2 GB comprimida (cifra del módulo 5) y bastante más en disco: muy justa para los 14 GB que garantiza un runner estándar.
- La g4dn se registra como runner **efímero** (`--ephemeral`): toma un job, se desregistra y la apagas. Para prenderla cuando hay un job en cola: ARC (en Kubernetes) o tu propia automatización con el webhook `workflow_job`.
- Costo de una corrida de 40 min (est.): la g4dn on-demand ≈ US$0.526/h → ≈ US$0.35, más el arranque. El runner GPU de GitHub (también una T4) → 40 × US$0.052 = US$2.08, sin máquina que mantener, pero solo en Team o Enterprise Cloud. El curso estima menos de US$10 al mes para este runner.
- Solo `workflow_dispatch`: la T4 nunca corre código de un PR. En un repo público, ni pensarlo: cualquiera puede abrir un PR y una máquina persistente guarda lo que el job anterior dejó.
- `timeout-minutes` en cada job. En self-hosted podrías subirlo hasta 5 días, pero el `GITHUB_TOKEN` se renueva solo hasta 24 h (en GitHub-hosted dura lo que el job, como mucho 6 h).

CÓMO RECORDARLA

labels eligen, timeout corta, minutos cuestan. GitHub-hosted: una VM limpia por job, gratis en repos públicos. Self-hosted: tu hardware (la T4), tu mantenimiento y tu riesgo; efímero, por `workflow_dispatch` y nunca en un repo público.

TRAMPA

un runner self-hosted persistente que corre jobs de `pull_request`, o cualquiera en un repo público. El código de un PR corre en tu máquina, que no se limpia entre jobs: puede leer credenciales del disco o dejar algo instalado para el próximo job. Y `ubuntu-latest` se mueve: cuando pase a otra versión de Ubuntu, cambian Python, compiladores y librerías sin que toques el YAML.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · C01 · Costo y cómo reducirlo · M8 · TensorRT · M13 · Instancias GPU y costo por hora · Herramientas · Infraestructura como código, CI y seguridad

M15 · Supply chain: imágenes, dependencias y pesos verificados

12 Diagnóstico: leer el log, re-run con debug, reproducir en local, tipos de falla

El check se puso rojo: ¿dónde está el error de verdad, qué tipo de falla es, cómo la reproduces en tu máquina y cuándo sirve un re-run?

SINTAXIS CON VARIABLES

`gh run list --workflow` *workflow* `--branch` *rama* `--limit 5`

`gh run view` *run_id* `--log-failed`

Leer el log: run rojo → job rojo → paso rojo. El error de verdad está **arriba** de `Process completed with exit code N` (N es el código con que terminó el comando; 0 es éxito) y casi siempre es el primero, no el último. `--log-failed` muestra solo los pasos que fallaron; sin `gh`, lo mismo en la pestaña Actions.

`gh run rerun` *run_id* `--failed --debug`

Re-run: vuelve a correr lo que falló con el mismo commit (mismo `GITHUB_SHA` y `GITHUB_REF`). `--debug` (en la web, **Enable debug logging**) suma el log detallado de cada paso y del runner. Si falla igual, no fue mala suerte.

`git clone --branch` *rama* *url_repo* `/tmp/ci && cd /tmp/ci/` *dir*

`python` *version* `-m venv .venv && . .venv/bin/activate`
`pip install -r requirements-dev.txt && pytest`

Reproducir en local: un clon limpio ve lo mismo que el checkout de CI (solo lo commiteado) y un venv (un entorno de Python aislado) nuevo tiene solo lo que dicen los requirements. Si ahí falla igual, ya no adivinas: lo arreglas en tu máquina.

`git bisect start` *malo* *bueno*
`git bisect run` *prueba*
`git bisect reset`

¿Qué commit lo rompió? `bisect` parte la historia a la mitad y corre tu prueba en cada paso: exit 0 = bueno; de 1 a 127 (menos 125) = malo; 125 = no se puede probar, sáltalo. `reset` te devuelve a tu rama.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
workflow	el archivo del workflow	`gateway.yml`
rama	la rama del PR	`feat/json-rapido (ilustrativo)`
run_id	número del run: sale de gh run list o de la URL del run	`1234567890`
url_repo	URL del repo, la misma que usas para clonar	`git@github.com:acme/classifier.git (ilustrativo)`
dir	la carpeta donde el workflow corre los tests	`reference/c01/gateway`
version	la misma versión de Python que pide el workflow	`3.13`
malo	un commit donde falla	`HEAD`
bueno	un commit donde pasaba (el del último run verde)	`un SHA o un tag`
prueba	comando que sale con 0 si todo está bien	`pytest -q tests/test_api.py`
n	commits entre el bueno y el malo	`100 → 7 pruebas; 1,000 → 10`
$n_{\text{señal}}$	número de la señal que mató al proceso	`9 = SIGKILL (casi siempre falta de memoria) · 15 = SIGTERM`

