

Ingeniería de Software

Despliegue y operación

Contenedores · Apagarse bien · Feature flags · SLOs · Incidentes y postmortems

Gerson Garrido

Semana 13

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: desplegar

1. Entrega y despliegue continuos
2. Imágenes y capas
3. El estado y los volúmenes
4. Apagarse bien
5. Estrategias y feature flags

Sesión B: operar

6. Telemetría y salud
7. SLI, SLO, SLA y presupuesto
8. Incidentes y postmortems
9. DORA

Lecturas

SWEG, caps. 24 *Continuous Delivery* y 25 *Compute as a Service*, hasta *Choosing a Compute Service*.

SRE, caps. 4 *Service Level Objectives* y 15 *Postmortem Culture*.

Complementarias: *La aplicación de doce factores* (III, IX, X); Hodgson, *Feature Toggles*; Docker, *Get started* y la referencia del Dockerfile; SRE, cap. 6; *SRE Workbook*, *Implementing SLOs*; DORA, *software delivery performance metrics*; *What is OpenTelemetry?*

Producción cambia las preguntas

- ¿Qué versión está corriendo?
- ¿Dónde vive la base de datos cuando se reemplaza el proceso?
- ¿Qué le pasa a la reserva que llegó justo al apagar?
- ¿Cómo nos enteramos de una caída antes de que llame la administradora?
- ¿Cuánto puede fallar sin que sea un problema?
- Cuando algo sale mal, ¿cómo aprendemos sin buscar a quién culpar?

> **Hoy se mide casi todo sobre Kallpa**
>
> Un `docker stop` de 10 s o de 0.1 s por una línea del `Dockerfile`. Un servidor que ignora una señal solo en un contenedor. Un contenedor `healthy` sin las reservas.

SECCIÓN 01

Entrega y despliegue continuos

01



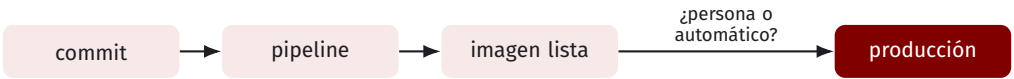
Dos cosas con las mismas siglas

Entrega continua ***(continuous delivery)***

Cada cambio que pasa el pipeline queda listo para desplegarse. Una persona aprieta el botón.

Despliegue continuo ***(continuous deployment)***

Además, cada cambio que pasa el pipeline se despliega solo.



La diferencia no es técnica

Es de confianza: lo que la persona revisaba lo tiene que revisar otra cosa.

Más rápido es más seguro

El círculo vicioso

1. El release cuesta y da miedo.
2. Se espera para probarlo un poco más.
3. Entra “una funcionalidad más”.
4. El release crece. Cuesta más.

La salida: *faster is safer*. Lotes pequeños: fáciles de entender y de revertir.

Dos casos de Google

YouTube: 50 horas de pruebas manuales por release.

Search: de una vez por semana, con suerte, a cada dos días, tras años de automatización.

Relatos de una empresa, contados por sus autores: no es un experimento.

SWEG, cap. 24

Tres cosas que no se mezclan

Código
igual en todos los entornos
en la imagen

Configuración
cambia entre entornos
en el entorno (factor III)

Estado
las reservas: no se reemplaza
en un volumen

La prueba del factor III
¿Podrían publicar el código hoy sin filtrar ninguna credencial?

SECCIÓN 02

Imágenes y capas

02

Contenedor e imagen

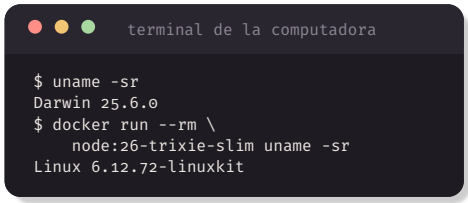
Contenedor

Un proceso con su propio sistema de archivos, red y lista de procesos, que comparte el kernel. No es una máquina virtual.

Imagen

El sistema de archivos inicial, en capas de solo lectura, más metadatos. De una imagen salen muchos contenedores.

¿Y en mi Mac? Docker Desktop corre una máquina virtual Linux:



Mascotas y ganado (SWEG cap. 25): cualquier instancia se puede borrar en cualquier momento.

