VNIVERSITAT DE VALÈNCIA

Git, revisió i integració contínua en entorns Oracle

Material complet del curs

Raúl Salinas Monteagudo

2026-09-16

Taula de continguts

1 Objectiu del curs

Este material acompanya un curs introductori de quatre hores, en una única sessió, sobre Git, revisions i integració contínua aplicades a processos Oracle, SQL, scripts operatius i automatitzacions recurrents.

L'objectiu no és convertir l'equip en especialista de Git, sinó establir una manera més traçable, revisable i reproduïble de treballar. Estes pràctiques —traçabilitat dels canvis, revisió entre companys abans d'executar i capacitat de reproduir i recuperar— contribueixen també als objectius de l'Esquema Nacional de Seguretat (ENS) i al sistema de gestió de la seguretat del servei.

El curs no explica cada comanda amb detall: explica **quan** i **per què** s'usa, i quins criteris s'apliquen. El detall d'execució està en este manual i es pot consultar després, amb ajuda d'una IA si convé.

1.1 Guió de la sessió

Una única sessió de **quatre hores, de 10:00 a 14:00**.

Hora	Bloc	Pràctica
10:00–10:15	Per què: problemes operatius i el patró habitual	—
10:15–11:00	Git essencial: el cicle mínim, veure què ha canviat, recuperar una versió	Exercici 1
11:00–11:45	Branques, conflictes i revisió entre companys	Exercicis 2, 3 i 3b
11:45–11:55	Pausa	
11:55–12:15	Què versionem d'Oracle, què no hi va mai, quants repositoris	—
12:15–12:45	Entorn de proves efímer, dades sintètiques i traçabilitat	Demostració
12:45–12:55	Pausa	
12:55–13:35	Integració contínua: què comprova i els tres nivells	Exercici 5
13:35–13:50	Per on començar: el primer pilot	—
13:50–14:00	Tancament i preguntes	

L'entorn de proves efímer (exercici 4) es veu **en demostració**, no com a pràctica: queda escrit amb totes les passes per a qui el vulga repetir després amb calma. Els apartats de documentació executable, coneixement tàcit i configuració no s'imparteixen: queden ací com a lectura posterior, igual que l'exercici 6 i la major part de «Git en el dia a dia», del qual a la sessió només es veuen els gestos quotidians.

1.2 Objectius comprovables

En acabar la sessió, cada participant ha de poder fer estes coses sense ajuda. Són la manera de saber si el curs ha servit:

- llegir un `diff` i explicar què canvia;
- veure exactament què ha tocat un company en una branca o en un commit;
- recuperar l'estat anterior d'un fitxer, i distingir mirar, restaurar i revertir;
- fer commits menuts amb un missatge que diga el motiu;
- crear una branca, publicar-la i obrir una pull request;
- resoldre un conflicte senzill i explicar per què Git l'atura;
- revisar la pull request d'una altra persona amb criteri;
- identificar què no ha d'entrar mai al repositori;
- arrancar l'entorn de proves efímer i validar-hi un SQL;
- provocar una fallada de la pipeline i interpretar per què és roja;
- explicar la diferència entre integració, entrega i desplegament continu, i quin correspon a un procés que toca dades reals;
- explicar per què revertir un commit no desfà un `UPDATE` ja executat.

2 Problemes operatius que volem reduir

En una institució gran, no és suficient que un script funcione una vegada. També cal saber quina versió és l'oficial, per què es va modificar, qui la va revisar, quines comprovacions va superar i com podem recuperar-nos si alguna cosa falla. Els processos actuals han donat servei durant anys; Git i la integració contínua poden afegir eixa traçabilitat de manera **gradual i complementària**, sense exigir substituir el que ja funciona.

Problema	Risc	Millora amb Git/CI
Scripts dispersos	No se sap quina és la versió correcta	Repositori com a font de veritat
Canvis manuals	Poca traçabilitat	Commits i pull requests
Procediments orals	Dependència personal	Documentació versionada
Cada u té la seua còpia del mateix script	El mateix arreglament s'ha de fer N vegades, i se'n fan N−1	Un sol script parametritzat, compartit i revisat
Comprovacions manuals	Errors per omissió	Validacions automàtiques
Registre escrit a mà	Còpia i apega, omissions, reconstrucció a posteriori	Log automàtic de cada execució
Proves sobre dades canviants	Hui passa i demà no, sense tocar el codi	Joc de dades fix i determinista

Problema	Risc	Millora amb Git/CI
Processos programats sense definició versionada	La definició viu només dins de la màquina	Wrappers i definicions versionades
Credencials en scripts	Risc de seguretat	Separació de secrets
Dos canvis simultanis	L'últim que guarda sobreescriu l'altre	Conflicte explícit, mai silenciós
Configuració tocada a mà en producció	Divergix del que està escrit i ningú ho sap	Línia base exportada i comparació periòdica
No se sap quina versió està desplegada	No es pot reproduir ni diagnosticar	Commit desplegat registrat en cada execució
Error ja executat sobre dades reals	Pèrdua o corrupció de dades	**Git no ho resol**: cal un pla de recuperació a banda

L'última fila hi és a propòsit. La major part d'esta taula són problemes que el versionat i la validació automàtica redueixen de veritat, però n'hi ha almenys un que no: quan un `UPDATE` o un DDL ja s'ha executat, recuperar el codi anterior no recupera les dades. Conéixer la frontera és part d'usar bé la ferramenta, i l'apartat sobre configuració i reversibilitat hi torna amb detall.

3 Un patró habitual en entorns que han crescut

En entorns tècnics que han crescut durant anys és habitual trobar-se un mateix patró: scripts repartits entre carpetes, còpies identificades amb dates o números de versió, dependències de rutes locals i documentació escampada entre fitxers i correus. No és el defecte d'un lloc concret ni de ningú en particular: sol ser el que passa quan l'única ferramenta disponible durant molt de temps ha sigut una carpeta compartida. Pot ser suficient per al treball quotidià, i alhora dificultar saber quina versió és vigent, revisar els canvis i reproduir una execució.

3.1 Control de versions fet a mà

Sense historial, una manera de poder tornar arrere és inventar-se'l amb el nom del fitxer:

```
informe.sql
informe_v2.sql
informe_final.sql
informe_final_bueno.sql
```

Cada còpia va ser una decisió raonable en el seu moment. El problema no és fer còpies; és que, fetes a mà, no diuen qui, ni quan, ni per què, ni què va canviar respecte de l'anterior. Git guarda exactament això sense haver d'ocupar el nom del fitxer.

La substitució és literal, i val la pena veure-la: on hi havia quatre fitxers, en queda **un**, i totes les versions continuen accessibles.

```
git log --oneline -- informe.sql      # totes les versions, amb data, autor i motiu
git show HEAD~3:informe.sql           # com estava fa tres canvis
git diff HEAD~3 -- informe.sql        # què ha canviat des d'aleshores
```

El que es guanya no és estalviar-se tres fitxers: és que `informe_final_bueno.sql` no diu qui el va fer bo, ni quan, ni respecte de què, i `git log` sí. I no cal recordar cap convenció de noms, perquè el nom del fitxer torna a ser només el nom del fitxer.

3.2 Organització per persona o per any

Quan el material s'organitza per qui el fa o per l'any, per a trobar alguna cosa cal saber primer **qui** la va fer o **quan**. Organitzar per persona pot dificultar la continuïtat quan algú no està disponible —una excedència, un canvi de servei, una baixa, unes vacances—. Sol ser preferible agrupar per procés, producte o domini funcional, amb l'any com a etiqueta o paràmetre i no com a directori.

3.3 Cada u amb la seua còpia

Hi ha una cosa que passa abans que tot això, i que sol quedar invisible perquè no la veu ningú des de fora: **el codi quasi no es compartix**. La còpia d'un script viatja per correu, s'apega en un directori propi i allí es toca el que faça falta —un esquema, un centre, una data— fins que fa el que necessitava qui la va rebre.

El resultat no és una versió amb variants: són N scripts que fan quasi el mateix i que ja no es poden comparar. Quan apareix un error, l'arreglament s'ha de fer N vegades, i el que passa en la pràctica és que se'n fan N−1: sempre queda una còpia en algun lloc que ningú recorda.

I convé dir-ho abans de continuar: **duplicar no és descuit, és la conducta prudent**. Poseu-vos en el lloc de qui necessita que l'script d'un company accepte un paràmetre més. Les opcions reals són tres:

1. **Editar-l'hi.** Si es treballa amb un compte compartit, això li canvia l'script al company sense que ell se n'assabente. Inacceptable.
2. **Demanar-li-ho.** Sense cap canal, sense termini i sense manera d'ensenyar-li exactament quin canvi proposes. I li deus un favor.
3. **Duplicar-lo.** No molesta ningú i funciona hui.

La tercera és l'única que pot triar algú que treballa amb cura. El que falta, per tant, no és disciplina: és un **mecanisme per a proposar un canvi en el codi d'un altre**. Eixe mecanisme és la pull request, i té el seu apartat més avant.

L'alternativa no és demanar-li a la gent que comparteixa més, que és una petició sense conseqüències. És que compartir **valga la pena**, i això vol dir que l'script accepte com a **paràmetre** allò que cada u estava canviant a mà:

```
./ronda-escrivibles.sh --entorn preproduccio --data 2026-09-15
```

Un script parametritzat es pot compartir sense que ningú l'haja de tocar, i justament per això es pot revisar, corregir una sola vegada i provar automàticament. Un script que cal editar abans de cada ús no es compartix: es copia.

3.4 El coneixement, dins del nom o del correu

És habitual que part del coneixement viatge en el nom del fitxer (una instrucció per a qui el llegisca), en l'estat codificat en carpetes (`Pas_1/Fet`, `Pas_2/Fet`), o en correus guardats com a documentació. Tot això és documentació real, escrita per gent que volia deixar constància. El que sol faltar no és voluntat, sinó un lloc sistemàtic on posar-la i mantindre-la viva —perquè un document que ningú executa envelleix en silenci.

3.5 Codi lligat a una màquina

Un patró tècnic freqüent és el codi que només funciona en un ordinador concret: rutes absolutes de Windows, recursos de xarxa o rutes fixes dins d'un script.

```
C:\Users\usuari\Desktop\treball\...
\\servidor\recurs\...
/ruta/fixa/al/disc/...
```

Un script així no s'executa en el lloc d'un company, ni en un servidor, ni en una pipeline. La bona notícia és que això és **detectable automàticament**: una comprovació d'una línia el troba, i pot formar part de la pipeline.

```
grep -RInE "[A-Za-z]:\\\\|\\\\\\\\\[A-Za-z]|/ruta/fixa/" *.sql && exit 1 || true
```

3.6 Separar dades de codi

Abans de versionar res convé separar el codi de qualsevol dada personal o secret: noms amb identificadors, dades d'alumnat o credencials no haurien d'acabar mai en un repositori. Es versiona la font, mai les dades reals ni els secrets (com es tracten els secrets es detalla en el bloc d'Oracle).

3.7 En resum

Estes pràctiques han donat servei durant anys i van ser respostes raonables a les ferramentes disponibles en cada moment. Git i CI no invaliden el treball anterior: ofereixen una manera més sistemàtica de gestionar els mateixos problemes —saber quina versió és vigent, revisar els canvis i reproduir una execució— i es poden adoptar de forma **gradual**, procés a procés, sense substituir de colp res del que ja funciona.

4 Idees bàsiques de Git

Un flux mínim i raonable per a l'equip podria ser:

```
Petició o incidència
        ↓
Branca específica
        ↓
Canvis menuts i revisables
        ↓
Pull request
        ↓
Validacions automàtiques
        ↓
Revisió humana
        ↓
Integració en la branca principal
```

La branca principal hauria de representar una versió estable, revisada o almenys coneguda.

5 Git en el dia a dia

Les utilitats que fan que un equip li veja sentit a Git el primer dia no són les sofisticades: són les trivials. Saber exactament què ha tocat un company, què s'ha executat esta nit, com estava un fitxer abans, o quins fitxers formen realment un procés. Cap d'estes coses requerix entendre Git a fons, i totes resolen una fricció quotidiana.

5.1 Què ha canviat exactament?

És la pregunta més freqüent d'un equip i, amb una carpeta compartida, no té resposta: cal obrir les dues còpies i comparar-les a ull, si és que encara existixen les dues.

```
git diff                      # el que he tocat i encara no he consolidat
git diff main...validacio     # tot el que aporta una branca respecte de main
git show a183fe2              # un canvi concret, amb el seu motiu i el seu autor
```

La vista de diferències d'una pull request és exactament `git diff main...branca` presentat en el navegador i amb un lloc per a comentar cada línia. Qui revisa no ha de recórrer el fitxer sencer: només veu el que canvia.

La pregunta útil en el dia a dia no és «què és un diff?», sinó:

> Un company ha canviat el procés de càrrega. Què ha tocat exactament?