FÓRMULAS

$$\text{pruebas de bisect} \approx \lceil \log_2 n \rceil$$

$$\text{exit code} = 128 + n_{\text{señal}} \qquad 137 = 128 + 9 \ (\text{SIGKILL})$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Annotations	El resumen de errores y warnings arriba del run, con enlace al paso. Sirve para ubicarte; el detalle está en el log.	`Error: Process completed with exit code 1.`
Process completed with exit code N	El paso terminó con un código distinto de 0. Es la consecuencia: la causa está en las líneas de arriba.	`N = 1, 2, 127, 137…`
pytest: exit 1 · 2 · 4 · 5	1: fallaron tests (busca FAILED). 2: interrumpido, también por un error al importar los tests. 4: flags mal usados. 5: no encontró tests.	`1 failed, 23 passed`
exit 127 · 137	127: el shell no encontró el comando (no está instalado o no está en el PATH). 137 = 128 + 9: el proceso murió por SIGKILL, casi siempre por memoria.	`command not found`

PIEZA	QUÉ HACE	EJEMPLO
bash -e (el default de run)	El paso corta en el primer comando que falla, pero no mira dentro de un pipe (comando1 | comando2): solo cuenta cómo terminó el último. Con shell: bash suma -o pipefail.	`pytest | tee log.txt pasa aunque fallen tests`
Re-run failed jobs	Mismo commit y mismo GITHUB_REF; hasta 30 días después y hasta 50 veces; con los permisos de quien disparó el run original.	`gh run rerun ID --failed`
ACTIONS_STEP_DEBUG · ACTIONS_RUNNER_DEBUG	Secret o variable en true: log detallado en todos los runs, no solo en un re-run. El del runner viene en la carpeta runner-diagnostic-logs al bajar los logs.	`aparecen las líneas ##[debug]…`
::group:: · ::error file=…,line=…::	Agrupa líneas del log y crea anotaciones que apuntan a un archivo y una línea.	`echo "::error file=app/main.py,line=12::falta MODEL_NAME"`
$GITHUB_STEP_SUMMARY	Markdown que aparece en la página del run (hasta 1 MiB por paso). Ideal para el resumen de los tests.	`echo "### 24 tests ok" >> "$GITHUB_STEP_SUMMARY"`
fetch-depth: 1 (el default)	actions/checkout baja solo el último commit: git log, git describe o un diff contra main no encuentran historia. fetch-depth: 0 baja todo. En un PR, además, queda en detached HEAD (HEAD apunta a un commit, no a una rama).	`with: { fetch-depth: 0 }`
clon limpio	Reproduce el checkout de CI: solo lo commiteado, sin archivos ignorados ni archivos a los que nunca les hiciste add.	`git clone --branch RAMA URL /tmp/ci`
retención	Logs y artifacts se borran a los 90 días por defecto (1–90 en repos públicos, 1–400 en privados). Desde el 1 de octubre de 2026, la misma regla borra también runs, checks y statuses. Guarda lo que necesites antes.	`Settings → Actions → General`
test flaky	Pasa y falla con el mismo código: orden, tiempo, red, seeds. Un re-run lo tapa, no lo arregla.	`falla 1 de cada 10 runs`
githubstatus.com	Si falla algo que no tocaste (bajar actions, caché, artifacts), y le pasa a otros repos, mira el estado de GitHub antes de depurar.	`incidente en Actions`