Un primer `Dockerfile`: funciona

Dockerfile.ingenuo

```
1  FROM node:26-trixie-slim
2  COPY . /app
3  WORKDIR /app
4  CMD node src/main.ts
```

Con un `.env` y una base de datos en la carpeta, como la de quien estuvo desarrollando. Miremos adentro.

docker run –rm –entrypoint sh kallpa:ingenua -c '...'

```
$ id -u; ls -a /app; cat .env; ls /app/data
0
.
..
.env
.git
.gitignore
Dockerfile
Makefile
README.md
data
eslint.config.js
package.json
public
src
test
tsconfig.json
DB_PASSWORD=contrasena-de-ejemplo
kallpa.db
```

Cinco problemas que ninguna prueba ve

1. **Copia todo:** `.env`, `.git`, pruebas, base de datos de desarrollo.
   → `COPY` de lo justo y un `.dockerignore` como segunda barrera.
2. **Corre como root** (el `0` de `id -u`).
   → `USER node`: las imágenes de Node ya traen ese usuario.
3. **`CMD` en forma shell.**
   → `CMD ["node", "src/main.ts"]`. Ya vamos a ver por qué.
4. **No dice cómo saber si está sano.**
   → `HEALTHCHECK`. La imagen `slim` no trae `curl`.
5. **No prepara la carpeta de datos.**
   → `RUN mkdir data && chown node:node data`, antes de `USER node`.

Capas y caché: lo que cambia seguido va al final

Cambiamos una línea de `src/domain/court.ts`:

docker build (filtrado)

```
#6 [2/6] WORKDIR /app
#6 CACHED
#7 [4/6] COPY package.json ./
#7 CACHED
#8 [3/6] RUN mkdir data && chown node:node data
#8 CACHED
#9 [5/6] COPY public ./public
#9 CACHED
#10 [6/6] COPY src ./src
#10 DONE 0.0s
```

Regla

La primera capa que cambia invalida las de abajo.

Con dependencias se nota más:

1. `COPY package.json package-lock.json`
2. `RUN npm ci`
3. el código

Al revés, una línea reinstala todo.

Borrar no borra

Un `COPY` y luego un `RUN rm`: la capa no cambia, otra capa la esconde.

ejemplos/secreto

```
$ docker run --rm secreto ls -a /app
.
..
$ docker history secreto --format "{{.CreatedBy}} | {{.Size}}" | head -3
RUN /bin/sh -c rm credenciales.txt # buildkit | 8.19kB
COPY credenciales.txt . # buildkit | 12.3kB
WORKDIR /app | 8.19kB
$ docker save secreto | tar -x -C build/secreto
$ cd build/secreto/blobs/sha256 && for capa in *; do tar -xOf "$capa" app/credenciales.txt 2>/dev/null; done
DB_PASSWORD=contrasena-de-ejemplo
```

Error común

Credenciales en la imagen: entran al correr, por el entorno. `ENV` y `ARG` también quedan a la vista.

Imágenes inmutables: la misma pasa por todo

etiquetar con el SHA

```
$ git log --oneline | head -3
e979270 Cambia una linea
423e2b2 Ordena las capas de la imagen
b6585a3 Agrega health, logs JSON, apagado ordenado y flag ...
$ docker images kallpa --format "{{.Tag}} {{.ID}}"
e979270 c8b329c848c4
latest c8b329c848c4
423e2b2 f9148b91579f
...
```

rollback: correr la anterior

```
$ docker run -d --name kallpa -p 8080:3000 \
    -v kallpa-data:/app/data kallpa:423e2b2
...
$ docker ps --filter name=kallpa --format "{{.Names}} {{.Image}}"
kallpa kallpa:423e2b2
```

- `latest` no quiere decir “la última”: es la etiqueta por omisión, y se mueve.
- Con el SHA se sabe qué corre, y `git show` dice qué código es.
- Entre entornos solo cambia la configuración.

Ojo con las migraciones

El rollback es fácil si la base de datos sigue siendo compatible (semana 16).

SECCIÓN 03

El estado

03

La base de datos está en la capa escribible

sin volumen

```
$ docker run -d --name kallpa -p 8080:3000 kallpa
...
$ docker diff kallpa
C /app
C /app/data
A /app/data/kallpa.db
```

Una `A` es *agregado*: el archivo vive en este contenedor.

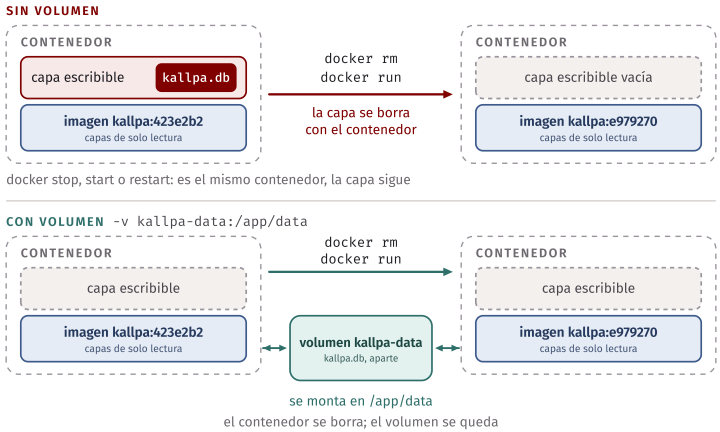
Comando	La capa
`stop` + `start`	sigue
`restart`, `--restart`	sigue
`rm`	se borra

Desplegar es `rm` más `run`: otra imagen, otra capa vacía. Y nada falla.