5.2 Funcionava fa una setmana

Quan una cosa deixa de funcionar, la primera pregunta és «què ha canviat?». Amb l'historial, això no depén de la memòria de ningú:

```
git log --oneline -- examples/validacio_cites.sql   # què li ha passat a este fitxer
git show a183fe2                                    # què deia eixe canvi i per què
git log -p -- examples/validacio_cites.sql          # tota la seua evolució, amb diffs
```

Git no servix només per a tornar arrere: servix per a **reconstruir la seqüència** que ha portat fins a l'estat actual. En una incidència, això sol ser més valuós que la reversió.

Quan el dubte és sobre una línia concreta i no sobre un fitxer sencer, `git blame` diu quin commit la va introduir. És només la porta d'entrada: el que interessa és el commit, que es mira després amb `git show`.

```
git blame examples/create_user.sql
```

Convé dir-ho perquè el nom de la comanda és desafortunat: no servix per a buscar culpables, sinó per a recuperar el context i la decisió que hi havia darrere.

5.3 Recuperar una versió anterior

Tres operacions que es confonen sovint i que fan coses distintes:

Vull…	Comanda	Què fa
Mirar com estava un fitxer abans	`git show HEAD~3:informe.sql`	El mostra, sense tocar res
Tornar un fitxer a com estava	`git restore --source=HEAD~3 informe.sql`	El deixa així en el meu directori
Desfer un canvi ja integrat	`git revert a183fe2`	Crea un commit nou que el desfà

En una branca compartida convé `git revert` i no reescriure l'historial: el commit de reversió queda com un canvi més, amb el seu motiu i la seua revisió, i no trenca la còpia de ningú.

5.4 Quins fitxers formen realment el projecte

`git ls-files` llista el que està versionat. No diu què és important —això no ho sap Git—, però en un repositori ben mantingut és **l'inventari intencional** del projecte: el que algú va decidir que en formava part.

```
git ls-files              # tot el que hi ha realment al projecte
git ls-files '*.sql'      # només el SQL
git ls-files | wc -l      # quantes peces són
```

És la cara complementària del `.gitignore`: fora queden els generats, els logs, els temporals i tot el que no és font. La diferència amb un `ls -R` sobre una carpeta compartida és precisament eixa: `ls` mostra el que hi ha, `git ls-files` mostra el que s'ha decidit tindre.

I una conseqüència directa: buscar només dins d'eixe corpus.

```
git grep DBMS_SCHEDULER       # quins processos programats toquem
git grep -n PREPROD           # on apareix preproducció
git grep 'GRANT DBA'          # el que no hauria d'aparéixer enlloc
```

Un cas d'ús que ha aparegut de seguida: quan volem donar context d'un projecte a una IA, `git ls-files` permet seleccionar el corpus rellevant en compte d'enviar un directori sencer amb els seus binaris, els seus generats i els seus temporals.

> **Compte:** que un fitxer estiga versionat **no vol dir** que estiga autoritzat enviar-lo a una IA externa. Són dues decisions independents: una és tècnica i l'altra és de protecció de dades i de política institucional.

5.5 Dues persones toquen el mateix fitxer

Passa, i és el cas on la diferència amb una carpeta compartida és més gran:

- **En una carpeta compartida**, l'últim que guarda sobreescriu el treball de l'altre. En silenci, sense avís, i normalment ningú se n'assabenta fins que falta alguna cosa.
- **En Git**, si els dos canvis no es poden combinar automàticament, la integració s'atura i obliga algú a decidir explícitament quin resultat és el correcte.

Dit d'una altra manera: en una carpeta compartida **guanya l'últim** que guarda, i el treball del primer desapareix sense que ningú ho decidisca. En Git **guanya el primer** que integra, i qui arriba després s'ho ha d'apanyar. Això no és un inconvenient: la faena extra recau precisament en l'únic que té els dos canvis davant i pot decidir bé.

Un conflicte no és una avaria: és Git negant-se a triar per tu en l'únic cas en què no pot fer-ho sense inventar. Resoldre'l és editar el fitxer deixant el que ha de quedar i llevant els marcadors:

```
<<<<<<< HEAD
WHERE data_inici <= SYSDATE
=======
WHERE data_inici <= TO_DATE('&&data_referencia', 'YYYY-MM-DD')
>>>>>>> validacio-cites
```

La revisió humana decidix el contingut; la pipeline hi posa la xarxa de baix. Les comprovacions estàtiques d'este curs busquen precisament eixos marcadors, perquè una resolució feta amb pressa i a mitges és un error fàcil i completament detectable.

5.6 Canvis menuts i el motiu del canvi

Dues disciplines que no són estètiques: abaraten tot el que hem vist ací dalt.

Un commit, una cosa. Un commit que barreja una correcció funcional, un reformatat, un canvi de configuració i un parell de renoms és difícil de llegir, difícil de revisar, difícil de diagnosticar mesos després i impossible de revertir sense arrossegar la resta. La revisió d'un canvi menut és barata, i per això es fa.

El missatge diu per què, no què. El *què* ja el diu el `diff`; escriure'l una altra vegada en el missatge no aporta res:

```
Canvia timeout a 60                                  ← el diff ja ho diu
Evita timeout durant la càrrega de matrícula (INC-1234)  ← això no està enlloc més
```

Amb la referència al tiquet, `git log` deixa de ser una llista de modificacions i passa a ser el registre de les decisions. Mesos després, quan algú es pregunte per què eixe valor és 60, la resposta està a un `git show` de distància.

La forma verbal. En anglés la convenció és l'imperatiu —*Add validation*— perquè el missatge completa la frase «si s'aplica, este commit…». En valencià la traducció natural és el **present**, i hi ha una coincidència còmoda: per a quasi tots els verbs, la tercera persona del present i l'imperatiu són la mateixa paraula. Així, «Afig validació de cites duplicades» es llig igual de bé com «este commit afig…» que com una ordre. No cal triar.

no això	això
Afegir validació de cites duplicades	**Afig** validació de cites duplicades
S'ha afegit la validació de cites	**Afig** validació de cites duplicades
Vaig canviar el timeout	**Evita** timeout durant la càrrega (INC-1234)

L'infinitiu i el passat fallen pel mateix motiu: descriuen el que va fer una persona, i el que interessa és el que fa el canvi.

I què passa amb Conventional Commits? És un estàndard que demana prefixar cada missatge amb el tipus de canvi —`feat:`, `fix:`, `chore:`— i existix per a una cosa molt concreta: que una ferramenta puga calcular a soles el número de versió i el registre de canvis d'un producte que es publica. On es publiquen artefactes versionats, val la pena. En un repositori de scripts d'operació, que no es publica amb números de versió, no compra res i afig un ritual que acaba fent-se malament.

Si algun dia un prefix ajuda, el que sol servir de veres és el **procés** que toca el canvi —`matricula: evita timeout en la càrrega (INC-1234)`—, perquè fa que `git log --oneline` es puga recórrer amb la vista. Això no és l'estàndard; és només un prefix, i funciona.

5.7 Fusionar o rebasar

Quan la teua branca i `main` han divergit, hi ha dues maneres d'incorporar el que ha passat en `main`, i no fan el mateix. És de les coses que més dubtes generen, així que val la pena mirar-la amb calma.

```
              A ── B     ← la teua branca
             /
o ── o ── o ── X ── Y    ← main
```

`git merge main` crea un commit nou amb **dos pares**:

```
              A ── B ─────\
             /             \
o ── o ── o ── X ── Y ──────── M   ← la teua branca
```

No reescriu res: `A` i `B` es queden exactament com van passar, i l'historial mostra que hi va haver dues línies i quan es van ajuntar. Els conflictes es resolen una sola vegada, en el moment de fusionar. L'única pega és estètica: amb molta gent, l'historial s'ompli de bifurcacions i de commits de fusió que no expliquen res.

`git rebase main` torna a aplicar els teus commits damunt de `main`:

```
o ── o ── o ── X ── Y ── A' ── B'   ← la teua branca
```

La prima és el que importa: **`A'` no és `A`**. Git no mou commits, els torna a crear amb el mateix contingut i un hash distint. L'historial queda lineal, com si hagueres començat el treball hui; a canvi es perd la informació de quan es va integrar, i els conflictes poden aparéixer commit a commit —amb sis commits, pots haver de resoldre el mateix sis vegades—.

	`merge`	`rebase`
L'historial	fidel: es veu que hi va haver dues línies	lineal i fàcil de llegir
Els teus commits	queden intactes	es reescriuen, amb hash nou
Conflictes	es resolen una vegada	poden repetir-se commit a commit
Es veu quan es va integrar	sí	no
En una branca compartida	segur	**mai**

La regla és una i no admet excepcions: rebasa només la teua branca, i només mentre no l'haja baixada ningú. Reescriure commits que un company ja té al seu clon és el mateix problema que hi ha en l'apartat següent, i acaba en un `push --force` damunt de faena d'un altre.

Per a un equip que comença, el consell barat és **`merge` per defecte**: no té cap manera d'eixir malament. El `rebase` es reserva per a dues coses molt concretes: netejar la teua branca abans d'obrir la pull request, i posar-te al dia amb `git pull --rebase`, que evita que cada `pull` deixe un commit «Merge branch main of…» que no aporta res i embruta la revisió. Això últim es pot deixar posat per sempre:

```
git config --global pull.rebase true
```

5.8 Reescriure l'historial

Fins ací tot són operacions que **afigen**: un commit nou, un `revert` que en crea un altre que desfà. Git també permet **reescriure** el que ja hi ha —`commit --amend`, `rebase -i`, `filter-repo`—, i això és una altra categoria d'operació que convé entendre abans de tocar-la.

Reescriure no modifica els commits: en **crea uns altres** amb un identificador distint i abandona els vells. L'historial queda igual de net a la vista, però tots els commits a partir del punt reescrit són objectes nous.

Per què això trenca una branca compartida. Els clons dels altres apunten als commits antics, que ja no formen part de la branca. Quan el següent company faça `git pull` es trobarà les dues històries alhora i, si no sap què ha passat, la manera natural de resoldre-ho és tornar a pujar els commits vells. Per a publicar un historial reescrit fa falta `git push --force`, i eixe `--force` vol dir literalment «descarta el que hi ha al servidor». Si algú havia pujat alguna cosa entremig, es perd.

Per això el repartiment habitual és:

Branca	Reescriure
`main` i altres branques compartides	**Mai.** Per a desfer, `git revert`
La teua branca de treball, encara sense fusionar	Endavant, i sovint convé

En la pràctica no sol dependre de la disciplina de ningú: en Bitbucket, com en qualsevol altra plataforma, les branques principals es configuren com a **protegides** i el servidor rebutja el `push --force`. Que no et deixe no és una limitació: és la xarxa funcionant.

En la branca pròpia sí que convé. Mentre proves, els commits reals són «ara sí», «prova 3», «torna arrere lo d'abans». Eixe historial no ajuda ningú a revisar. Abans d'obrir la pull request, `git rebase -i` permet ajuntar-los en un o dos commits que expliquen el canvi de veres. El que revisa el teu company és això, no el camí que vas fer per a arribar-hi.

Si has de forçar, **`git push --force-with-lease` en compte de `--force`**: fa el mateix, però falla si algú havia pujat res des de l'última vegada que vas mirar, en compte de passar-li per damunt.

El cas del secret. És l'únic on reescriure l'historial d'una branca compartida està justificat, i encara així és el **tercer** pas, no el primer. Netejar l'historial obliga tothom a tornar a clonar, i no lleva la credencial dels clons que ja existixen ni dels servidors on va arribar. La credencial es rota primer; la neteja és higiene posterior.

6 Revisió de canvis

La revisió és el pas que convertix un canvi en un canvi **acordat**. No és una capa de burocràcia damunt del treball: és el moment en què una segona persona mira què es va a executar, abans que s'execute, i en què el motiu d'un canvi queda escrit on el trobarà qui vinga després.

6.1 Què és una pull request

Una pull request (en altres ferramentes, *merge request*) és la proposta d'integrar una branca dins d'una altra. No és res més que això, però al voltant du les quatre coses que fan útil la revisió:

- el **context**: el títol, la descripció i el tiquet que la motiva;
- les **diferències** línia a línia, que és exactament `git diff main...branca` presentat en el navegador;
- un lloc per a **comentar cada línia**, i no el fitxer sencer;
- l'**aprovació**, que queda registrada amb qui la va donar i quan.

Eixe últim punt sol interessar fora de l'equip: sense muntar cap circuit a banda, queda constància de qui va revisar què, que és el que demana un sistema de gestió de la seguretat quan pregunta com es controlen els canvis.

6.2 No és control personal

És la renuència previsible, i convé dir-la abans que la pense ningú. Una revisió no és una inspecció del treball d'una persona. És:

- **transmissió de coneixement**: qui revisa aprén com funciona eixe procés, i l'endemà el pot agafar si fa falta;
- **detecció d'errors** en el moment més barat de tots, que és abans d'executar;
- **criteris compartits**: les discussions que es tenen en tres o quatre revisions acaben sent la manera de treballar de l'equip, sense que ningú haja hagut d'escriure una normativa;
- **continuïtat del servei**: cap procés queda amb una sola persona que sàpia com va.