TIPO DE FALLA	CÓMO SE VE	DÓNDE MIRAR PRIMERO	ARREGLO TÍPICO
El workflow no arranca	No aparece el run, o aparece con Invalid workflow file y ningún job	on: del YAML; que el archivo esté en .github/workflows en ese commit; la línea y columna del error	Corregir el YAML o el trigger (sección 6)
Configuración de una action	Unable to resolve action · Input required and not supplied · Unexpected input(s)	El primer paso rojo, al principio del job	Versión que exista e inputs con el nombre exacto (sección 6)
Tu código o tus tests	FAILED tests/… y Process completed with exit code 1	El resumen de pytest, arriba del exit code	Reproducir con un clon limpio y arreglar
Entorno distinto	ModuleNotFoundError, FileNotFoundError, otra versión de Python	Clon limpio y venv nuevo; python --version en el log	Versiones fijas; commitear o generar el archivo
Permisos o credenciales	Resource not accessible by integration · Could not assume role with OIDC · no basic auth credentials	permissions: del job; la trust policy del rol; la región	El permiso mínimo que falta (sección 8) o el login a ECR (sección 9)
Recursos del runner	exit code 137 · No space left on device · lost communication with the server · exceeded the maximum execution time	free -h y df -h en un paso; la duración del job	Menos memoria o disco, otro runner o un timeout realista (sección 11)
Red o servicio externo	Timeouts y errores 5xx de un registry o de PyPI	¿Falla igual al re-correr? githubstatus.com	Re-run failed jobs; si se repite, no es mala suerte
Test flaky	Pasa y falla con el mismo commit	El historial del check en varios runs	Arreglar la causa (orden, tiempo, red, seed), no re-run infinito

Las primeras dos se ven en segundos y están en el YAML, no en tu código; permisos, recursos y red dependen de la configuración, del runner o de afuera; código, entorno y tests flaky se reproducen y se arreglan en tu máquina.

Ocho tipos de falla, más o menos en el orden en que aparecen en un run. Cada fila: cómo se ve, dónde mirar primero y el arreglo típico.

EJEMPLO REAL · C01: PYTEST PASA EN TU MAC Y FALLA EN CI

BASH · ZSH

Ilustrativo, con los nombres del C01 (reference/c01/gateway, pytest.ini con -q, pytest==9.1.1): la dependencia orjson, la rama y el número de run son inventados para el ejemplo. Salidas abreviadas, con la forma de gh 2.x y pytest 9.

```
# El check "tests" del PR se puso rojo; en tu Mac, pytest pasa. 1) Leer el log del paso que falló
gh run list --workflow gateway.yml --branch feat/json-rapido --limit 3
gh run view 1234567890 --log-failed | tail -n 6
# tests  Run pytest  E   ModuleNotFoundError: No module named 'orjson'
# tests  Run pytest  =========================== short test summary info ============================
# tests  Run pytest  ERROR tests/test_api.py
# tests  Run pytest  !!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
# tests  Run pytest  1 error in 0.52s
# tests  Run pytest  ##[error]Process completed with exit code 2.

# 2) Reproducir como CI: clon limpio y venv nuevo, solo con lo que dicen los requirements
git clone -q --branch feat/json-rapido git@github.com:acme/classifier.git /tmp/ci
cd /tmp/ci/reference/c01/gateway
python3.13 -m venv .venv && . .venv/bin/activate
pip install -q -r requirements-dev.txt && pytest
# … el mismo ModuleNotFoundError: en tu venv de siempre, orjson estaba instalado a mano

# 3) Confirmarlo: la dependencia no está fijada en requirements.txt
grep -n orjson requirements.txt || echo "falta orjson en requirements.txt"
# falta orjson en requirements.txt → agrégala con == (la imagen también la necesita), commit y push
```

- Primero el run y el paso; después, el texto. `--log-failed` muestra solo los pasos rojos. La causa es la línea con `E` (el error de Python), no el exit code del final.
- `exit code 2` con `Interrupted: 1 error during collection`: pytest ni empezó a correr tests; falló al importar `tests/test_api.py`, que importa `app.main`. Un 1 habría sido un test que falló.
- El clon limpio y el venv nuevo son el checkout y el `pip install` del workflow. En tu carpeta de siempre pasaba porque orjson ya estaba instalado. **Pasa en mi máquina** casi siempre es esto, o un archivo que nunca se commiteó (el laboratorio).
- Un re-run no habría servido: mismo commit, mismo resultado. Re-run es para lo intermitente (la red, un registry caído) y conviene con `--debug`.
- Para Linux como en CI (el disco de tu Mac no distingue mayúsculas por defecto, y Linux sí), corre lo mismo en un contenedor: `docker run --rm -v "$PWD":/src -w /src python:3.13-slim sh -c "pip install -r requirements-dev.txt && pytest"`.