Volumen

Almacenamiento de Docker, aparte de cualquier contenedor: `-v kallpa-data:/app/data`. Sobrevive a un despliegue, no a perder el disco.

Capa escribible y volumen



Error común: la carpeta con otro owner

Con `USER node`, sin preparar `/app/data` en la imagen:

dos errores distintos

```
$ docker run --rm -v w13v:/app/data kallpa:sin-chown 2>&1 | grep -E "Error|errcode|code:|at " | head -6
Error: unable to open database file
    at openDatabase (file:///app/src/store/database.ts:41:14)
...
$ docker run --rm --entrypoint ls -v w13v:/app/data kallpa:sin-chown -ld /app/data
drwxr-xr-x 2 root root 4096 Nov  2 06:13 /app/data
$ docker run --rm kallpa:sin-chown 2>&1 | grep -E "Error|code:" | head -4
Error: EACCES: permission denied, mkdir 'data'
  code: 'EACCES',
```

Por qué funciona `chown` en la imagen

Un volumen con nombre, vacío, copia lo que la imagen tiene en esa ruta, con owner y permisos. Un bind mount no.

SECCIÓN 04

Apagarse bien

04

Señales

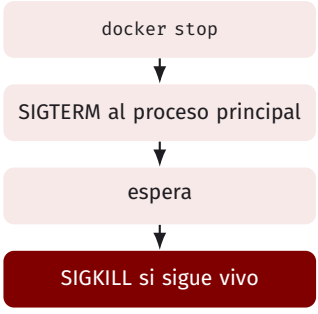
SIGTERM (15)

"Termina, por favor." Se puede atrapar.

SIGKILL (9)

Mata sin avisar. No se puede atrapar.

Código de salida: 128 + *n*. **143** por SIGTERM, **137** por SIGKILL.



`docker rm -f` va directo a SIGKILL. Factor IX: desechable, arranca rápido y termina bien.

El resultado que sorprende

Kallpa con manejador, en Docker

```
$ time docker stop kallpa
kallpa
real 0.113s
$ docker inspect -f '{{.State.ExitCode}}' kallpa
0
```

sin manejador, en Docker

```
$ time docker stop kallpa
kallpa
real 10.141s
$ docker inspect -f '{{.State.ExitCode}}' kallpa
137
```

Y la Kallpa publicada, sin manejador, en el contenedor del curso:

make start + pkill

```
$ pkill -TERM -f 'node src/main.ts'
...
make: *** [Makefile:22: start] Terminated
```

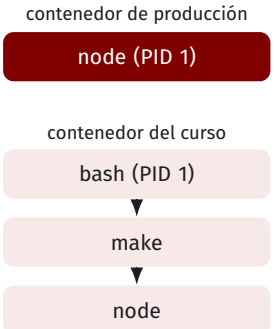
El mismo programa, sin manejador, muere de inmediato en un caso e ignora la señal en el otro. ¿Por qué?

El proceso 1

La regla del kernel

Al proceso con PID 1 de un espacio de procesos no se le aplica la acción por omisión de una señal. Si no tiene manejador, la señal se ignora.

- `CMD ["node", ...]`: Node es el proceso 1.
- `make start`: Node es hijo de `make`. Muere.



Por eso el laboratorio se comporta distinto en sus pasos 3 y 5.

Cómo se arranca decide si corre el manejador

Cómo se arranca	Proceso 1	¿Manejador?	Tiempo	Salida
`CMD ["node", ...]`, con manejador	`node`	corre	0.113 s	0
`CMD ["node", ...]`, sin manejador	`node`	no hay	10.141 s	137
`CMD node ...`, con manejador	`/bin/sh -c`	no corre	10.141 s	137
`CMD ["npm", "start"]`, con manejador	`npm start`	no corre	0.669 s	1
`--init`, sin manejador	el `init` de Docker	no hay	0.106 s	143
`--init`, con manejador	el `init` de Docker	corre	0.229 s	0

El tiempo engaña en los dos sentidos

Con `npm start` tarda poco, pero el log no tiene ni una línea `shutting down`. Lo que hay que mirar es el log.

Medido con `docker stop` y 10 s de espera

Cuánto espera `docker stop`: compruébenlo

La documentación dice 10 s en Linux. En el Docker Desktop 29.2.1 con el que se preparó esta clase:

terminal de la computadora

```
$ docker run -d --name kallpa kallpa:sin-sigterm
b7d3ad4d8f15f8b45d7db1e9498015b17b3141e293c216bb295cd94daca01a0a
$ docker inspect -f '{{.Config.StopTimeout}}' kallpa
1
$ time docker stop kallpa
kallpa
real 1.148s
$ docker inspect -f '{{.State.ExitCode}}' kallpa
137
```

- Para fijarlo: `docker run --stop-timeout 10` o `docker stop -t 10`.
- En su hosting será otro número: búsquenlo. Es lo que tiene su aplicación para apagarse.