I hi ha una cosa més, que sol passar desapercebuda perquè no s'anomena mai: una pull request és **el canal per a proposar un canvi en el codi d'un altre**. Sense ell, qui necessita que un script d'un company accepte un paràmetre més només pot editar-l'hi el fitxer, demanar-li-ho de paraula o duplicar-lo; i duplicar-lo és l'única de les tres que no molesta ningú. Les còpies que divergixen no són un problema de disciplina: són el que passa quan no existix esta manera de demanar.

Qui més guanya sol ser qui escriu el canvi. Un canvi revisat és un canvi que ja no és teu a soles, i això lleva el telèfon de damunt de la taula en agost.

6.3 Què es mira en una revisió

Amb SQL i scripts d'operació, la llista és curta i quasi sempre la mateixa:

Pregunta	Per què
Té tiquet o explicació?	El *per què* no està en cap altre lloc
El `WHERE` és prou restrictiu?	És l'error que no es desfà amb Git
Si modifica dades, du `WHENEVER SQLERROR EXIT`?	Sense això, un error a mitjan script deixa les dades a mitges
Hi ha una comprovació prèvia i una posterior?	Saber abans si toca fer-ho, i després si ha eixit bé
Es pot tornar arrere? Com?	Escriure-ho abans, no a les 3 de la matinada

Pregunta	Per què
Hi ha cap secret?	Una vegada consolidat, l'historial el guarda per sempre

Bona part d'això la pot mirar una màquina, i convé que la mire: el que la pipeline atrapa sola no s'ha de discutir en una revisió. El que queda per a les persones és el que una màquina no pot contestar —si el canvi és el correcte, si el motiu es sosté, si hi ha una manera més simple—, i eixe és precisament el temps que val la pena gastar.

6.4 Canvis menuts

Una revisió és barata quan el canvi és menut, i cara quan no ho és. No és una qüestió d'estil: una pull request de vint fitxers amb un arreglament funcional, un reformatat i un parell de renoms no es revisa de veres, s'aprova. Si voleu que la revisió es faça, feu-la barata.

6.5 Dos toquen la mateixa línia

Passa, i és on la diferència amb una carpeta compartida es nota més. En una carpeta compartida guanya **l'últim** que guarda i el treball del primer desapareix sense que ningú ho decidisca; en Git guanya **el primer** que integra, i qui arriba després ha de mirar-se els dos canvis i triar.

Això no és un inconvenient: la faena extra recau en l'únic que té els dos canvis davant i pot decidir bé. La revisió humana decidix el contingut, i la pipeline busca els marcadors oblidats, que és l'error fàcil quan es resol amb pressa.

7 Aplicació directa a SQL i Oracle

En un equip Oracle es pot versionar molt més que codi d'aplicació. La pregunta útil no és *què podem posar en Git*, sinó: **si açò es perdera o canviara sense avisar, quant costaria adonar-se'n i reconstruir-ho?**

7.1 Base de dades

- scripts SQL operatius i consultes d'informe;
- DDL i migracions, amb ordre i versió;
- paquets, procediments, funcions i disparadors PL/SQL;
- vistes, sinònims i definicions d'índexs;
- definicions de rols, grants i perfils;
- usuaris i la seua configuració, sense contrasenyes;
- jobs, cadenes i finestres de `DBMS_SCHEDULER`;
- polítiques d'auditoria i de seguretat de files;
- paràmetres d'instància rellevants que no siguen secrets;
- dades mestres poc canviants: codis de centre, titulacions, calendaris acadèmics, taules de conversió;

- jocs de dades sintètics per a proves;
- validacions funcionals i de negoci.

Les dades mestres solen ser l'oblit més car: viuen dins de la base, canvien poc, ningú recorda qui les va posar i, quan una es modifica per error, no hi ha manera de saber què hi havia abans.

7.2 Sistema i execució

- wrappers de Bash i scripts d'operació;
- definicions de `cron` i unitats de `systemd`;
- rotació de logs i neteja de temporals;
- configuració de connexions: `tnsnames.ora`, `sqlnet.ora`, `ldap.ora`, àlies i serveis, **sense credencials**;
- inventari de màquines, serveis i responsables;
- variables d'entorn no secretes i perfils d'execució;
- configuració de servidors web i proxies inversos;
- certificats públics, mai les claus privades;
- `Dockerfile`, `docker-compose.yml` i definicions d'entorn;
- playbooks o rols d'automatització;
- definicions de pipelines de CI.

La configuració de connexions és un cas especialment agraït: és informació que tothom necessita i que sovint està distribuïda entre diversos equips, sistemes i entorns. Versionar-la no té cap risc mentre no incloga secrets, i estalvia moltes consultes (`examples/inventory.yaml`).

7.3 Documentació del procés

- `README.md` per procés, amb què fa, quan s'executa i qui el manté;
- procediments d'operació i de recuperació;
- calendari dels processos crítics: matrícula, actes, beques, canvi d'any;
- llistes de comprovació de posada en producció;
- decisions tècniques i el seu motiu, encara que després es descarten;
- diagrames escrits com a text, no com a imatge exportada;
- el material de formació —este mateix, sense anar més lluny.

7.4 Què no ha d'anar mai

- contrasenyes, wallets, claus privades, tokens i cookies;
- volcats de producció i qualsevol dada personal real;
- fitxers generats a partir d'una altra cosa del repositori;
- binaris grans que canvien sovint;
- logs d'execució.

Dels secrets sí que pot anar la **referència**: quin secret fa falta, on es guarda i qui el gestiona. El que no pot anar és el valor.

7.5 Si un secret ja ha entrat

És un cas prou important i prou mal entés per a dir-lo explícitament: **esborrar la línia en el commit següent no lleva el secret del repositori**. Continua en l'historial, i qualsevol persona que ja tinga un clon el té complet en el seu disc.

Per tant l'ordre de les accions no és opinable:

1. **Rotar o revocar la credencial**, immediatament. És l'única passa que realment tanca el problema, i no depén de Git.
2. Traure-la del codi i posar-la on toca: wallet, gestor de secrets o variable protegida de la CI.
3. Netejar l'historial, si es considera necessari. És una operació posterior, incòmoda —reescriu tots els commits i obliga tothom a tornar a clonar— i mai substituïx la primera.

Els detectors de secrets (`gitleaks`, `trufflehog`) poden escanejar també l'historial complet, no sols l'últim commit. Val la pena passar-lo una vegada abans de donar per bo un repositori que ve d'una migració.

7.6 Una regla senzilla

> Es versiona la font, no el producte.

Este repositori n'és un exemple: els `.qmd` i els exemples estan versionats; el web generat en `_site/` no, perquè es pot tornar a produir en qualsevol moment a partir de la font.

La idea de fons és agrupar en una unitat traçable:

```
SQL
  + paràmetres
  + documentació
  + validació prèvia
  + validació posterior
  + revisió
  + versió
```

8 Baixar a Git el codi que ja viu dins d'Oracle

El codi PL/SQL d'una aplicació que ja funciona no viu en fitxers: viu dins de la base de dades. Els `.sql` que hi ha escampats pel disc són còpies d'edat desconeguda, i normalment ningú pot dir quina d'elles és la que està compilada en producció. Per tant la primera passa no és «muntar Git»: és **baixar el codi de la base de dades de manera que demà es puga tornar a baixar i comparar**.

8.1 Els paquets no necessiten migracions

Un paquet es desplega amb `CREATE OR REPLACE`, que és **idempotent**: el fitxer *és* l'estat final. No fa falta saber quins canvis s'han aplicat ja a cada entorn, perquè no s'apliquen canvis, es posa el contingut.

Objecte	Com es versiona
Paquets, procediments, funcions, vistes, disparadors	Git + `CREATE OR REPLACE`
Taules, índexs, restriccions, dades mestres	Ací sí que cal migracions: Liquibase o scripts numerats amb taula de control

Esta és tota la diferència. Mentre es parle de codi, Git ja fa la faena; la ferramenta de migracions apareix quan es toquen estructures que no es poden tornar a crear damunt.

8.2 L'exportació inicial: dos comandaments

SQLcl porta el comandament `project` des de la versió 23.4. Fa exactament açò: exporta els objectes a una jerarquia de fitxers pensada per a un repositori.

```
mkdir paquets-ewp && cd paquets-ewp && git init
sql usuari@servei          # o sqlx sqlcl <perfil>
```

```
project init -name ewp -schemas EWP
project export -list       -- assaig: llista què eixiria, sense escriure res
project export             -- de veres
```

`project init` deixa l'esquelet:

```
.dbtools/project.config.json     # transformacions de DBMS_METADATA
.dbtools/filters/project.filters # predicats SQL per a triar objectes
.gitignore
src/database/ewp/                # un fitxer per objecte, per tipus
dist/
```

8.3 Només el codi que interessa

Per defecte exporta tot l'esquema, i «tot» inclou coses que no volem en un repositori de codi. En un esquema real de preproducció, `project export -list` llista **1.357 objectes**, dels quals 849 són `GRANT`. Tres línies afegides al final de `.dbtools/filters/project.filters` ho deixen en 18:

```
-- Només codi PL/SQL: paquets
export_type = 'ALL_OBJECTS',
object_type in ('PACKAGE','PACKAGE BODY'),
```

Els predicats s'afigen a les consultes del diccionari, i la coma final hi va. Amb el filtre posat, l'exportació completa d'eixe esquema —9 especificacions i 9 cossos, 560 KB— tarda **7 segons**.

De propina, en acabar avisa dels objectes que estan invàlids a la base de dades. Això sol és informació que molts equips no tenen a mà.

8.4 Per què no un script propi de `DBMS_METADATA`

Perquè el que costa no és cridar `GET_DDL`, és que el resultat siga comparable. `project init` escriu estes transformacions:

```
"segmentAttributes": false, "storage": false,
"tablespace": false, "partitioning": false
```

Sense elles, cada DDL arrossega `INITIAL 65536 NEXT 1048576 PCTINCREASE 0 TABLESPACE ...`, i el primer canvi d'espai ja mou fitxers que ningú ha tocat. Un `git diff` que s'omplí de soroll deixa de llegir-se a la setmana, i un repositori que ningú llig no servix per a res.

8.5 La prova que l'exportació val

```
project export     # una segona vegada, sense tocar res
git status         # ha d'eixir buit
```

Si la segona exportació mou fitxers, l'exportació no és determinista i encara no es pot confiar en cap `diff` posterior. Això es corregix **abans** del primer commit, no després.

8.6 El primer commit, cru

Tal com ix, encara que el format siga lleig i no seguisca cap estil de la casa. Formatar, reordenar o renomenar va en un segon commit separat. Si les dues coses entren juntes, el dia que algú busque qui va canviar una condició es trobarà 40.000 línies de sagnat en el mateix commit que el canvi real, i l'historial ja no li contestarà.

8.7 El regal: detectar desviaments

Amb el repositori poblat, el mateix comandament respon a una pregunta que abans no tenia resposta:

```
project export && git diff --stat    # si ix res, algú ha tocat la base de dades per fora
```

Un `cron` setmanal amb estes dues comandes val més que qualsevol norma escrita sobre no tocar producció a mà.

8.8 SQL Developer i Git

SQL Developer porta un client de Git integrat (basat en JGit, al menú **Team** i a les finestres *Versions* i *Pending Changes*). Fa el cicle quotidià sense eixir de la ferramenta: clonar, afegir, *commit*, *push*, *pull*, canviar de branca, veure l'historial d'un fitxer i comparar-lo amb la versió anterior. Per a qui viu dins de SQL Developer, és suficient.

El que no fa bé és la resta: resoldre conflictes, reescriure historial, o triar quins trossos d'un fitxer entren en un *commit*. Ahí es baixa al terminal, i no passa res per fer-ho.

Però la limitació important no és cap d'estes. És esta:

> Si el paquet s'obri amb el **navegador de connexions**, s'està editant el codi de la base de dades. Si s'obri amb **File → Open**, s'està editant el fitxer del repositori.

Són dos camins que es pareixen molt en pantalla i que porten a llocs oposats. El primer compila un canvi que Git no veurà mai. La regla de treball, doncs, és que **els paquets s'obrin com a fitxers** i s'executen contra la connexió triada amb *Run Script* (`F5`), que és el que aplica el `CREATE OR REPLACE` del fitxer sencer. El navegador queda per a consultar, no per a editar.

8.9 I Liquibase?

Mireu la configuració que ha escrit `project init`:

```
"stage": { "generatedFormat": "liquibase" }
```

El mateix comandament genera *changelogs* de Liquibase per a la fase de desplegament, i SQLcl ja el porta dins. O siga: **s'exporta a Git sense Liquibase, i el dia que faça falta per a les taules ix de la mateixa ferramenta**. No hi ha res a decidir al principi, que és justament quan menys se sap.

9 Com repartim el treball en repositoris

Abans de moure cap fitxer cal respondre una pregunta que després costa molt de canviar: **quants repositoris fem i què posem dins de cadascun?**

9.1 Provar junt el que canvia junt

La validació sol ser més senzilla quan els components que canvien junts es poden provar de manera coordinada. Una pipeline s'engega amb un commit d'un repositori i valida sobretot **eixe** repositori; pot coordinar-ne diversos, però com més coses relacionades es proven juntes, més senzill sol ser el conjunt.