CÓMO RECORDARLA

leer → clasificar → reproducir → arreglar. El log dice dónde; el tipo de falla dice quién la arregla; el clon limpio dice si es tu código; el re-run solo sirve para lo intermitente.

TRAMPA

re-run hasta que pase. Con el mismo commit, una falla de tu código vuelve igual, y si a veces pasa es un test flaky que el re-run esconde. Y no te quedes con la última línea: `Process completed with exit code 1` es la consecuencia; la causa está arriba. Ojo también con un pipe: `pytest | tee log.txt` en un `run` sin `shell: bash` deja el paso en verde aunque fallen tests.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M10 · Reproducibilidad: los cinco ingredientes · C01 · Archivos de referencia · C01 · Modos de falla y degradación

13 Errores que debes reconocer rápido (git y Actions)

Lees el mensaje: ¿quién lo dijo (git, GitHub, el runner, una action o AWS), qué significa, qué corres primero y dónde está la explicación?

SINTAXIS CON VARIABLES

`fatal:` *mensaje*
`error:` *mensaje*
`hint:` *consejo*

Git, en tu máquina. `fatal` : abortó el comando. `error` : algo falló. `hint` : un consejo, casi siempre con el comando que lo arregla. Lee las tres líneas completas.

`remote:` *mensaje*
`! [remote rejected]` *rama* **`->`** *rama* **`(`** *motivo* **`)`**

GitHub, durante un push o un pull: reglas del repo, permisos, autenticación o un secreto detectado. Las líneas `remote:` dicen cuál regla; `[remote rejected]` es el resultado.

`Error:` *mensaje*
`##[error]` *mensaje*

El runner o una action, en el log de Actions. Aparece también en **Annotations**. `Process completed with exit code N` es la consecuencia; la causa está en las líneas de arriba.

`Could not assume role with OIDC:` *motivo*
`An error occurred (` *codigo* **`) when calling the`** *operacion* **`operation:`** *mensaje*

AWS. La primera forma la imprime `configure-aws-credentials` con lo que respondió AWS STS (el servicio que entrega credenciales temporales); la segunda, la AWS CLI dentro de un `run` .

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
mensaje	el texto que explica qué pasó	`Need to specify how to reconcile divergent branches.`
consejo	la sugerencia de git, con el comando	`use 'git pull' before pushing again.`
rama	la rama que empujabas	`main`
motivo	la razón corta del rechazo o del fallo	`pre-receive hook declined · Not authorized to perform sts:AssumeRoleWithWebIdentity`
codigo	el código de error de AWS	`AccessDenied, ExpiredToken`
operacion	la operación de AWS que falló	`AssumeRoleWithWebIdentity, GetAuthorizationToken`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`fatal:`	Git abortó el comando: no hizo el cambio. La misma línea dice por qué.	`fatal: Need to specify how to reconcile divergent branches.`
`error: · hint:`	error: algo falló. hint: un consejo con el comando que lo arregla.	`hint: use 'git pull' before pushing again.`
`! [rejected]`	Tu push no avanza el remoto: (fetch first) o (non-fast-forward). Integra primero.	`git pull --rebase`
`! [remote rejected] · remote:`	Lo rechazó GitHub, no Git: una regla del repo, permisos o un secreto.	`remote: error: GH013 …`
`CONFLICT (…)`	Git no pudo juntar dos cambios. content: mismas líneas. modify/delete: uno borró y el otro editó.	`CONFLICT (modify/delete): … deleted in otra and modified in HEAD.`
`Warning: · Note:`	Avisos: no fallaron, pero te dicen algo que puedes perder (commits sueltos, detached HEAD).	`Warning: you are leaving 1 commit behind`
`Invalid workflow file · The workflow is not valid`	El YAML no valida: el run no arranca. El mensaje trae línea y columna.	`(Line: 12, Col: 5): Unexpected value 'runs_on'`
`Error: (anotación)`	Un paso falló o una action llamó a setFailed: aparece en Annotations y en el log.	`Error: Input required and not supplied: aws-region`
`Warning: Unexpected input(s)`	Pasaste en with: un input que la action no declara (un typo). Se ignora sin fallar.	`Unexpected input(s) 'python_version', valid inputs are […]`
`Process completed with exit code N`	El paso terminó con N distinto de 0: mira arriba.	`exit code 2 en pytest: error al importar los tests`
`Could not assume role with OIDC:`	configure-aws-credentials no consiguió el rol; lo que sigue lo dijo AWS STS.	`Not authorized to perform sts:AssumeRoleWithWebIdentity`