Un apagado ordenado en Node

ejemplos/apagado.ts

```ts
export function shutdown(deadlineMs: number): void {
  console.log('apagando');
  // Si algo no termina a tiempo, se corta a propósito antes de que llegue SIGKILL.
  const deadline = setTimeout(() => server.closeAllConnections(), deadlineMs);
  deadline.unref();
  // close deja de aceptar conexiones y avisa cuando ya no queda ninguna abierta.
  server.close(() => {
    console.log('apagado');
    process.exitCode = 0;
  });
}
process.once('SIGTERM', () => shutdown(8000));
```

- Deja de escuchar, deja terminar lo que está en curso y sale (factor IX). En Kallpa, además, `db.close()`.
- El límite va por debajo de lo que espera quien lo apaga.

Con una solicitud en curso

`curl` manda un `POST` y tarda 3 s en mandar el cuerpo. Al segundo, apagamos:

con manejador

```
$ time docker stop -t 10 kallpa
cliente: http=200 exit=0 t=3.005829
kallpa
real 2.203s
$ docker logs kallpa | tail -4
{"time":"2026-11-02T06:14:00.713Z","level":"info","msg":"shutting down","signal":"SIGTERM"}
[whatsapp] para 912345678: Hola Luis Huamán Torres, tu reserva de Vóley el 2026-11-05 de 10:00 ...
{"time":"2026-11-02T06:14:02.698Z","level":"info","msg":"request","method":"POST",...,"status":200,...}
{"time":"2026-11-02T06:14:02.699Z","level":"info","msg":"stopped"}
```

sin manejador, con menos paciencia

```
$ time docker stop -t 1 kallpa
kallpa
real 1.226s
cliente: http=100 exit=52 Empty reply from server
```

¿Qué le pasa a la reserva si llega SIGKILL?

La base de datos queda bien

Una transacción se confirma entera o no se confirma. Lo prueba `ejemplos/transaccion-cortada.ts`: se mata con SIGKILL en medio de un `INSERT`, y la fila no está.

Quien reservó no sabe qué pasó

La conexión se corta. Puede haber quedado confirmada, y el reintento da `SLOT_TAKEN`. O confirmada sin WhatsApp.

Detalle. `server.close` termina cuando se cierran las *conexiones*. Una conexión keep-alive que estaba ocupada queda abierta hasta su `keepAliveTimeout`, de 5 s: en la prueba, la respuesta llega en menos de 1 s y el proceso sale a los 6.3 s.

SWEG cap. 25: en Google, Borg avisa antes de mover un contenedor, y el load balancer manda el tráfico a otra réplica.

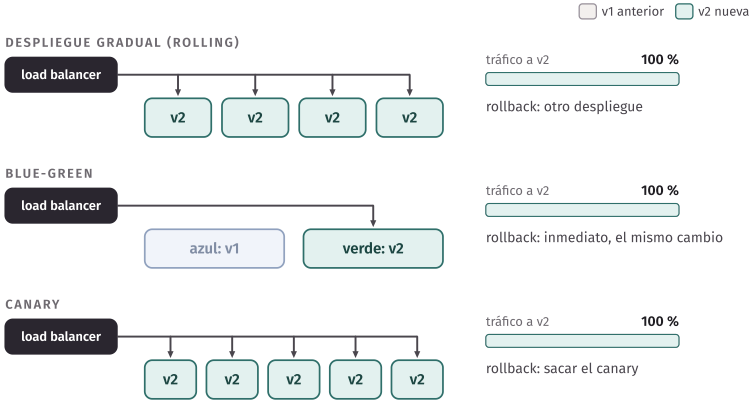
SECCIÓN 05

Estrategias y flags

05



Tres estrategias de despliegue



Rolling termina, blue-green cambia todo en un solo paso, y el canary sube porque le fue bien

Las estrategias, en Kallpa

- Todas piden **varias instancias** y que dos versiones entiendan la misma base de datos.
- Kallpa: una instancia y un SQLite. Dos procesos en el mismo archivo dan `database is locked` (semana 7), y un canary con pocas solicitudes por hora no mide nada.
- Lo honesto: cortes de pocos segundos en horas sin reservas, y un rollback ensayado.

Feature flags: desplegar no es lanzar

Tipo	Para qué	Duración	En Kallpa
Release	Esconder algo a medio terminar	Semanas	Turnos fijos
Experimento	Comparar variantes (A/B)	De horas a semanas	Texto del botón
Operación	Apagar algo caro o roto sin desplegar (*kill switch*)	Corta; algunos, muy larga	`KALLPA_BOOKINGS_OPEN`
Permisos	Dar acceso a algunos	Larga	Reserva anticipada a socios

Error común: flags para siempre

Cada flag es una rama más: diez son más de mil combinaciones. Tienen owner y fecha para retirarse.