Si dues coses que depenen l'una de l'altra viuen en repositoris distints:

- no existeix cap commit que represente "les dues juntes"; no hi ha, per tant, cap estat conegut del conjunt al qual tornar;

- un canvi que afecta les dues són dos commits, dues revisions i dues pipelines, i entremig hi ha una finestra en què el conjunt està trencat;
- cap de les dues pipelines prova el que de veritat importa, que és la combinació.

D'ací ix la regla pràctica:

> El que canvia junt, viu junt.

9.2 Criteri per a decidir

Al mateix repositori quan:

- se solen modificar en el mateix canvi;
- comparteixen esquema, dades mestres o convencions;
- es desplegen alhora;
- es validen amb les mateixes proves.

En repositoris separats quan:

- tenen cicles de vida independents;
- els permisos han de ser distints, per exemple si un conté dades sensibles;
- els manté gent diferent sense coordinació habitual;
- el repositori es fa tan gran que la pipeline es torna lenta de manera insuportable.

La mida no sol ser el criteri decisiu: solen pesar més els permisos, les dependències, els responsables i la freqüència dels canvis.

9.3 Submòduls

Git permet incloure un repositori dins d'un altre com a **submòdul**: el pare no guarda el codi del fill, sinó **un punter a un commit concret** d'eixe altre repositori. Això té un avantatge real: el pare deixa constància exacta de **quina versió del fill utilitza**, i eixa referència queda versionada.

El que no resolen és l'atomicitat: un canvi que toque pare i fill segueixen sent dos commits en dos repositoris, i cal recordar moure el punter (i clonar amb `--recursive`). Per això són especialment útils per a **incorporar codi que evoluciona pel seu compte** —una llibreria, codi de tercers—; per a repartir faena pròpia que canvia junta, sol eixir més còmode un sol repositori.

9.4 Un exemple que ajuda a valorar-ho

Hi ha projectes de programari molt grans que es mantenen en un únic repositori Git sense fragmentar-lo, i molts projectes menuts repartits en diversos repositoris. Això suggereix que la **mida** rarament és el factor determinant: solen pesar més la coherència del que canvia junt, els permisos, els responsables i les dependències entre components.

9.5 Traducció al nostre cas

Una divisió raonable seria per **domini funcional**, amb carpetes per procés a dins:

```
matricula/          cites, càrregues, validacions, calendari
cobros-rebuts/      remeses, liquidacions, conciliació
efectius-nomines/   consultes i processos de personal
infraestructura/    inventari, entorns, jobs, plantilles
```

I un consell d'ordre: **si hi ha dubte, comenceu amb menys repositoris dels que sembla que fan falta.**

No és perquè dividir siga una operació més senzilla que ajuntar. Les dues es poden fer, i cap de les dues és especialment difícil: `git filter-repo` extrau una carpeta amb el seu historial sencer, i dos repositoris s'ajunten amb un `merge`. És que els dos errors no costen el mateix:

- Si heu posat junt el que havia d'anar separat, ho pagueu **una vegada**: el dia que ho dividiu.
- Si heu separat el que havia d'anar junt, ho pagueu **cada vegada** que un canvi toque els dos costats: dos commits, dues revisions, dues pipelines i una finestra entremig en què el conjunt està trencat. I ajuntar-los després no recupera res, perquè els canvis que ja s'han fet continuen sense ser atòmics.

A més, la decisió de dividir es pren amb la informació de hui, que és la pitjor que tindreu mai sobre com evolucionarà això.

10 Exemple: alta d'un usuari Oracle

Un alta d'usuari pot passar d'una execució manual a un procés versionat:

```
provisioning/users/
├── request.yaml
├── create_user.sql
├── validate_user.sql
└── README.md
```

Exemple de petició declarativa:

```yaml
request: SIUV-401234
username: APP_ORDENAMAT
profile: PROFILE_APPLICATION
default_tablespace: USERS
temporary_tablespace: TEMP
roles:
  - ROLE_ORDENAMAT
environments:
  - preproduction
  - production
```

Esta definició no conté contrasenyes. Els secrets han de viure fora del repositori: wallet, gestor de secrets, variables protegides de CI o entrada interactiva controlada.

11 Exemple: processos programats

«Què s'executa cada nit?» hauria de tindre una resposta que no consistisca a entrar en una màquina concreta i llegir un `crontab`. Els processos programats són, precisament, els que ningú mira mentre funcionen i els que ningú sap explicar quan fallen.

11.1 Què es versiona d'un procés programat

No sols la definició del `cron` o del job de `DBMS_SCHEDULER`:

- el **wrapper** que realment s'executa, que és on està la lògica;
- la **configuració no secreta**: àlies de base de dades, rutes, límits, finestres;
- la **referència** al secret que necessita —quin és i qui el gestiona—, mai el valor;
- el `README.md` amb què fa, quan s'executa, què cal mirar si falla i qui el manté.

Amb això, la pregunta anterior es contesta llegint el repositori.

11.2 El registre que deixa cada execució

El wrapper d'exemple (`examples/run-job.sh`) anota data, host, base de dades i, sobretot, **el commit que s'estava executant**:

```
#!/usr/bin/env bash
set -euo pipefail

SCRIPT_DIR="$(cd -- "$(dirname -- "${BASH_SOURCE[0]}")" && pwd)"
REPO_DIR="$(git -C "$SCRIPT_DIR" rev-parse --show-toplevel)"
COMMIT="$(git -C "$REPO_DIR" rev-parse --short HEAD)"
LOG_DIR="${LOG_DIR:-$REPO_DIR/logs}"
DB_ALIAS="${DB_ALIAS:-PREPROD}"

mkdir -p "$LOG_DIR"

echo "start_time=$(date '+%Y-%m-%d %H:%M:%S')"
echo "host=$(hostname)"
echo "commit=$COMMIT"
echo "db_alias=$DB_ALIAS"

# Exemple docent. Substituir per sqlplus/sqlcl segons l'entorn real.
# sql -L "/@${DB_ALIAS}" @"$REPO_DIR/examples/validacio_cites.sql"

echo "end_time=$(date '+%Y-%m-%d %H:%M:%S')"
```

Eixa última línia és la que lliga les dues meitats del problema. Considereu una incidència real:

> A les 08:00 es detecta que la càrrega de la nit ha deixat dades incorrectes. El log de les 03:00 diu `commit=a183fe2`.

```
git show a183fe2          # exactament el codi que es va executar, i el seu motiu
git log --oneline a183fe2..HEAD   # què ha canviat des d'aleshores
```

En trenta segons se sap quin codi va córrer, qui el va escriure, per què i què s'ha tocat després. Sense eixa línia, la mateixa pregunta és una conversa amb tres persones i una reconstrucció aproximada.

La diferència de fons és que este registre **el genera la màquina**: no depén que algú recorde apuntar-lo, i no pot dir una cosa distinta de la que va passar.

11.3 Quina versió està realment en producció

És la mateixa idea, generalitzada. Una versió desplegada ha de poder relacionar-se amb un commit, una etiqueta o un registre de desplegament. Si «producció» vol dir només «els fitxers que ara mateix hi ha en aquella màquina», no hi ha manera de saber si coincidixen amb cap estat conegut del repositori, ni de reproduir-los.

11.4 Versionar i orquestrar són dos passos

Com es llancen i es centralitzen eixos treballs és una decisió a part que Git no resol: el `cron` del sistema, `DBMS_SCHEDULER` o una ferramenta d'orquestració dedicada (per exemple, una plataforma d'automatització de fluxos com n8n) són opcions a valorar. Versionar els scripts es pot fer hui i no obliga a triar orquestrador.

12 Entorns reproduïbles i dades de prova

12.1 "En la meua màquina funciona"

Un script pot funcionar en el portàtil d'una persona i fallar en producció per motius que no estan en el codi: una versió distinta d'Oracle o del client, un joc de caràcters diferent, una variable `NLS_`, una zona horària, un paquet del sistema que estava instal · lat des de fa anys i ningú recorda per què.

Mentre l'entorn viu en la memòria de qui el va muntar, no és reproduïble. Un contenidor sí que ho és: la definició de l'entorn passa a ser un fitxer versionat, revisable en una pull request com qualsevol altre canvi.

Això ens dona tres coses:

- la CI executa exactament el mateix entorn que qualsevol persona de l'equip pot arrancar en local;
- una persona nova té l'entorn muntat en minuts, sense una llista de passos manuals;
- actualitzar la versió d'Oracle de proves és un commit, amb el seu historial i la seua revisió.

12.2 Una tasca està acabada quan passa la CI

Convé fixar este criteri de manera explícita, perquè canvia la manera de treballar:

> Una tasca no està acabada quan funciona en el meu entorn, sinó quan passa les validacions automàtiques.

No és burocràcia. És l'única definició d'"acabat" que no depén de la màquina, del dia ni de la persona. Si la CI és verda, qualsevol company pot reproduir el mateix resultat; si no ho és, encara no sabem si el canvi funciona.

12.3 Pensar les condicions de prova

Per a poder validar un canvi automàticament cal respondre abans una pregunta que sovint no ens fem: **de què depén realment este procés?**

- quines taules necessita;
- quines dades mínimes fan que el cas tinga sentit;
- quins casos límit hauria de cobrir;
- què hauria de passar quan les dades són incorrectes.

Este exercici sol ser el més valuós de tot el procés, encara abans d'escriure cap prova: obliga a explicitar dependències que fins ara només estaven en el cap d'algú.

12.4 Els permisos també són una dependència

La llista anterior es queda curta en un punt: un procés no depén només de taules i de dades, depén també **d'allò que té permís per a fer**. I això sol descobrir-se en el pitjor moment, quan el procés ja porta set passes i s'atura amb un `ORA-00942` que, a més, no distingix entre «la taula no existix» i «existix i no la veus».

Escrit de manera declarativa, es paga tres vegades amb el mateix esforç:

```
process: validacio-cites
objects:
  - MATRICULA.CENTRE_FRANJA: SELECT
  - MATRICULA.ESTUDIANT: SELECT
role: ROLE_ORDENAMAT
```

- és **documentació que no pot envellir en silenci**, perquè es comprova;
- és una **comprovació prèvia**: abans de fer res, el procés verifica que té el que necessita i falla clar i d'immediat si no;
- és la **petició de permisos**, i com que està escrita al detall, ningú acaba demanant DBA per si de cas.

La comprovació prèvia és curta, i té una propietat que la fa millor del que sembla:

```
WHENEVER SQLERROR EXIT FAILURE

DECLARE
    falten PLS_INTEGER;
BEGIN
    SELECT COUNT(*) INTO falten
      FROM (SELECT 'MATRICULA.CENTRE_FRANJA' AS obj FROM dual
            UNION ALL
            SELECT 'MATRICULA.ESTUDIANT' FROM dual) requerits
     WHERE NOT EXISTS (SELECT 1 FROM all_tables
                       WHERE owner || '.' || table_name = requerits.obj);

    IF falten > 0 THEN
        raise_application_error(-20002, falten || ' objectes necessaris no accessibles');
    END IF;
END;
/
```

La propietat és `all_tables`: mostra el que **tu** pots veure, no el que existix. Per tant la mateixa consulta comprova alhora que l'objecte hi és i que tens el permís, que són les dues coses que et poden faltar.

Això connecta amb el patró de l'alta d'usuari que hem vist abans: allí la petició declarativa descrivia què havia de tindre un usuari; ací descriu què necessita un procés. És la mateixa idea aplicada a l'altre costat.

12.5 Un subconjunt, no una còpia

El reflex natural quan es vol provar alguna cosa contra Oracle és demanar una còpia de preproducció, o directament apuntar a la base de test completa. Per a una exploració manual pot valdre; per a una validació automàtica és una mala elecció, i convé dir per què amb claredat:

> Per a provar una regla no fan falta totes les dades. Fan falta les dades que fan que el cas tinga sentit, i cap més.

Un subconjunt menut i triat guanya en tot el que importa: es carrega en segons, cap en el cap de qui llig la prova, no canvia sota els peus i cada fila hi és per un motiu que es pot explicar. Una còpia completa és lenta, opaca, es queda antiga i —sobretot— **canvia sense avisar**, que és la manera més segura de tindre una prova que hui passa i demà no sense que ningú haja tocat el codi.

L'excepció és real i convé nomenar-la: les **proves de rendiment** sí que necessiten volum, perquè el que mesuren és precisament com es comporta el sistema quan n'hi ha molt. Però són un tipus de prova concret, amb el seu moment i el seu entorn, no la manera per defecte de validar cada canvi.

I hi ha un efecte secundari que sol ser el més valuós: per a construir eixe subconjunt has d'esbrinar **de què depén exactament** el que estàs provant. És l'exercici del qual parlàvem adés, però fet a la força i amb resultat comprovable.

12.6 Esquema buit i càrrega sintètica

No fa falta una còpia de producció. Al contrari: copiar dades reals d'estudiants a un entorn de proves és un risc innecessari i sovint una infracció. El patró raonable és:

1. crear l'esquema buit amb el mateix DDL que va a producció;
2. carregar un joc de dades sintètic, menut i amb noms inventats;
3. incloure-hi expressament els casos límit que volem detectar;
4. destruir-ho tot en acabar.