PIEZA	QUÉ HACE	EJEMPLO
`An error occurred (…) when calling the … operation:`	La AWS CLI dentro de un run. El código entre paréntesis dice el tipo de error.	**`(AccessDenied) when calling the AssumeRoleWithWebIdentity operation`**
`no basic auth credentials · tag invalid:`	Salen en un docker push a ECR. El primero lo dice docker: no tiene un login válido para ese registry. El segundo lo responde ECR: ese tag ya existe en un repo configurado con tags inmutables.	**`no basic auth credentials`**
`Waiting for … · Canceling since …`	No es un error de tu código: GitHub espera un runner, una aprobación o su turno en un grupo de concurrency.	**`Waiting for a runner to pick up this job...`**

LOS 26 MENSAJES

`fatal: not a git repository (or any of the parent directories): .git`

Corriste git en una carpeta que no es parte de un repo: ni ella ni las de arriba tienen .git. Pasa en una terminal nueva o fuera del proyecto.

PRIMERO: `cd ~/code/classifier && git status` (entra a la carpeta del repo)

→ Sección 2 · El día a día: clone, status, add, commit, diff, log, push, pull, fetch y .gitignore

`fatal: pathspec 'gateway/app/mian.py' did not match any files`

git add no encontró ese archivo: la ruta tiene un error de tipeo o estás en otra carpeta (las rutas son relativas a donde estás).

PRIMERO: `git status --short` (los archivos con cambios, con su ruta real)

→ Sección 2 · El día a día: clone, status, add, commit, diff, log, push, pull, fetch y .gitignore

`! [rejected] main -> main (fetch first) · ! [rejected] main -> main (non-fast-forward)`

El remoto tiene commits que tú no tienes: alguien empujó antes, y Git no deja que tu push los borre. fetch first: todavía no los bajaste; non-fast-forward: ya los bajaste, pero tu rama no los incluye.

PRIMERO: `git pull --rebase` (o --no-rebase), corre los tests y git push

→ Sección 2 · El día a día: clone, status, add, commit, diff, log, push, pull, fetch y .gitignore

`fatal: Need to specify how to reconcile divergent branches.`

Tu rama y la remota tienen commits distintos (divergieron) y no configuraste cómo juntarlas al hacer pull. Git no elige por ti: merge o rebase.

PRIMERO: `git pull --rebase` (o git config pull.rebase true para que sea tu default)

→ Sección 2 · El día a día: clone, status, add, commit, diff, log, push, pull, fetch y .gitignore

`CONFLICT (content): Merge conflict in k8s/gateway.yaml · Automatic merge failed; fix conflicts and then commit the result.`

Las dos ramas cambiaron las mismas líneas y Git no sabe cuál dejar. Marcó el archivo con <<<<<<<, ======= y >>>>>>>: eliges tú, guardas y haces add.

PRIMERO: `git status` (los archivos en both modified)

→ Sección 3 · Ramas, merge y rebase; conflictos y cómo resolverlos

`error: the branch 'fix/cpu' is not fully merged`

git branch -d no borra una rama cuyos commits no llegaron a tu rama actual (o a su rama remota, si la sigue): los perderías. Pasa, por ejemplo, después de un Squash and merge en GitHub: main recibió un commit nuevo con tus cambios, no tus commits.

PRIMERO: `git log --oneline main..fix/cpu` (qué commits tiene la rama que main no); si ya entraron con el squash, git branch -D fix/cpu

→ Sección 3 · Ramas, merge y rebase; conflictos y cómo resolverlos

`error: Your local changes to the following files would be overwritten by checkout: · Please commit your changes or stash them before you switch branches.`

Tienes cambios sin commit en un archivo que la otra rama tiene distinto: cambiar de rama los pisaría, así que Git no cambia. git switch también dice checkout.

PRIMERO: `git stash push -m "wip"` (y, ya en la otra rama, git stash pop)

→ Sección 4 · Deshacer: restore, reset, revert, amend, stash y reflog

`Warning: you are leaving 1 commit behind, not connected to any of your branches:`

Hiciste commits en detached HEAD y te fuiste a una rama: esos commits quedan sueltos. No se borraron: el aviso trae el SHA y el reflog los guarda unos 30 días (lo que ninguna rama alcanza vence antes que el resto).