Hodgson, *Feature Toggles*

Un flag de operación, sin construir otra imagen

terminal de la computadora

```
$ docker run -d --name kallpa -p 8080:3000 -e KALLPA_BOOKINGS_OPEN=false kallpa
e934624cb4170adeb057e4ec1b1eb17f3c615d567a28c0c760664123bc573102
$ curl -s -o /dev/stdout -w '\n%{http_code}\n' -X POST localhost:8080/api/bookings -d ...
{"error":{"code":"BOOKINGS_CLOSED","message":"Las reservas están suspendidas por el momento."}}
503
$ curl -s -w '\n%{http_code}\n' 'localhost:8080/api/courts/3/availability?date=2026-11-05' | cut -c1-60
{"courtId":3,"date":"2026-11-05","slots":[{"hour":7,"availab
200
```

`docker restart` conserva el entorno: cambiar el flag es `rm` más `run`, con la misma imagen y el mismo `-v`.

Con IA: un agente que despliega tiene acceso a producción

Pipeline, permisos mínimos, revisión y rollback probado. Nunca credenciales de producción en su contexto. Un `CMD npm start` como root funciona y pasa las pruebas: lo delata medir.

SECCIÓN 06

Telemetría y salud

06

Tres tipos de telemetría

Logs
eventos individuales
“a las 19:04 falló este POST”

Métricas
números agregados
“142 solicitudes por minuto, 2 % de error”

Traces
el recorrido por varios servicios
Kallpa: un solo proceso

OpenTelemetry

Un estándar abierto y herramientas para generar, recolectar y exportar los tres.
No es un backend: no guarda ni muestra los datos.

Logs estructurados

de una línea que se lee a una que se consulta

```
GET /api/courts 200
{"time":"2026-11-02T06:10:47.116Z","level":"info","msg":"request","method":"POST","path":"/api/bookings",...
```

ejemplos/kallpa-muestra.txt

```
$ grep '^{' kallpa-muestra.txt | jq -r 'select(.msg=="request") | .status' | sort | uniq -c
     57 200
      1 400
      1 404
      2 409
      1 422
      3 500
$ grep '^{' kallpa-muestra.txt | jq -r 'select(.status>=500) | .path' | sort | uniq -c
      2 /api/courts/%ZZ/availability
      1 /api/members/%ZZ/bookings
$ grep '^{' kallpa-muestra.txt | jq -c 'select(.level=="error") | {time,path,error}' | head -1
{"time":"2026-11-02T06:15:17.512Z","path":"/api/courts/%ZZ/availability","error":"URI malformed"}
```

En tres líneas: cuántos errores, en qué rutas, desde qué hora y por qué.

Reglas de un log

- A **stdout**: la aplicación no decide adónde van (factor XI).
- Campos fijos, un nivel y `time` en UTC. Un `requestId` si una solicitud deja varias líneas.
- **Nada personal ni secreto.** Los logs se copian y se guardan más que los datos.

el mismo log de Kallpa

```
[whatsapp] para 912345678: Hola Luis Huamán Torres, tu reserva de Vóley el 2026-11-05 de 10:00 a 11:00
...
```

Error común

En Kallpa es un simulador del canal, con datos de ejemplo. En producción, esa línea es una fuga de datos personales. En su proyecto no debe existir.

Un endpoint de salud

Dockerfile

```
1  HEALTHCHECK --interval=5s --timeout=2s --start-
       period=5s --retries=3 \
2    CMD ["node", "-e", "fetch('http://localhost
       :3000/health').then(r => process.exit(r.ok ?
        0 : 1), () => process.exit(1))"]
```

cada segundo

```
t=0x1s starting
...
t=4x1s starting
t=5x1s healthy
```

- Rápido y sin autenticación.
- ¿Vivo, o puede atender? P4 pide lo segundo: `SELECT 1`, y `503` si no.
- No revisa a terceros: reiniciar no los arregla.
- Por omisión revisa cada 30 s.

`unhealthy` no reinicia. Con `--restart=always` y un chequeo que siempre falla: `status=running health=unhealthy restarts=0`. Reiniciar por salud lo hace otra cosa: Swarm, las probes de Kubernetes, su hosting.

Un pedido apaga el servidor, y nada se entera

Kallpa publicada

```
$ curl -s --path-as-is localhost:3000//; echo "curl salio con $?"
curl salio con 52
$ curl -s localhost:3000/api/courts; echo "curl salio con $?"
curl salio con 7
```

el log

```
TypeError: Invalid URL
    at new URL (node:internal/url:911:25)
    at Server.<anonymous> (file:///tmp/pub/src/http/server.ts:49:17)
...
  input: '//',
```

`new URL` está fuera del `try`, en una función `async`: promesa rechazada sin manejar, y Node termina.