El DDL de proves (`examples/01_schema_matricula.sql`):

```
WHENEVER SQLERROR EXIT SQL.SQLCODE

-- Els scripts d'inici del contenidor s'executen com a SYSDBA contra el CDB,
-- i l'usuari de l'aplicació viu dins del PDB: cal dir a quin contenidor va.
ALTER SESSION SET CONTAINER = FREEPDB1;

-- Esquema mínim per a proves. És el mateix DDL que aniria a producció:
-- el que canvia entre entorns són les dades, no l'estructura.

CREATE TABLE matricula.estudiant (
    id     NUMBER        PRIMARY KEY,
    nom    VARCHAR2(100) NOT NULL,
    centre VARCHAR2(10)  NOT NULL
);

-- La franja és un instant, no un text: amb DATE, Oracle rebutja "2026-13-01",
-- l'ordenació i les comparacions funcionen soles, i no poden conviure dues
-- escriptures del mateix moment ('2026-09-01 09:00' i '01/09/2026 9:00') que
-- el GROUP BY de la validació comptaria com a franges diferents.
-- DATE i no TIMESTAMP: la precisió de segon ja sobra per a una franja.
CREATE TABLE matricula.centre_franja (
    estudiant_id NUMBER       REFERENCES matricula.estudiant (id),
    centre       VARCHAR2(10) NOT NULL,
    titulacion   VARCHAR2(10) NOT NULL,
    franja       DATE         NOT NULL
);
```

La càrrega sintètica (`examples/02_seed_matricula.sql`):

```
WHENEVER SQLERROR EXIT SQL.SQLCODE

-- Els scripts d'inici del contenidor s'executen com a SYSDBA contra el CDB,
-- i l'usuari de l'aplicació viu dins del PDB: cal dir a quin contenidor va.
ALTER SESSION SET CONTAINER = FREEPDB1;

-- Càrrega de prova sintètica: noms inventats, volum menut i casos límit.
```

```sql
-- Mai s'han de copiar dades reals d'estudiants a un entorn de proves.
-- La base de dades és efímera, per això el joc de dades no esborra res.

INSERT INTO matricula.estudiant (id, nom, centre) VALUES (1, 'Aitana Ferrer Bonet', 'FIS');
INSERT INTO matricula.estudiant (id, nom, centre) VALUES (2, 'Joan Marí Escrivà', 'FIS');
INSERT INTO matricula.estudiant (id, nom, centre) VALUES (3, 'Neus Bellver Tormo', 'ETSE');

-- Cas normal: cada estudiant, una franja distinta.
-- TO_DATE explícit i no la cadena a seques: així la càrrega no depén del
-- NLS_DATE_FORMAT de qui l'execute i dona el mateix resultat.
INSERT INTO matricula.centre_franja VALUES
  (1, 'FIS', 'G1100', TO_DATE('2026-09-01 09:00', 'YYYY-MM-DD HH24:MI'));
INSERT INTO matricula.centre_franja VALUES
  (2, 'FIS', 'G1100', TO_DATE('2026-09-01 09:30', 'YYYY-MM-DD HH24:MI'));
INSERT INTO matricula.centre_franja VALUES
  (3, 'ETSE', 'G1400', TO_DATE('2026-09-01 09:00', 'YYYY-MM-DD HH24:MI'));

COMMIT;
```

Un joc de dades menut és una virtut, no una limitació: cap en el cap de qui llig la prova, es carrega en segons i cada fila hi és per un motiu concret.

12.7 La mateixa base en local i en la CI

En local, un sol fitxer descriu l'entorn (`examples/docker-compose.yml`):

```yaml
# Base de dades de proves efímera: la mateixa que arranca la CI.
#   docker compose up -d     # crea l'esquema i carrega el joc de dades
#   docker compose down -v   # no deixa rastre
#
# La contrasenya és local i d'usar i tirar: esta base no conté dades reals.
services:
  oracle:
    image: gvenzl/oracle-free:23-slim-faststart
    environment:
      ORACLE_PASSWORD: local_throwaway
      APP_USER: matricula
      APP_USER_PASSWORD: local_throwaway
      # Fixat perquè el resultat no depenga de la màquina que ho execute.
      TZ: Europe/Madrid
      ORACLE_CHARACTERSET: AL32UTF8
      NLS_DATE_FORMAT: "YYYY-MM-DD HH24:MI:SS"
    ports:
      - "1521:1521"
    volumes:
      - ./01_schema_matricula.sql:/container-entrypoint-initdb.d/01_schema.sql:ro
      - ./02_seed_matricula.sql:/container-entrypoint-initdb.d/02_seed.sql:ro
```

```
    healthcheck:
      test: ["CMD", "healthcheck.sh"]
      interval: 10s
      retries: 30
```

```
docker compose -f examples/docker-compose.yml up -d
docker compose -f examples/docker-compose.yml down -v
```

El job `validate:oracle` de la pipeline anterior arranca exactament la mateixa imatge, li carrega el mateix esquema i el mateix joc de dades, i executa contra ella la validació de negoci. La diferència entre l'entorn local i el de la CI és el temps d'arrancada, no la configuració.

12.8 El registre l'escriu la màquina

Quan un procés s'executa a mà, el registre del que s'ha fet també s'escriu a mà: es copia la comanda al tiquet, s'apega un tros d'eixida, s'anota l'hora aproximada. Este registre es fa **després** i el fa una persona cansada, així que sol ser incomplet, i de vegades no coincideix amb el que realment es va executar.

Una execució automatitzada produeix el registre com a subproducte, sense esforç i sense possibilitat d'oblit:

- la comanda exacta que es va executar;
- el commit del codi;
- la data, l'hora i la duració reals;
- l'entorn i la imatge utilitzats;
- l'eixida completa, no un fragment seleccionat;
- el resultat, en verd o en roig.

La diferència no és de comoditat, és de naturalesa: un log generat per la màquina és evidència; un registre transcrit a mà és un testimoni. Quan cal explicar mesos després per què un procés va fer una cosa determinada, només el primer serveix.

El wrapper d'exemple ja anota el commit executat precisament per això: enllaça el que va passar en producció amb la línia de codi exacta que ho va fer.

12.9 Execucions deterministes

Una prova només val si, amb les mateixes entrades, dona sempre el mateix resultat. La situació que tots hem patit —*això funcionava i ara no, i ningú ha tocat res*— quasi sempre ve de dependre d'alguna cosa que canvia per baix.

Fonts habituals de no-determinisme, i com fixar-les:

Depén de	Solució
`SYSDATE` o `SYSTIMESTAMP`	Data de referència com a paràmetre
Còpia de producció que canvia cada dia	Joc de dades sintètic i versionat

Depén de	Solució
`NLS_DATE_FORMAT`, `NLS_LANG`, zona horària	Fixats en la definició de l'entorn
Ordre de files sense `ORDER BY`	`ORDER BY` explícit sempre
Seqüències i identificadors generats	No comparar-los literalment
Estat deixat per execucions anteriors	Base efímera, recreada des de zero
Serveis o fitxers externs	Simulats o fixats en el repositori

En la pràctica, la primera fila és la que més sorpreses dona en processos de matrícula i de canvi d'any:

```
-- Depén del dia que s'execute: la prova caduca sola.
WHERE data_inici <= SYSDATE

-- Reproduïble: la data d'execució és una entrada més.
WHERE data_inici <= TO_DATE('&&data_referencia', 'YYYY-MM-DD')
```

Amb la data com a paràmetre, el canvi d'any acadèmic es pot assajar en juliol tantes vegades com faça falta, i el mateix cas es pot tornar a executar l'any que ve amb el mateix resultat.

Per això la definició de l'entorn fixa també la zona horària i els formats: no perquè els valors concrets siguen millors, sinó perquè així no depenen de la màquina que els execute.

12.10 Assajar els processos crítics

Este és el guany real per a processos com la matrícula o el canvi d'any acadèmic. Si el procés es pot executar contra una base efímera amb dades sintètiques, es pot assajar **tantes vegades com faça falta**: cada canvi, cada dia, de matinada, i sempre des de zero.

- cada modificació es valida contra el mateix joc de proves;
- els casos límit es proven una vegada i queden provats per sempre;
- una regressió es detecta el dia que s'introdueix, no el dia de la matrícula;
- l'assaig no depén de reservar un entorn compartit ni de coordinar-se amb ningú.

El dia de la matrícula deixa de ser un dia especial: és una execució més d'un procés que ja s'ha executat centenars de vegades.

13 Integració contínua

13.1 La pipeline més menuda

Abans d'entrar en què es pot comprovar, convé llevar-se de damunt la idea que això és una cosa gran. La pipeline més menuda que serveix per a alguna cosa cap en una pantalla i no necessita ni proves ni entorn:

```yaml
# Comprova que tot el Python del repositori compila: no és
# cap prova, però atrapa l'error d'edició que arriba a
# producció a les 3 de la matinada.
# Es pot posar el primer dia, sense proves i sense entorn.
image: python:3-slim

pipelines:
  default:
    - step:
        name: El Python compila
        max-time: 5
        script:
          # compileall torna un codi distint de zero si un
          # fitxer té un error de sintaxi, i això ja posa la
          # pipeline en roig. No fa falta res més.
          - python -m compileall -q .
```

Comprova que tot el Python del repositori compila. No és cap prova, però atrapa l'error de sintaxi que hauria arribat a producció, i es pot posar el primer dia. És la resposta a «nosaltres encara no tenim proves, així que la CI no ens toca».

La **integració contínua** (CI) vol dir que, cada vegada que algú puja un canvi, s'executa a soles un codi que el comprova. Eixe codi és la **pipeline**, i viu versionada en el repositori com qualsevol altra cosa.

Una CI inicial per a SQL i Oracle pot començar de manera modesta, en dos nivells.

13.2 Comprovacions estàtiques (sense Oracle)

Ràpides —segons— i sense base de dades. Detecten coses com secrets al repositori, marcadors de conflicte sense resoldre, `DELETE`/`UPDATE` sense `WHERE`, `GRANT DBA`, o scripts crítics sense `WHENEVER SQLERROR EXIT`.

Convé no reinventar-les. Hi ha ferramentes fetes i mantingudes que en fan bona part: `gitleaks` o `trufflehog` per a detectar secrets, i linters de SQL com `sqlfluff` o `sqlcheck` per a l'estil i els antipatrons. Un `grep` propi serveix per a entendre el concepte i per a alguna regla molt específica, però per a la resta és millor recolzar-se en una ferramenta existent. A tall d'il · lustració del concepte (`examples/pipeline-comprovacions.yml`):

```yaml
# Exemple de comprovacions estàtiques amb Bitbucket Pipelines.
# És un exemple del curs; la pipeline real d'este repositori és
# bitbucket-pipelines.yml, a l'arrel.
image: alpine:latest

pipelines:
  default:
    - step:
        name: Comprovacions estàtiques
```

```
    # Un pas sense límit s'atura sol als 120 minuts. Amb comprovacions que
    # duren segons, qualsevol cosa que passe d'ací és un error (un bucle,
    # una comanda que espera entrada) i no val la pena pagar-la.
    max-time: 5
    script:
      - |
        # cap_coincidencia PATRÓ [FITXERS...] — falla si troba el patró, i també
        # si grep falla ell mateix. grep torna 0 si troba, 1 si no en troba i 2 o
        # més si té un error (fitxer il·legible, expressió invàlida). L'únic cas
        # bo és l'1: per això no val "grep ... || true", que amagaria un error de
        # grep i passaria en verd sense haver mirat res.
        cap_coincidencia() {
          codi=0
          grep -RInE --exclude-dir=.git "$@" || codi=$?
          case $codi in
            1) return 0 ;;
            0) echo "FALLA: hi ha coincidències de «$1»" ;;
            *) echo "FALLA: grep ha acabat amb codi $codi buscant «$1»" ;;
          esac
          return 1
        }
      # Cap secret al repositori
      - cap_coincidencia "(password|passwd|pwd|secret|token)[[:space:]]*=" .
      # Cap marcador de conflicte sense resoldre
      - cap_coincidencia "^(<<<<<<<|=======|>>>>>>>)" .
      # Cap grant perillós ni DELETE sense WHERE en els SQL d'exemple
      - cap_coincidencia "GRANT[[:space:]]+DBA|WITH[[:space:]]+GRANT[[:space:]]+OPTION|WITH[[
      - cap_coincidencia "DELETE[[:space:]]+FROM[[:space:]]+[A-Za-z0-9_.]+[[:space:]]*;" examp
      # Cap ruta lligada a una màquina concreta (unitat de Windows o recurs de xarxa)
      - cap_coincidencia "[A-Za-z]:\\\\|\\\\\\\\\[A-Za-z]" examples/*.sql
      # Documentació mínima present
      - test -f README.md
      - test -f manual.qmd
```

13.3 Validació sobre un Oracle efímer

El segon nivell connecta a una base de dades **creada i destruïda en cada execució**, carregada amb **dades sintètiques**. Amb això es poden validar regles de negoci que s'han d'acomplir sempre, la compilació de PL/SQL, objectes invàlids, migracions o el recompte esperat de files.