PRIMERO: `git branch rescate SHA_DEL_AVISO`

→ Sección 4 · Deshacer: restore, reset, revert, amend, stash y reflog

`fatal: The current branch fix/timeout has no upstream branch.`

Tu rama nueva todavía no existe en el remoto y no sabe a qué rama empujar. El primer push la crea allá y la enlaza (su upstream).

PRIMERO: `git push -u origin fix/timeout`

→ Sección 5 · GitHub: remotos, acceso, pull requests y reglas de la rama

`remote: error: GH013: Repository rule violations found for refs/heads/main.`

GitHub rechazó el push por una regla del repo (un ruleset): por ejemplo, main solo acepta cambios por PR con checks en verde, o push protection (el detector de secretos de GitHub) encontró un secreto. Las líneas remote: que siguen dicen qué regla. GH006 es el mismo rechazo con la protección de ramas clásica.

PRIMERO:

```
git switch -c fix/mi-cambio && git push -u origin fix/mi-cambio
```

(y abre un PR)

→ Sección 5 · GitHub: remotos, acceso, pull requests y reglas de la rama

---

`remote: Support for password authentication was removed on August 13, 2021. · fatal: Authentication failed for 'https://github.com/…'`

Usaste tu contraseña de GitHub cuando git la pidió por HTTPS. GitHub ya no la acepta para git: usa SSH, un personal access token con un credential helper, o gh auth login.

PRIMERO: `git remote -v` (si dice https://, entra con gh auth login o un token; si prefieres SSH, cambia la URL)

→ Sección 5 · GitHub: remotos, acceso, pull requests y reglas de la rama

---

`git@github.com: Permission denied (publickey). · fatal: Could not read from remote repository.`

GitHub no aceptó ninguna llave SSH tuya: no tienes una cargada, no la subiste a tu cuenta o se ofrece otra. Could not read from remote repository también sale si el repo no existe o no tienes acceso.

PRIMERO: `ssh -T git@github.com` (si todo está bien, responde Hi TU_CUENTA! …)

→ Sección 5 · GitHub: remotos, acceso, pull requests y reglas de la rama

---

`The workflow is not valid. .github/workflows/gateway.yml (Line: 12, Col: 5): Unexpected value 'runs_on'`

GitHub no pudo validar el YAML: una clave mal escrita o en el nivel equivocado (runs_on en vez de runs-on), o mala indentación. El run falla al arrancar y no corre ningún job.

PRIMERO: `git diff HEAD~1 -- .github/workflows/gateway.yml` (qué cambió en el YAML)

→ Sección 6 · Anatomía de un workflow de Actions

`Unable to resolve action `actions/checkout@v9`, unable to find version `v9`

La action existe, pero esa versión (tag o rama) no. Pasa con un error en el número, o al copiar un ejemplo que pide una versión que no existe.

PRIMERO:

```
git ls-remote --tags https://github.com/actions/checkout
```

(los tags que existen)

→ Sección 6 · Anatomía de un workflow de Actions

`The version '3.1' with architecture 'x64' was not found for Ubuntu 24.04.`

setup-python no encontró esa versión. El clásico: python-version: 3.10 sin comillas; YAML lo lee como el número 3.1. Escribe las versiones entre comillas: '3.10', '3.13'.

PRIMERO: `grep -n 'python-version' .github/workflows/*.yml`

→ Sección 7 · Tests en CI: Python, caché, matrix y checks obligatorios

---

`Resource not accessible by integration`

El GITHUB_TOKEN del job no tiene permiso para esa llamada a la API de GitHub (HTTP 403): comentar el PR, crear un release, escribir checks. En repos personales y organizaciones nuevas, por defecto solo puede leer contents y packages; en un PR desde un fork, siempre es de solo lectura.

PRIMERO:

```
grep -n -A4 'permissions' .github/workflows/*.yml
```

→ Sección 8 · Secretos, permisos y OIDC: entrar a AWS sin llaves

---

`Credentials could not be loaded, please check your action inputs: Could not load credentials from any providers`

configure-aws-credentials no consiguió ninguna credencial. Con role-to-assume y OIDC, casi siempre falta permissions: id-token: write en el job; unas líneas antes, el mismo paso pregunta: Did you mean to set the id-token permission?