Sin política de reinicio:
`status=exited exit=1 restarts=0`

Con `--restart=on-failure`:
`status=running exit=0 restarts=1`

Mitigar no es arreglar

Sin vigilar el contador de reinicios, el error se vuelve invisible.

Las cuatro señales doradas

- **Latencia**: separando la de las buenas de la de los errores.
- **Tráfico**: solicitudes por segundo, o lo que tenga sentido.
- **Errores**: explícitos (`500`), implícitos (`200` con contenido equivocado) o por política.
- **Saturación**: el recurso más escaso. En Kallpa, el event loop y el disco del volumen.

Se alerta por síntomas

Síntoma: *qué* se rompió. Causa: *por qué*. “CPU al 80 %” despierta a alguien sin que nadie note nada.

Error común: medir lo que es fácil

`/health` con `200` y todas las reservas fallando: “arriba” para el monitoreo, caído para los vecinos.

SRE, cap. 6

SECCIÓN 07

SLI, SLO, SLA y presupuesto

07

Tres siglas, tres preguntas

SLI: ¿qué mido?

Eventos buenos sobre válidos. Kallpa: solicitudes a `/api/*` sin 5xx, sobre todas las de `/api/*`.

SLO: ¿cuánto quiero?

Un objetivo sobre un SLI en una ventana: 99.5 % de esas solicitudes en 30 días. Es interno.

SLA: ¿qué pago si no llego?

Un contrato con consecuencias. Si no hay ninguna explícita, es un SLO. Un SLA razonable promete menos que el SLO, para tener margen.

Un `409` o `422` son **buenos**: el error es del pedido. `/health` no es válido: ningún vecino lo pide.

SRE, cap. 4; *Workbook*, *Implementing SLOs*

Calcular un SLI: mirar qué se cuenta

el conteo rápido del laboratorio

```
$ grep '^{' kallpa-muestra.txt | jq -s '{total: length, errores: map(select(.status >= 500)) | length}' -c
{"total":72,"errores":3}
$ grep '^{' kallpa-muestra.txt | jq -r .msg | sort | uniq -c
      1 listening
      1 new database, sample data loaded
     65 request
      1 shutting down
      1 stopped
      3 unhandled error
```

ejemplos/sli.jq: solo /api/*

```
$ grep '^{' kallpa-muestra.txt | jq -s -c -f sli.jq
{"validas":43,"buenas":40,"sli":0.9302325581395349}
```

Un log solo ve lo que le llega. Con el proceso caído no hay línea, y el SLI sale perfecto. Por eso, también, un monitor externo.

Presupuesto de errores

Presupuesto de errores

Lo que el SLO permite fallar: $1 - \text{SLO}$.

$0.005 \times 720\ \text{h} = 3.6\ \text{h}$ de caída total, o
$0.005 \times 20\,000 = 100$ solicitudes al mes.

Vuelve la discusión una cuenta

Mientras queda, se despliega. Si se acaba, se prioriza la confiabilidad. 150 errores gastan 150 %.

SLO	En 30 días
99 %	7.2 h
99.5 %	3.6 h
99.9 %	43.2 min
99.99 %	4.32 min

100 % es un error

Inalcanzable, congela todo cambio, y el vecino no lo nota: su WiFi falla más.

Elegir el objetivo

- No por el desempeño actual.
- Simple: lo complicado esconde cambios.
- Sin absolutos: “siempre disponible”.
- Pocos: si no decide una discusión, sobra.
- La perfección puede esperar: mejor apretar que aflojar.

Con la administración: ¿es igual a las 3 de la mañana que un sábado a las 6 de la tarde?

Chubby

Tan confiable que otros dependían de que nunca cayera. Si en un trimestre no bajó al objetivo, lo apagan a propósito. El SLO es un objetivo, no un piso.

SRE, cap. 4

Alertar por lo que se quema

Velocidad de quemado ***(burn rate)***

Tasa de error observada entre la permitida. Con 1, el presupuesto se acaba al final de la ventana.

Error	Velocidad	Dura
0.5 %	1	30 días
5 %	10	3 días
100 %	200	3.6 h

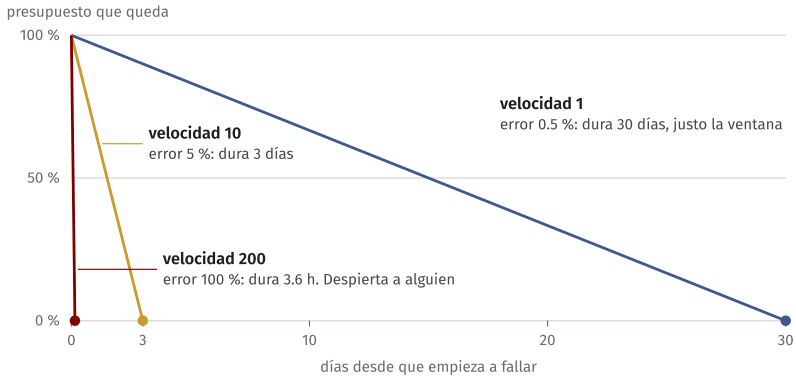
Workbook, Alerting on SLOs

Para avisar a una persona, como punto de partida: 2 % del presupuesto en una hora (velocidad 14.4).