La clau és que l'entorn siga **efímer i determinista**: sempre les mateixes dades, creades des de zero. Una prova sobre una **base compartida** —preproducció, per exemple— perd eixa previsibilitat, perquè les dades canvien sota els peus; per això no és un bon lloc per a una validació automàtica de fiar. Si algun dia es fa un assaig més complet contra un entorn compartit, ha de ser amb dades renovades i entenent que el resultat és orientatiu, no una garantia.

Les proves unitàries de PL/SQL amb `utPLSQL` són el pas natural següent, però no fan falta per a començar ni s'aborden en este curs.

13.4 Validacions barates que ja aporten

Cap d'estes necessita una infraestructura de proves; totes es poden escriure en una vesprada i cadascuna elimina una categoria d'error:

Comprovació	Falla quan
Compilació de tot el PL/SQL	Un canvi d'esquema ha trencat un paquet
`dba_objects` sense objectes invàlids	Alguna cosa ha quedat a mitges
Recompte de files esperat després d'un procés	El procés ha carregat de menys o de més
Constraints i claus foranes actives	S'ha deshabilitat una i ningú l'ha tornada a posar
Cap duplicat en les claus funcionals	La regla de negoci s'ha trencat
Els objectes que el procés necessita existixen	S'ha renombrat una taula
Cap grant més ampli del previst	Algú ha obert un permís de més

El patró és sempre el mateix i és el que fa que la pipeline puga fer-les servir: **el script ha d'acabar amb codi diferent de zero quan la postcondició no es complix**. Un `SELECT` que mostra el problema per pantalla no val: la CI no llig la pantalla, llig el codi d'eixida. Per això els exemples del curs comencen amb `WHENEVER SQLERROR EXIT SQL.SQLCODE`.

13.5 Cada incidència, una prova

Quan apareix una incidència que es pot reproduir, val la pena convertir-la en un cas sintètic **abans** d'arreglar-la: un joc de dades menut que falla amb la versió antiga i passa amb la nova.

El guany no és la prova en si, és on queda el coneixement. Sense això, el que es va aprendre de la incidència viu en un tiquet tancat que ningú tornarà a obrir; amb això, viu en una comprovació que s'executa en cada canvi i que impedix que el problema reaparega en silenci d'ací a dos anys, quan ja ningú recorde que va passar.

13.6 Codi versionat no vol dir codi viu

Un script pot estar perfectament versionat i haver deixat de funcionar fa mesos sense que ningú se n'assabente: un canvi d'esquema, una versió nova del client, un objecte renombrat. Mentre no s'execute, el seu estat és **desconegut**.

Per això val la pena que la pipeline no s'execute només quan algú toca alguna cosa, sinó **també periòdicament** —una vegada a la setmana és prou— contra l'entorn efímer. Una pipeline que es posa roja un dimarts sense que ningú haja canviat res acaba de donar una informació que no tenia preu: alguna cosa de davall s'ha mogut, i ho sabem ara i no el dia de la matrícula.

13.7 Abans de pujar: els hooks de pre-commit

Les mateixes comprovacions es poden executar **en la teua màquina, abans que el commit existisca**. `pre-commit` és la ferramenta estàndard per a això: una configuració en `.pre-commit-config.yaml`, `pre-commit install` una vegada, i a partir d'ahí cada `git commit` passa les regles sobre els fitxers que vas a consolidar.

```
# Les mateixes comprovacions que la pipeline, però abans del commit.
#   pip install pre-commit && pre-commit install
#
# El fitxer real es diu .pre-commit-config.yaml i viu a l'arrel del repositori.
# Ací du un altre nom perquè és un exemple del curs, no la configuració activa.
#
# Un hook no substituïx la pipeline: es pot saltar amb `git commit --no-verify`
# i només el té qui se l'haja instal·lat. El control és la CI; el hook és la
# comoditat d'assabentar-te'n ara i no d'ací a quatre minuts.
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-merge-conflict     # marcadors <<<<<<< sense resoldre
      - id: detect-private-key       # claus privades
      - id: end-of-file-fixer
      - id: trailing-whitespace

  # Les regles pròpies també poden ser hooks. Este és el mateix grep de la
  # pipeline, però només sobre els fitxers que estàs a punt de consolidar.
  - repo: local
    hooks:
      - id: delete-sense-where
        name: DELETE sense WHERE
        language: system
        files: \.sql$
        # Per a un hook, trobar és fallar: per això la lògica va invertida
        # respecte de grep. I un codi 2 o més és un error de grep, que tampoc
        # pot passar en verd.
        entry: |
          sh -c '
            codi=0
            grep -InE "DELETE[[:space:]]+FROM[[:space:]]+[A-Za-z0-9_.]+[[:space:]]*;" "$@" || codi
            [ $codi -eq 1 ] && exit 0
            exit 1
          ' --
```

El guany és de temps: assabentar-te que has deixat un marcador de conflicte costa segons ací i quatre minuts si has d'esperar la pipeline, obrir-la i llegir-la. Sobre un canvi d'una línia, eixa diferència és la que decidix si la comprovació es percep com una ajuda o com una molèstia.

El que no és, és un control. Un hook viu en el `.git/` de cada u, o siga que només el té qui se l'haja instal·lat, i `git commit --no-verify` se'l salta sense demanar permís. No està pensat per a impedir res: està pensat perquè el que va a fallar de totes maneres falle com més prompte millor. La pipeline és la que talla, perquè s'executa en el servidor i no depén de la màquina de ningú.

Per això les dues coses conviuen i no se substituïxen, i per això convé que les regles estiguen definides **una sola vegada** i les use tant el hook com la pipeline. Dues llistes de regles que han de dir el mateix acaben dient coses distintes.

14 Del canvi validat al canvi publicat

Validar un canvi és la meitat de la faena. L'altra meitat és que allò validat arribe a algun lloc, i ací convé distingir tres nivells que sovint es confonen.

Nivell	Què vol dir	Qui decideix quan es publica
Integració contínua	Cada canvi es construeix i es valida automàticament	Ningú: passa sempre
Entrega contínua	El resultat validat queda llest per a desplegar	Una persona, amb un botó
Desplegament continu	El resultat validat es publica tot sol	Ningú: passa sempre

La diferència entre els dos últims no és tècnica, és de decisió. Amb entrega contínua el paquet ja està fet i provat, i algú tria el moment. Amb desplegament continu ningú tria: si la pipeline es posa verda, allò ja està publicat.

14.1 Què fa falta per a arribar al tercer nivell

No es tracta de valentia, sinó de tres condicions:

1. **Que la validació siga de fiar.** Si la pipeline es posa verda amb coses trencades, desplegar automàticament només serveix per a publicar errors més ràpid.
2. **Que el desplegament siga reproduïble.** Sempre les mateixes passes, sense intervenció manual ni "ara toca copiar això a mà".
3. **Que tornar arrere siga barat.** Si desfer costa una vesprada, la publicació automàtica fa por, i amb raó.

Són les mateixes condicions de què hem parlat abans, aplicades al final del procés.

14.2 On sí i on no

Ací convé ser molt clar, perquè el nivell adequat depén del que es publique:

- **Documentació, informes, material i entorns de proves**: desplegament continu sense dubtar. El cost d'un error és baix i es corregeix amb el commit següent.

- **Processos que toquen dades reals de producció** —matrícula, rebuts, actes—: entrega contínua. La pipeline deixa tot preparat i provat, però qui prem el botó és una persona, en una finestra decidida.

Que un canvi es desplegue sol a producció no és l'objectiu de tot equip. L'objectiu és que **quan es decidisca desplegar, no faça falta cap passa manual ni cap coneixement que no estiga escrit**. Si això es compleix, automatitzar l'últim pas ja només és una decisió de risc, no un problema tècnic.

14.3 Les branques no són els entorns

Una temptació habitual quan es munta això per primera vegada és fer una branca per entorn: `develop` per a desenvolupament, `test` per a proves, `main` per a producció. Sembla ordenat i acaba malament, perquè el que es promou d'un entorn a un altre deixa de ser una cosa concreta:

- les tres branques divergixen a poc a poc i cap representa un estat conegut del conjunt;
- un pegat aplicat en producció no està en desenvolupament, i ningú sap quan tornarà a desaparéixer;
- el que es prova en TEST no és, exactament, el que arribarà a PROD.

La idea preferible és que **un mateix commit validat avance** DES → TEST → PROD, i que el que canvie entre entorns siga una altra cosa:

Canvia entre entorns	No canvia entre entorns
Configuració: àlies, rutes, límits, finestres	El codi
Secrets: credencials, wallets	El DDL i les migracions
La decisió de desplegar i qui la pren	Les validacions que ha de superar

Així «què hi ha en producció?» té resposta —un commit— i la promoció és una decisió, no una còpia.

14.4 La publicació, un pas més

Publicar el resultat és, tècnicament, **un pas final més de la pipeline**, i es pot fer de diverses maneres: deixar-lo com a **artefacte** que algú descarrega, pujar-lo a un **registre** o a un magatzem d'objectes, o **copiar-lo** a un servidor. Cap no és l'única correcta; depén d'on ha d'arribar el resultat.

Siga com siga, val un principi senzill: la pipeline ha d'usar una **credencial dedicada i limitada**, mai el compte d'una persona. Amb el permís mínim per a fer només allò que necessita, un error a la pipeline queda acotat.

Este curs, de fet, es construeix i es publica exactament així; l'apartat «Com està muntat este document» ho descriu peça per peça.

15 Com està muntat este document

Este material no és un exemple inventat: es construeix exactament amb el que s'explica ací. Esta secció descriu el muntatge, peça per peça, perquè es puga copiar.

Construcció automàtica activa

Cada canvi que arriba al repositori es construeix des de zero i, si la construcció va bé, el web queda empaquetat en un zip que qualsevol es pot baixar.

```
push a main
     ↓
Bitbucket Pipelines
     ↓
[1] comprovacions estàtiques                    ← segons; si falla, s'acaba ací
     ↓
[2] construir en un contenidor Debian net       ← si falla, s'acaba ací
     ↓
[3] empaquetar el web en un zip                 ← artefacte de l'execució
     ↓
qualsevol se'l baixa de la pipeline
```

15.1 El repositori

El codi viu en Bitbucket Cloud, en el workspace de la Universitat. Tot el que fa falta per a construir el material està dins: els documents, els exemples, els estils, el logotip, el script d'entorn i la definició de la pipeline mateixa. No hi ha res guardat "a banda".

15.2 Les comprovacions

La primera passa són les comprovacions estàtiques, les mateixes que es veuen en l'apartat d'integració contínua: secrets, marcadors de conflicte sense resoldre, `GRANT DBA`, `DELETE` sense `WHERE` i rutes lligades a una màquina. Van en un pas propi i amb una imatge `alpine` per dos motius: costen segons, i quan la pipeline es pose roja es veurà **quina etapa** ha fallat en compte d'un genèric «no s'ha construït».

L'ordre importa: primer el que costa segons, després el que costa minuts. Així una errada típica es veu de seguida i no després d'instal · lar mig sistema.