PRIMERO: `grep -n 'id-token' .github/workflows/*.yml`

→ Sección 8 · Secretos, permisos y OIDC: entrar a AWS sin llaves

---

`Could not assume role with OIDC: Not authorized to perform sts:AssumeRoleWithWebIdentity`

AWS recibió el token de GitHub, pero la trust policy del rol (la regla que dice qué tokens pueden asumirlo) no lo acepta: el sub (quién pide) o el aud (para quién es) no coinciden. Típico: el job usa environment (el sub termina en :environment:produccion), corre en un PR (:pull_request) o el repo se creó, renombró o transfirió después del 15 de julio de 2026 y su sub trae IDs (repo:acme@123456/classifier@456789:…).

PRIMERO:

```
aws iam get-role --role-name gha-classifier-deploy --query Role.AssumeRolePolicyDocument
```

→ Sección 8 · Secretos, permisos y OIDC: entrar a AWS sin llaves

---

`no basic auth credentials`

docker push o pull contra ECR sin un login válido: no corriste el login, lo hiciste en otra región (el token es por región), pasaron las 12 h que dura o tu identidad no tiene permiso sobre ese repo.

PRIMERO:

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 123456789012.dkr.ecr.us-east-1.amazonaws.com
```

→ Sección 9 · Build y push de imágenes en CI

`tag invalid: The image tag '3f9c2ab' already exists in the 'classifier-gateway' repository and cannot be overwritten because the repository is immutable.`

El repo de ECR está configurado con tags inmutables (el default de ECR es mutable) y ese tag ya existe (en la API, ImageTagAlreadyExistsException). Es bueno: nadie pisa una versión. Pasa con :latest o al re-correr todo el workflow del mismo commit.

PRIMERO:

```
aws ecr describe-images --repository-name classifier-gateway --image-ids imageTag=3f9c2ab
```

→ Sección 9 · Build y push de imágenes en CI

`error: deployment "gateway" exceeded its progress deadline`

kubectl rollout status, el último paso del job deploy, falló: los pods nuevos no quedaron listos en progressDeadlineSeconds (600 s sin avance, por defecto). El job queda rojo, pero Kubernetes no vuelve atrás solo: con maxUnavailable: 0, los pods viejos siguen sirviendo. Si vence antes el --timeout del paso, el mensaje es otro, pero la causa es la misma.

PRIMERO:

```
kubectl -n classifier get pods -l
app.kubernetes.io/name=gateway
```

(cuáles no están Ready y por qué)

→ Sección 10 · CD: GitOps o kubectl, aprobación, concurrency y rollback con revert

`Canceling since a higher priority waiting request for 'deploy-gateway-produccion' exists`

concurrency canceló un run que esperaba: en ese grupo solo espera uno, y llegó uno más nuevo. No es un error del código. Si cada deploy tiene que correr, usa queue: max; nunca cancel-in-progress: true en un deploy.

PRIMERO: `gh run list --workflow gateway.yml --limit 5` (cuál quedó y cuál se canceló)

→ Sección 10 · CD: GitOps o kubectl, aprobación, concurrency y rollback con revert

`GitHub Actions is not permitted to create or approve pull requests.`

El workflow quiso abrir un PR (el del manifest) con GITHUB_TOKEN y el repo no lo permite: Allow GitHub Actions to create and approve pull requests está apagado, que es el default en repos personales y en organizaciones nuevas. Con un token de GitHub App no hace falta esa opción.

PRIMERO:

`gh api repos/OWNER/REPO/actions/permissions/workflow` (mira can_approve_pull_request_reviews)

→ Sección 10 · CD: GitOps o kubectl, aprobación, concurrency y rollback con revert

`Waiting for a runner to pick up this job...`

Ningún runner libre tiene todas las labels que pide runs-on: el self-hosted está apagado u offline, las labels no coinciden, o llegaste al límite de jobs en paralelo. Con self-hosted, a las 24 h en cola, el job falla.

PRIMERO:

```
gh api repos/OWNER/REPO/actions/runners --jq '.runners[]
| .name + " " + .status'
```

→ Sección 11 · Runners: GitHub-hosted o self-hosted (la T4 del C01), labels, timeouts y costos

`The job running on runner GitHub Actions 7 has exceeded the maximum execution time of 360 minutes.`

El job llegó a su timeout-minutes (360 si no pusiste nada, que también es el máximo en GitHub-hosted) y GitHub lo canceló. Algo se colgó (un test esperando la red, un prompt interactivo) o el job es demasiado grande para ese runner.

PRIMERO: `gh run view RUN_ID --log | tail -n 50` (qué hacía al final)

→ Sección 11 · Runners: GitHub-hosted o self-hosted (la T4 del C01), labels, timeouts y costos

`Error: Process completed with exit code 1.`

Un comando del paso terminó con un código distinto de 0 y el paso falló. No es la causa: la causa está en las líneas de arriba (en pytest, el resumen FAILED; exit 2 = error al importar los tests; 5 = no encontró tests; 127 = comando no encontrado; 137 = lo mató la memoria).

PRIMERO: `gh run view RUN_ID --log-failed`

→ Sección 12 · Diagnóstico: leer el log, re-run con debug, reproducir en local, tipos de falla

EJEMPLO REAL · CÓMO LEER UN PASO DE OIDC QUE FALLA EN EL JOB DE DEPLOY

TEXTO Forma del log de GitHub Actions con configure-aws-credentials v6.3.0 (los mensajes salen de su código fuente; el de STS, de su README). Nombres del C01; salida abreviada.

```
Run aws-actions/configure-aws-credentials@e1253824e5c10ff9df46874f81ed3ec929e19cfd
  with:
    role-to-assume: arn:aws:iam::123456789012:role/gha-classifier-deploy
    aws-region: us-east-1
    …
Assuming role with OIDC
…
Error: Could not assume role with OIDC: Not authorized to perform sts:AssumeRoleWithWebIdentity
```

- Tres voces: `Run …` y `with:` los imprime el runner; `Assuming role with OIDC`, la action; lo que sigue a `Could not assume role with OIDC:` lo dijo AWS STS.
- `Not authorized`, y no `Credentials could not be loaded`: el token OIDC sí existe (el `id-token: write` está bien) y AWS lo rechazó. Casi siempre, la trust policy del rol (la regla que dice qué tokens pueden asumirlo) no acepta ese `sub` (quién pide el token) o ese `aud` (para quién es).
- El job tiene `environment: produccion`, así que su `sub` es `repo:acme/classifier:environment:produccion`, no `…:ref:refs/heads/main`. Con un environment, el `sub` por defecto no trae la rama.
- En un repo creado después del 15 de julio de 2026 (o renombrado o transferido desde entonces, o que eligió el formato nuevo), el `sub` trae además los IDs: `repo:acme@123456/classifier@456789:…`. Una trust policy copiada de un tutorial viejo falla igual.
- Primero, mira la condición: `aws iam get-role --role-name gha-classifier-deploy --query Role.AssumeRolePolicyDocument`. El arreglo va en AWS (la trust policy), no en el workflow.

CÓMO RECORDARLA

quién habla dice dónde mirar. `fatal`, `error` y `hint`: git en tu máquina. `remote:` y `[remote rejected]`: GitHub. `Error:` en el log: el runner o una action. `Could not assume role` o `An error occurred (…)`: AWS. `Waiting` o `Canceling`: GitHub esperando algo, no un bug.

TRAMPA

leer solo la última línea. `Process completed with exit code 1` y `error: failed to push some refs` son consecuencias: la causa está arriba (el FAILED de pytest, el hint de git, las líneas `remote:`). Y los textos cambian con la versión: git reescribe sus hints y las actions sus mensajes entre releases; busca la parte central, no la línea entera.

En el curso: M10 · CI/CD para modelos: la pirámide de tests · M5 · Registries y tags inmutables · M15 · Secretos y credenciales · C01 · Modos de falla y degradación

---

Comandos, campos y límites revisados en octubre de 2026 contra la documentación oficial: git-scm.com (Git 2.56), docs.github.com (GitHub Actions, protección de ramas y OIDC), los README de las actions que se citan y la documentación de AWS (IAM, STS y ECR). Los laboratorios de git se probaron de verdad en una carpeta de prueba con git 2.50.1; los de GitHub Actions se verificaron contra la documentación. Si tu terminal dice otra cosa, créele a tu terminal. Los ejemplos salen de `reference/` del curso; los workflows son ilustrativos: el C01 no trae ninguno.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error