Kallpa tiene poco tráfico

Unas 28 solicitudes por hora: es 3.6 % de error y 1 % del presupuesto. Un monitor externo da más señal que una fórmula. Una alerta, como una prueba, suena solo si se viola algo que importa; la recibe quien está *on-call*.

Cuánto dura el presupuesto



SECCIÓN 08

Incidentes y postmortems

08

Un incidente

Incidente

Algo que afecta a los usuarios y requiere una respuesta coordinada.

Roles, aunque sean tres

- **Quien coordina**: decide; no toca nada.
- **Quien investiga**: mira logs, ejecuta.
- **Quien comunica**: le cuenta a la administración.

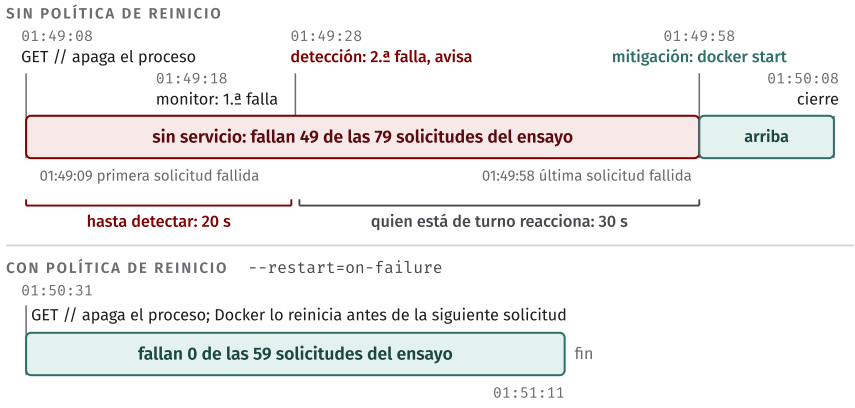
Primero mitigar, después entender

Rollback, apagar un flag, `docker start`. La causa se busca con el servicio arriba.

Error común: mitigar y borrar la evidencia

`docker rm -f` borra la capa y los logs. Antes: `docker logs` y `docker cp`.

Un game day: se provoca a propósito



reinicios=1, pero el monitor nunca avisó: si nadie mira ese número, el error queda escondido

Lo que deja ver el ensayo

- `ejemplos/game-day.sh kallpa`, sin y con `--restart=on-failure`. Al final, `estado=running` en los dos: `reinicios=0` y `reinicios=1`.
- **Con cuidado:** el reinicio tardó menos que el segundo entre dos solicitudes. La que tumbó el proceso sí falló. Y el monitor nunca avisó.
- Una solicitud por segundo, un monitor cada 10 s que avisa a la segunda falla, y una persona que tarda 30 s, simulada.

Postmortem sin culpables

Postmortem

Un registro escrito de un incidente: impacto, mitigación, causas y acciones para que no se repita.

Se escribe si hubo

caída visible, pérdida de datos de cualquier tipo, un rollback, una resolución larga o una falla de monitoreo. Se decide *antes*.

Sin culpables

Busca las causas sin acusar a nadie, y supone buenas intenciones. Con castigo, la gente esconde los problemas.

No se puede “arreglar” a las personas, pero sí los sistemas y procesos que las ayudan a decidir bien.

Sin culpables no es sin responsables: cada acción tiene owner. Y un postmortem sin revisar es como si no existiera.

SRE, cap. 15

Escribir sin culpables

En vez de	Escriban
“Pedro olvidó el `-v`.”	El comando de despliegue se escribe a mano, y nada comprueba que incluya el volumen.
“Fue un error humano.”	Un solo comando puede reemplazar producción sin revisión, y el monitor no lo detecta.
“No revisaron bien.”	La revisión no incluye el script de despliegue, porque no está en el repositorio.

“Tener más cuidado” no es una acción

No se comprueba, no tiene fecha y se olvida. Una acción es algo que después se puede señalar: un script, una prueba, una alerta.

Un postmortem: el proceso caído

Tipo: game day

Impacto: 50 s de caída total; 49 de 79 solicitudes fallidas; 49 % del presupuesto, en una cuenta gruesa. Ningún dato perdido.

Causa inmediata: `new URL` fuera del `try`.

Del proceso: las pruebas no mandan rutas mal formadas; no hay política de reinicio; el aviso depende de una persona.