15.3 La construcció

`bitbucket-pipelines.yml` és la definició completa:

```yaml
# Este repositori és, també, l'exemple de CI del curs: el material es
# construeix en cada canvi amb les mateixes comandes que en local
# (scripts/setup.sh i make). Si no es construeix, el canvi no és vàlid.
image: debian:stable-slim

definitions:
  caches:
    # Quarto s'instal·la en ~/.local; cachejar-ho estalvia la descàrrega.
    quarto: /root/.local
  steps:
    - step: &comprovar
        name: Comprovacions estàtiques
        # La mateixa imatge que la construcció, i no una d'alpine: el grep de
        # BusyBox no té `--exclude-dir` i el pas es va posar roig amb un codi 2.
        # Ací el grep és el mateix que en local, que és justament el que este
        # curs predica. Va en un pas propi perquè, quan es pose roja, es veja
        # quina etapa ha fallat i no «no s'ha construït».
        image: debian:stable-slim
        max-time: 5
        script:
          - |
            # cap_coincidencia PATRÓ [FITXERS...] — falla si troba el patró, i
            # també si grep falla ell mateix. grep torna 0 si troba, 1 si no en
            # troba i 2 o més si té un error. L'únic cas bo és l'1: per això no
            # val "grep ... || true", que amagaria un error de grep i passaria
            # en verd sense haver mirat res.
            cap_coincidencia() {
              codi=0
              grep -RInE --exclude-dir=.git "$@" || codi=$?
              case $codi in
                1) return 0 ;;
                0) echo "FALLA: hi ha coincidències de «$1»" ;;
                *) echo "FALLA: grep ha acabat amb codi $codi buscant «$1»" ;;
              esac
              return 1
            }
          # Cap secret enlloc del repositori
          - cap_coincidencia "(password|passwd|pwd|secret|token)[[:space:]]*=" .
          # La resta només en examples/: el material del curs conté marcadors de
          # conflicte i SQL perillós a posta, per a ensenyar-los.
          - cap_coincidencia "^(<<<<<<<|=======|>>>>>>>)" examples/
          - cap_coincidencia "GRANT[[:space:]]+DBA|WITH[[:space:]]+GRANT[[:space:]]+OPTION|WITH[[
          # Un DELETE que acaba en ";" just després del nom de la taula, és a
          # dir, sense WHERE. És la comprovació de la cançó, no una de "DELETE
          # sense FROM".
          - cap_coincidencia "DELETE[[:space:]]+FROM[[:space:]]+[A-Za-z0-9_.]+[[:space:]]*;" examp
          - cap_coincidencia "[A-Za-z]:\\\\|\\\\\\\\[A-Za-z]" examples/
          # Documentació mínima present
```

```
          - test -f README.md
          - test -f manual.qmd

    - step: &construir
        name: Construir el material
        caches:
          - quarto
        script:
          - apt-get update -qq
          - apt-get install -y --no-install-recommends ca-certificates curl git make tar zip fonto
          - export PATH="$HOME/.local/bin:$PATH"
          - ./scripts/setup.sh
          - make zip
        artifacts:
          - curs-git-ci-oracle.zip

# Publicació desactivada. Es va usar per a copiar el web a un servidor i es
# deixa ací, comentada, perquè torne a funcionar només llevant els comentaris i
# definint DEPLOY_USER i DEPLOY_HOST com a variables del repositori. La clau SSH
# la genera el mateix Bitbucket (Repository settings -> Pipelines -> SSH keys) i
# la part privada no ix mai d'ahí.
#
#    - step: &publicar
#        name: Publicar el material
#        deployment: production
#        script:
#          - apt-get update -qq
#          - apt-get install -y --no-install-recommends openssh-client rsync
#          - rsync -a --omit-dir-times --no-perms
#                  -e "ssh -o StrictHostKeyChecking=accept-new"
#                  _site/ "$DEPLOY_USER@$DEPLOY_HOST:/"

pipelines:
  default:
    - step: *comprovar
    - step: *construir
#  branches:
#    main:
#      - step: *comprovar
#      - step: *construir
#      - step: *publicar

# No hi ha pas de publicació: el material construït queda com a artefacte de la
# pipeline (pestanya «Artifacts» de l'execució), i qui el necessite se'l baixa.
# Així la pipeline no necessita cap credencial ni cap servidor.
```

Tres detalls que expliquen per què això funciona:

- **La imatge és `debian:stable-slim`**, buida. Cada construcció parteix d'un sistema sense res instal · lat, així que és impossible que el material depenga d'alguna cosa que només estava en un ordinador.
- **Les comandes són les mateixes que en local**: `./scripts/setup.sh` i `make`. No hi ha una recepta per a la CI i una altra per a les persones; si la CI construeix, tu construeixes igual.
- **`~/.local` es cacheja**, que és on `setup.sh` instal · la Quarto. Estalvia la descàrrega en cada execució sense canviar res del resultat.

Si esta passa acaba en roig, la pipeline s'atura ací i no hi ha res per a baixar: mai s'empaqueta una versió que no s'ha pogut construir.

15.4 L'entrega

Quan la construcció va bé, l'últim que fa el pas és `make zip`: el web sencer en un fitxer, declarat com a **artefacte** de l'execució. Bitbucket el guarda i el deixa descarregar des de la pestanya «Artifacts» de la pipeline.

És la forma d'entrega més barata que hi ha, i val la pena veure per què: **la pipeline no necessita cap credencial ni cap servidor**. No hi ha clau SSH, ni compte de desplegament, ni màquina que mantindre; si demà el resultat ha d'anar a un web, l'únic que canvia és eixa última línia.

Quan sí que hi haja un servidor al final, el principi és este, i generalitza molt més enllà d'este curs:

> Git, la revisió i la CI reduïxen la probabilitat d'un error. Els permisos mínims en limiten les conseqüències. Fan faenes diferents i no se substituïxen.

La credencial d'una pipeline ha de ser **dedicada i limitada** —mai el compte d'una persona—, i acotada pel servidor, no per la bona conducta de l'script.

15.5 On corren les pipelines

Una pipeline s'ha d'executar en algun lloc, i hi ha dues opcions:

- **En els agents de Bitbucket** (el que fem ací): còmode, sense mantindre res, però amb un **límit de minuts** al mes segons el pla contractat.
- **En agents propis** (*self-hosted runners*): màquines nostres on instal · lem l'agent. No tenen límit de temps ni de CPU i, sobretot, poden **accedir a recursos de la xarxa interna** —bases de dades, servidors o VLANs específiques que no són visibles des d'internet—. Un mateix agent pot donar servei a tots els repositoris del workspace.

Per a un servei que treballa contra Oracle intern, els agents propis solen ser l'opció natural quan la pipeline ha de connectar a recursos que no ixen a fora.

15.6 Què passa en cada canvi

1. Algú puja un canvi al repositori.
2. Es construeix el material des de zero en un contenidor net.
3. Si la construcció falla, la pipeline es posa roja i no hi ha artefacte.
4. Si va bé, el zip del web queda penjat de l'execució. Ningú construeix res a mà, i el que es baixa correspon exactament a un commit.

16 Documentació que es pot executar

16.1 La prosa accepta qualsevol cosa

Un document en disc o en una wiki té una llibertat total: pots escriure el que vulgues i ningú et dirà res, perquè no hi ha cap màquina que ho comprove. Això vol dir que un procediment pot contindre, sense que ningú se n'assabente:

- un `SELECT` sobre una taula que es va renombrar fa dos anys;
- un `GRANT` sobre un rol que ja no existeix;
- una passa que es refereix a un servidor apagat;
- un fragment de SQL que no s'ha executat mai, ni una sola vegada;
- una captura de pantalla d'una versió anterior de l'aplicació.

El document no dona cap error. Simplement deixa de ser cert, en silenci, i normalment ho descobrim el dia que algú l'ha de seguir en una situació d'urgència.

16.2 Les restriccions donen qualitat

Passar d'un document lliure a codi versionat i validat es viu al principi com una pèrdua de llibertat, i és normal que moleste: obliga a col · locar les coses on toca, a posar-los nom, a documentar-les i a fer-les executables.

Però cada restricció que acceptem és una categoria d'error que ja no pot arribar a producció:

Restricció	Error que elimina
El SQL ha d'executar-se sense error en proves	Codi que no s'ha provat mai
El fitxer ha d'estar en la ruta acordada	"Estava en la meua carpeta"
Els scripts crítics han de dur `WHENEVER SQLERROR`	Continuar després d'una fallada
No pot haver-hi secrets en el repositori	Credencials filtrades
Cada procés ha de tindre `README.md`	Coneixement només oral
Cada canvi ha de passar per pull request	Canvis sense revisar

És el mateix paper que fan un compilador o un analitzador estàtic: limiten el que pots escriure i, precisament per això, detecten errors que una persona llegint no veuria.

16.3 Codi sense prova és codi que no tens

> El codi que no té una prova associada és com si no el tingueres: ningú et diu que funcionarà el dia que el necessites.

No vol dir que siga mal codi. Vol dir que el seu estat és **desconegut**: pot funcionar, pot haver deixat de funcionar fa mesos per un canvi d'esquema, o pot no haver funcionat mai. I el dia que el necessitem sol ser el pitjor dia possible per a descobrir-ho: la matrícula, el tancament de l'any o una incidència de matinada.

Una prova, encara que siga molt simple, converteix esta incògnita en una resposta que es renova sola cada vegada que la pipeline s'executa.

16.4 De document a codi executable

La conversió no cal fer-la de colp. Per a cada procediment, la pregunta és quina part pot passar de prosa a codi:

Ara està com a	Pot passar a ser
Passa descrita en un document	Script versionat que l'executa
Consulta apegada en la wiki	Fitxer `.sql` que la CI executa
"Cal comprovar que no hi ha duplicats"	`validacio_cites.sql`, que falla si n'hi ha
Captura de pantalla del resultat	Eixida real registrada en el log
Llista de requisits previs	Definició de l'entorn en el repositori

El text no desapareix: continua explicant el perquè, les decisions i el context, que és el que el codi no diu. El que desapareix és el text que **pretén** ser executable sense ser-ho.

16.5 Este mateix material

Este curs aplica la mateixa regla. Els blocs de codi de les diapositives i del manual no estan copiats: s'inclouen automàticament des dels fitxers reals d'`examples/`, i estos fitxers passen per la pipeline com qualsevol altre codi del repositori.

Si un exemple deixa de funcionar, es nota. Si algú el millora, el material canvia sol. Una transparència amb SQL copiat a mà seria, exactament, el problema que estem intentant resoldre.

17 Coneixement tàcit i continuïtat del servei

17.1 Què és el coneixement tàcit

És el coneixement que només existeix en el cap d'algunes persones. No apareix en cap document, i sovint ni tan sols es percep com a coneixement: qui el té creu que és obvi.

En un servei d'informàtica té formes molt reconeixibles:

- l'ordre real en què s'han d'executar les passes, que no és el del document;
- el paràmetre que cal canviar cada any i que ningú va escriure enlloc;
- l'excepció d'un centre concret que es tracta de manera diferent;
- el fitxer que cal esborrar abans de tornar a llançar el procés;
- saber quin dels tres scripts amb nom semblant és el bo;
- saber què vol dir realment un error que ix cada any i s'ignora.

Documentar-lo no és guardar-lo en un document. Un fitxer que ningú executa envelleix en silenci: torna a ser coneixement tàcit el dia que deixa de ser cert i ningú se n'assabenta.

17.2 Continuïtat i dependència personal

La pregunta és senzilla: **quantes persones haurien de faltar perquè un procés quedara bloquejat?** Si la resposta és una, convé repartir eixe coneixement, per bé que funcione tot ara mateix.

El fenomen (de vegades anomenat «factor autobús») és quotidià i rarament té res a veure amb un accident: una excedència, un canvi de servei, una jubilació, una **baixa per malaltia —fins i tot un simple refredat—**, una baixa llarga o una oferta millor.

El més important és que esta concentració **es produeix sense que ningú la decidesca**. La persona que millor coneix un procés acaba fent-lo sempre, perquè és la que el fa més ràpid i amb menys risc; i cada vegada que el fa, la distància amb la resta creix. No és culpa de ningú: és una deriva natural que, si no es compensa activament, sempre va en la mateixa direcció.

I perjudica les dues parts:

- la institució depén d'una persona per a un servei que no pot fallar;
- la persona queda atrapada en un procés que ja no pot delegar, no pot desconnectar en els períodes crítics i li costa canviar de tasca o créixer cap a una altra cosa.

Repartir el coneixement no és llevar-li valor a ningú: és la manera que té d'agafar vacances en agost sense el telèfon damunt de la taula.

17.3 La pipeline contesta les preguntes

Hi ha un efecte secundari de la integració contínua que costa d'apreciar fins que es viu: **en un equip amb la CI ben muntada, la gent nova deixa de preguntar coses**.

Ningú ha d'explicar com es compila, com es desplega, quina versió es fa servir o en quin ordre s'executen les passes, perquè tot això està en fitxers que qualsevol pot llegir. I hi està d'una manera molt particular: **si mentiren, no funcionarien**. Un document pot quedar-se antic sense que passe res; una pipeline que no diu la veritat falla.

Els beneficis van en les dues direccions:

- qui arriba és autònom des del primer dia i pot aprendre llegint;
- qui coneix el sistema deixa de ser interromput per a explicar sempre el mateix;
- la resposta és idèntica per a tothom i està actualitzada per construcció.

A més, un llenguatge executable és formal i inequívoc: diu exactament una cosa. La prosa admet interpretacions —i quan hi ha pressa, cadascú tria la seua.

17.4 No és un problema teòric

És fàcil pensar que no cal preocupar-se perquè cada any hi ha la mateixa gent. En la pràctica, això no es compleix: hi ha baixes, canvis de servei, jubilacions, excedències i situacions que no es poden preveure. Es poden prohibir les vacances en període de matrícula, però no es pot prohibir una grip, i una pandèmia pot deixar fora de servei mitja plantilla alhora.

I la matrícula no es pot ajornar. La pregunta útil no és si això passarà, sinó què passa el dia que passe:

> Si les persones que coneixen este procés no estigueren disponibles, es podria executar igualment?

El valor de fer-se esta pregunta és que **té resposta comprovable**. No cal opinar: es pot provar.

17.5 La CI com a auditora del coneixement tàcit

Esta és la relació directa entre integració contínua i coneixement tàcit: **una màquina no pot preguntar-li res a ningú**.

La pipeline no sap què va fer l'any passat, no recorda l'excepció d'aquell centre i no pot trucar per telèfon a les tres de la matinada. Si el procés funciona en la CI, és perquè tot el que fa falta està escrit en algun lloc del repositori. Si no funciona, la pipeline acaba d'assenyalar exactament quin tros de coneixement continuava sent tàcit.

Cada fallada d'este tipus és una bona notícia barata: fa explícit al juliol el que altrament s'hauria descobert al setembre, ja en plena activitat.

17.6 La prova de continuïtat

Convertit en criteri operatiu, queda així:

> Un procés està sota control quan la seua execució no depén de cap persona concreta, i això s'ha comprovat executant-lo.

Comprovar-ho és senzill amb els entorns efímers:

1. partir d'un entorn net, creat des de zero;
2. seguir només el que hi ha en el repositori;
3. no permetre cap intervenció manual no escrita;
4. repetir-ho tantes vegades com faça falta.

Si la pipeline el munta i l'executa sense ajuda, la resposta és que sí. Si en algun punt fa falta que algú "prepare" o "toque" alguna cosa, la resposta és que no, i acabem de descobrir exactament què cal escriure.

L'objectiu no és que la gent deixe de ser valuosa: és que ningú haja de ser imprescindible un diumenge d'agost.

18 Configuració i por al canvi

18.1 La configuració també és codi

Bona part de la faena d'un servei d'informàtica no és escriure programes, sinó configurar-los: paràmetres de serveis, definicions de desplegament, fitxers d'arrancada, jobs programats, rutes, límits, temps d'espera, inventaris de connexions.

És, precisament, el que més canvia i el que sol estar pitjor versionat, perquè quasi sempre s'edita des de la mateixa ferramenta que el consumeix. Ningú obri un repositori Git per a canviar un paràmetre en un editor de configuració: es canvia i s'aplica.

El que sí que s'ha vist en tots els serveis del món és l'altra meitat del problema resolta a mà:

```
config.xml
config.xml.bak
config.xml.old
config.xml.bak2
config.xml.20240912
config.xml.ANTES_DE_TOCAR
```

Això és control de versions, però fet a mà i sense les garanties: no diu qui va fer el canvi, ni quan, ni per què, ni què va canviar exactament, ni assegura que la còpia siga la que realment estava en producció. El que aporta Git ací no és sofisticació: és el mínim.

L'inventari de connexions (`examples/inventory.yaml`) és un exemple menut d'això mateix: dades de configuració no secretes, versionades, revisables i consultables sense entrar en cap servidor.

18.2 Una regla per a tocar configuració

El patró habitual quan es versiona configuració és fer el commit **després** del canvi. És millor que res, però es perd la meitat del valor. La seqüència completa són cinc passes:

```
commit abans  →  canvi  →  diff  →  validació  →  commit després
```

Cada peça fa una faena distinta:

- el **commit abans** fixa un estat conegut al qual tornar, i que es va crear quan encara tot funcionava;
- el **diff** contesta «què he tocat exactament?» mentre encara ho recordes;
- la **validació** és el que separa un canvi provat d'un canvi aplicat;

- el **commit després** representa una configuració **ja validada**, no una que simplement s'ha escrit.

En `/etc`, `etckeeper` fa la primera i l'última passa a soles: manté el directori com un repositori Git i consolida abans i després de cada instal · lació de paquets. La història és curta i tothom l'ha viscuda:

> Toque `sshd_config`. Alguna cosa deixa de funcionar. `git diff` em diu exactament què vaig canviar, i `git restore` ho desfà.

18.2.1 Un canvi real, de cap a peus

`haproxy` es disputava el port 80 amb `apache2`. Primer, aïllar el que ja estava sense consolidar, i **en una comanda a banda**:

```
sudo etckeeper unclean && sudo etckeeper commit "desviaments previs a desactivar haproxy"
```

Eixe commit no parla del canvi que ve després: arreplega el que algú va deixar a mitges abans, perquè el nostre canvi quede sol i llegible en l'historial.

Va a banda per una raó concreta: **`etckeeper commit` acaba amb codi d'eixida 1 quan no hi ha res a consolidar**. És el cas normal —un `/etc` net és una bona notícia—, però encadenat amb `&&` a la resta avortaria tota la seqüència justament quan tot està bé. Per això la guarda `etckeeper unclean`, que és cert només si hi ha canvis pendents.

Ara sí, el canvi, la validació i el commit posterior, encadenats:

```
sudo systemctl disable --now haproxy &&
! systemctl is-active --quiet haproxy &&
sudo etckeeper commit "desactivat haproxy: es disputava el port 80 amb apache2 i apuntava a 4010"
```

Tres detalls que decidixen si això val o no:

- **`--now`**. `systemctl disable` només lleva els enllaços d'arrancada: el procés continua viu i continua ocupant el port fins al pròxim reinici. Si el motiu del canvi és alliberar el port 80, sense `--now` el problema continua ahí i el commit diu que està resolt.
- **La comprovació d'enmig** és la validació de les cinc passes: el commit final només s'escriu si el servei està parat de veres. Un commit que representa una configuració **validada** val molt més que un que representa una comanda executada.
- **El missatge diu per què**, no què. El `diff` ja diu què ha canviat; el que no es pot reconstruir mai és el motiu, i «apuntava a 4010» és exactament el que algú necessitarà d'ací a un any.

Si el commit final també diu *nothing to commit*, no és una molèstia: és que l'operació no ha tocat res de `/etc`. Val més saber-ho que suposar-ho.

I si cal garantir que ningú el torne a alçar —una dependència d'una altra unitat, una activació per socket—, `disable` no basta: la versió dura és `mask`.

Fora de `/etc` el criteri és el mateix: mirar si el directori ja està dins d'un repositori, i si no, triar una arrel raonable, excloure secrets, logs i generats, i inicialitzar-lo. No és una ferramenta nova, és la mateixa disciplina aplicada al lloc on de veritat es toquen les coses.

18.3 Quan la realitat s'allunya del repositori

El repositori pot descriure l'estat desitjat, però res impedix que algú modifique producció directament: SQL Developer, OEM, `srvctl`, APEX, un `ALTER` de matinada. A partir d'eixe moment el que està escrit i el que està en marxa són dues coses diferents, i ningú ho sap. Este fenomen té nom — **divergència de configuració**, o *configuration drift*— i la propietat que el fa perillós és que creix en silenci.

La resposta no és prohibir tocar res, que no funciona, sinó **mesurar la divergència**: exportar periòdicament l'estat real i comparar-lo amb la línia base versionada.

- definicions de jobs de `DBMS_SCHEDULER`;
- grants, rols i perfils;
- paràmetres d'instància rellevants;
- codi PL/SQL desplegat;
- configuració de serveis i de connexions.

La comparació pot ser una passa més de la pipeline, i el resultat interessant no és l'informe sencer sinó el **canvi**: què ha aparegut o desaparegut des de l'última vegada. És exactament el mateix criteri que aplica la ronda nocturna de `examples/ronda-escrivibles.sh`.

18.4 El pegat urgent en producció

Convé dir-ho sense moralina: davant d'una urgència real pot estar justificat tocar alguna cosa directament en producció. Un model que ho prohibisca en absolut no el complirà ningú, i el que passarà és que es farà igual i no es contarà.

El requisit no és no fer-ho mai, sinó que després **l'estat real es reconcilie amb el repositori**:

1. el canvi s'incorpora a Git, amb el seu motiu i el tiquet de la incidència;
2. es revisa, encara que siga a posteriori;
3. es comprova que el pròxim desplegament no el faça desaparéixer.

El tercer punt és el que sol fallar, i és el que converteix un pegat que va salvar la nit en la mateixa incidència repetida tres mesos després.

18.5 Si tornar arrere fa por, no s'avança

> Si tornar arrere no és fàcil, ràpid i segur, la gent té por d'anar cap avant.

Esta és, probablement, l'explicació de la major part de la paràlisi tècnica que patim. No és falta de ganes ni de criteri: és una resposta perfectament racional a un risc que no es pot desfer.

Les conseqüències es reconeixen a simple vista:

- sistemes que fa anys que no s'actualitzen;
- canvis acumulats fins a una finestra enorme, que per això mateix fa més por;
- la frase *això millor no ho toques, que funciona*;
- versions que ja no reben pedaços de seguretat.

I hi ha un bucle: com més es tarda, més gran és el salt; com més gran és el salt, més por fa; com més por fa, més es tarda. La deuda no es queda quieta.

A més del risc operatiu, hi ha un component normatiu: l'Esquema Nacional de Seguretat exigeix gestionar la configuració, mantindre els sistemes actualitzats i controlar els canvis. Un sistema que no es pot actualitzar sense por difícilment compleix cap de les tres coses.

18.6 Actualitzar a poc a poc

Amb l'entorn definit en el repositori i una pipeline que el valida, provar una actualització deixa de ser una operació i passa a ser un commit:

```
-    image: gvenzl/oracle-free:23-slim-faststart
+    image: gvenzl/oracle-free:26-slim-faststart
```

Branca, canvi d'una línia, pipeline. Si es posa verda, el salt està provat amb dades i validacions reals. Si falla, s'esborra la branca i no ha passat absolutament res: ningú ha tocat cap servidor.

Quan intentar-ho costa tan poc, s'intenta sovint, i això canvia el perfil de risc del servei:

- salts menuts i freqüents en lloc de salts grans i rars;
- l'evidència que funciona la dona la pipeline, no una opinió;
- tornar arrere és revertir un commit, no restaurar una còpia i creuar els dits;
- l'actualització es prova en juliol, no la nit abans del venciment.

18.7 Fins on arriba «revertir un commit»

Convé acotar la frase anterior, perquè el matís és important i és una font habitual de confiança mal posada.

Revertir un commit recupera la **font**: el fitxer de configuració, el script, la definició de l'entorn. Això cobre tot el que hem vist en este apartat, on el canvi encara no s'ha aplicat enlloc.

El que no fa és desfer els **efectes** d'una execució ja llançada. Si un canvi ja s'ha aplicat contra la base de dades, revertir el commit torna el codi a com estava, però les dades queden com havien quedat i el DDL aplicat no es desfà (en Oracle, a més, el DDL du un commit implícit que un `ROLLBACK` posterior no desfà).

Recuperar l'estat anterior de les dades és un problema diferent i té les seues ferramentes: una migració compensatòria escrita a propòsit, Oracle Flashback, o la restauració d'una còpia.

> Git et torna la definició anterior. Tornar l'estat anterior és un pla que has d'haver escrit abans d'executar.

Per això, per als canvis que toquen dades reals, el curs insistix en l'entrega contínua amb botó humà i no en el desplegament automàtic: no perquè la pipeline siga menys de fiar, sinó perquè el cost d'equivocar-se no és simètric.

19 Per on podem començar

La manera més senzilla de començar no és migrar un procés que ja funciona, sinó **crear un repositori nou** per als scripts i la configuració Oracle que es facen d'ara en avant. En el servei, molta activitat es fa directament contra la base de dades i no sempre existeix un conjunt de fitxers que es puga importar a Git. Això no impedeix començar; al contrari, permet una regla senzilla per al treball futur:

> Tot script operatiu nou naix versionat, revisable i amb les comprovacions necessàries.

No cal una migració massiva del patrimoni anterior: els scripts i processos antics s'incorporaran gradualment quan calga modificar-los, reutilitzar-los o documentar-los.

19.1 Un repositori nou d'operació Oracle

El primer projecte pilot pot reunir scripts nous de risc baix:

- comprovacions de tablespaces, sessions o objectes invàlids;
- scripts d'usuaris, rols, perfils i grants;
- definicions de `DBMS_SCHEDULER`;
- canvis de paràmetres;
- wrappers d'execució;
- plantilles de configuració;
- validacions prèvies i posteriors.

L'objectiu inicial no és automatitzar immediatament la producció, sinó aconseguir que cada operació repetible tinga:

1. un fitxer identificable;
2. un motiu o tiquet;
3. una revisió;
4. una comprovació prèvia;
5. una comprovació posterior;
6. una manera documentada de tornar arrere quan siga possible.

19.2 Dos tipus de contingut

Convé no barrejar-ho tot en un sol lloc:

- **Repositori compartit d'operació Oracle**: scripts reutilitzables d'administració, comprovació, configuració i manteniment.
- **Repositori de cada projecte o procés**: el DDL, el PL/SQL, les migracions, les dades mestres i les validacions específiques d'una aplicació o domini funcional.

Convé evitar que el repositori compartit es convertisca en un calaix amb tot el SQL de tots els projectes.

19.3 Què es versiona i què no

No cal versionar qualsevol consulta exploratòria. Un script hauria d'entrar al repositori quan:

- modifica dades, estructura, seguretat o configuració;
- s'executarà repetidament;
- forma part d'un procediment;
- necessita revisió;
- té impacte operatiu;
- o conserva coneixement que altres persones poden necessitar.

19.4 Configuració Oracle

La configuració no resideix tota en fitxers: pot estar en l'`SPFILE`, el diccionari de dades, Grid Infrastructure, `srvctl`, `DBMS_SCHEDULER` o fitxers del sistema. El repositori pot contindre els scripts que declaren el canvi, l'estat desitjat, plantilles, línies base exportades, comprovacions de divergència i documentació sobre la configuració gestionada externament.

No hi han d'anar mai secrets, wallets, claus, tokens, volcats de producció ni dades personals.

19.5 Adopció gradual

1. els scripts nous naixen en Git;
2. els canvis delicats es revisen abans d'executar-se;
3. els processos repetits incorporen comprovacions;
4. els objectes antics s'exporten quan cal modificar-los;
5. alguns processos passen gradualment de l'execució manual a l'entrega controlada.

Git no transforma immediatament tot el treball Oracle: és una disciplina que evita continuar acumulant operacions sense una font versionada.

20 Exercicis

Vegeu `exercicis.qmd`.