01:49:08	Empieza: llega `GET //`
01:49:18	El monitor ve la primera falla
01:49:28	Detección: segunda falla, 20 s después
01:49:58	Última solicitud fallida
01:49:58	Mitigación: `docker start`

Acción	Tipo	Owner	Fecha
Mover `new URL` al `try`, con una prueba que mande `GET //`	evita	backend	11-06
`--restart=on-failure` en el script; repetir el game day	mitiga	despliegue	11-06
Avisar cuando suba `RestartCount`	detecta	operación	11-13

`ejemplos/2026-11-02-proceso-caido.md`

Con IA: la línea de tiempo se comprueba

Un resumen como los que suele proponer una herramienta: *“Los errores empezaron a las 06:10, afectaron a todas las rutas de la API y se resolvieron solos.”*

contra el mismo log

```
$ grep '^{' kallpa-muestra.txt | jq -c 'select(.level=="error") | {time,path,error}' | head -1
{"time":"2026-11-02T06:15:17.512Z","path":"/api/courts/%ZZ/availability","error":"URI malformed"}
$ grep '^{' kallpa-muestra.txt | jq -r 'select(.status>=500) | .path' | sort | uniq -c
      2 /api/courts/%ZZ/availability
      1 /api/members/%ZZ/bookings
```

- Empezaron a las 06:15, en dos rutas, y fueron tres errores.
- La causa la confirman ustedes, reproduciéndola.
- Antes de pegar logs en una herramienta externa, miren qué llevan adentro.

SECCIÓN 09

DORA

09

Las cinco métricas de DORA

Rendimiento ***(throughput)***

- Tiempo de entrega de un cambio: commit a producción.
- Frecuencia de despliegue.
- Tiempo de recuperación de un despliegue fallido (reemplazó al MTTR).

Inestabilidad ***(instability)***

- Tasa de fallo de cambios.
- Tasa de retrabajo: despliegues no planeados, por un incidente.

Velocidad y estabilidad no se contraponen

Están correlacionadas para la mayoría de los equipos. Es correlación, no causa.

No son metas

Goodhart: la meta invita a la trampa. Un servicio a la vez, para ver si *ustedes* mejoran.

DORA, *software delivery performance metrics*

SECCIÓN 10

Laboratorio

10

Laboratorio: observar

1. `GET /health`, con su prueba.
2. Logs JSON: `time`, `level`, `msg`, `method`, `path`, `status`, `durationMs`, sin datos personales. Un `500` también. Calcular un SLI con `jq`.
3. Apagado ordenado: `SIGTERM`, `server.close`, `db.close`, salir con 0. Probar con `pkill`.

> **Dónde se corre**
>
> Los pasos 1 a 3 van en el contenedor del curso. **Docker no está dentro**: los pasos 4 a 7 van en la terminal de su computadora, con el puerto 8080.

En su repositorio de Kallpa: una rama nueva desde `main`, un pull request por paso.

Laboratorio: desplegar

4. `Dockerfile` y `.dockerignore`: `node:26-trixie-slim`, `USER node`, `HEALTHCHECK`, `/app/data` del usuario `node`.

su computadora

```
$ docker build -t kallpa:$(git rev-parse --short HEAD) -t kallpa .
$ docker run -d --name kallpa -p 8080:3000 -v kallpa-data:/app/data kallpa
$ docker inspect --format '{{.State.Health.Status}}' kallpa
```

5. Comparar `docker stop` con y sin manejador. Antes, miren `.Config.StopTimeout`.
6. Flag de operación `KALLPA_BOOKINGS_OPEN`: `503` con un código estable, y la disponibilidad se sigue viendo. Sin construir otra imagen.

Laboratorio: el incidente

7. `code/week-13/incidente.md`: lo que vio la administración un viernes. Reprodúzcanlo, encuentren la causa, mitiguen, y comprueben que una reserva sobrevive a un despliegue.
8. El postmortem en `docs/postmortems/AAAA-MM-DD-tema.md`, con `postmortem-template.md`: `Tipo: game day`, Resumen, Impacto, Línea de tiempo, Causa raíz y Acciones. Al menos tres acciones con owner y fecha.

Tres preguntas que el relato deja abiertas

¿Por qué aparece un equipo que nadie conoce? ¿Por qué `/health` dice que todo está bien? ¿A qué vecinos hay que llamar: a quienes reservaron antes del despliegue, o después?

Para la próxima clase

- Terminar el laboratorio: el postmortem del incidente.
- **P4**: despliegue, observabilidad y postmortem de su proyecto. Entrega y demo en la semana 14, con la URL de producción. Lean qué dice su hosting del disco.
- Semana 14, agentes y productos con modelos de lenguaje. Lecturas: Stanford CS146S, *The Modern Software Developer*; MCP, *What is the Model Context Protocol?* y la especificación (*Overview* y *Transports*).

Pregunta

En su proyecto, si hoy alguien borra el contenedor y lo vuelve a crear con el comando equivocado, ¿qué se pierde y quién se entera primero?