Actividad 6:

creamos la carpeta para el proyecto

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/D
n)
$ mkdir scrum-project

windows10@DESKTOP-UEI1KU9 MINGW64 ~/D
n)
$ cd scrum-project
```

Luego iniciamos el repositorio

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (main)
$ git init
Initialized empty Git repository in C:/Users/windows10/Desktop/DesarrolloDeSoftw
are/actividad6/scrum-project/.git/
```

Creamos nuestro archivo readme.md

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (main)
$ echo "# Proyecto Scrum" > README.md

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (main)
$ git add README.md
warning: in the working copy of 'README.md', LF will be replaced by CRLF the nex
t time Git touches it

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (main)
$ git commit -m "Commit inicial en main"
[main (root-commit) 0878607] Commit inicial en main
 1 file changed, 1 insertion(+)
 create mode 100644 README.md

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (main)
```

Creamos las 2 ramas

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSo
m-project (main)
$ git checkout -b feature-user-story-1
Switched to a new branch 'feature-user-story-1'

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSo
m-project (feature-user-story-1)
$ git checkout -b feature-user-story-2
Switched to a new branch 'feature-user-story-2'

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSo
m-project (feature-user-story-2)
```

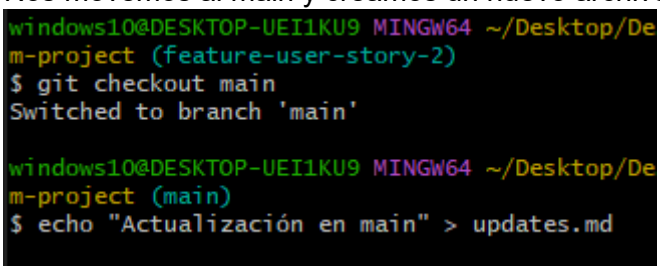
PREGUNTA:

¿Por qué es importante trabajar en ramas de funcionalidades separadas durante un sprint?

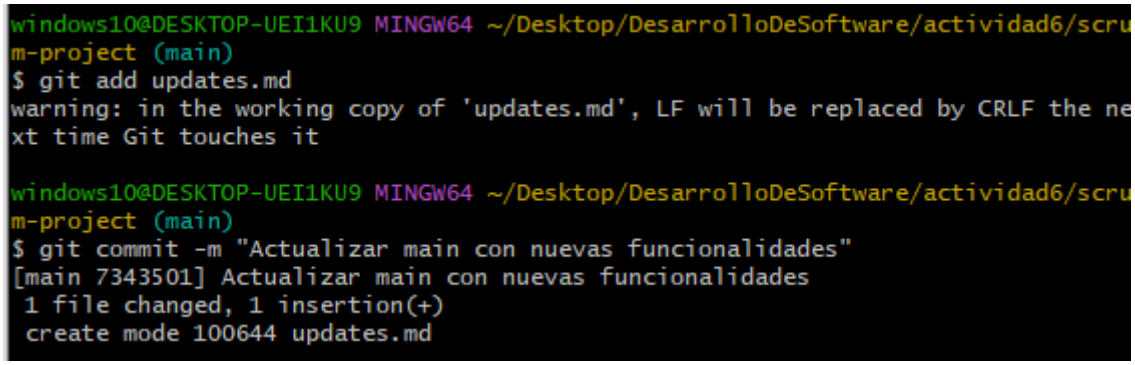
Trabajar en ramas de funcionalidades separadas permite que cada desarrollador trabaje de forma independiente sin afectar el código de otros. Facilita el control de versiones y pruebas aisladas. Además, ayuda a detectar errores más fácilmente al integrar cambios por separado.

Parte 2:

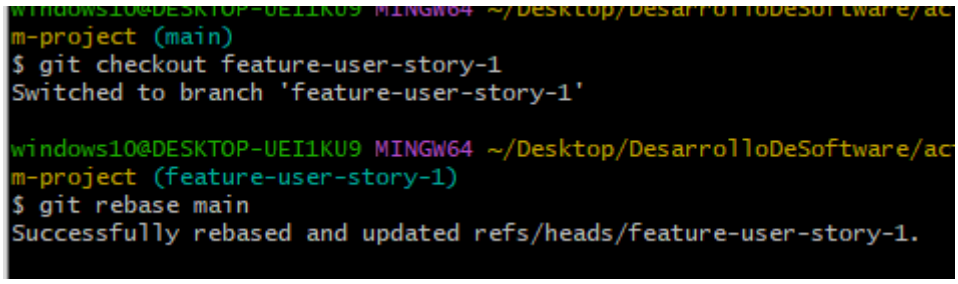
Nos movemos al main y creamos un nuevo archivo



Agregamos al repositorio



Nos movemos a la rama feature1 y hacemos el rebase de la rama feature1 sobre main



PREGUNTA:

¿Qué ventajas proporciona el rebase durante el desarrollo de un sprint en términos de integración continua?

El rebase mantiene un historial limpio y lineal, facilitando la revisión del código. Permite integrar los últimos cambios de main continuamente, reduciendo conflictos al final del sprint. Además, ayuda a detectar errores temprano y mejora la colaboración entre desarrolladores

FASE 3

Nos movemos a la rama feature-user-story-2

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDe
m-project (feature-user-story-1)
$ git checkout feature-user-story-2
Switched to branch 'feature-user-story-2'
```

Creamos el archivo funcionalidad lista y lo preparamos y realizamos el commit

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scr
m-project (feature-user-story-2)
$ echo "Funcionalidad lista" > feature2.md

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scr
m-project (feature-user-story-2)
$ git add feature2.md
warning: in the working copy of 'feature2.md', LF will be replaced by CRLF the
ext time Git touches it
```

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/s
m-project (feature-user-story-2)
$ git commit -m "Funcionalidad lista para revisión"
[feature-user-story-2 2151ea2] Funcionalidad lista para revisión
 1 file changed, 1 insertion(+)
 create mode 100644 feature2.md
```

Ahora creamos el archivo funcionalidad en progreso, lo preparamos y realizamos el commit

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloD
m-project (feature-user-story-2)
$ echo "Funcionalidad en progreso" > progress.md
```

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scr
m-project (feature-user-story-2)
$ git add progress.md
warning: in the working copy of 'progress.md', LF will be replaced by CRLF the
ext time Git touches it

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scr
m-project (feature-user-story-2)
$ git commit -m "Funcionalidad aún en progreso"
[feature-user-story-2 035c6d4] Funcionalidad aún en progreso
 1 file changed, 1 insertion(+)
 create mode 100644 progress.md
```

Revisamos los commits y nos movemos al main

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad
m-project (feature-user-story-2)
$ git log --oneline
035c6d4 (HEAD -> feature-user-story-2) Funcionalidad aún en progreso
2151ea2 Funcionalidad lista para revisión
0878607 Commit inicial en main

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad
m-project (feature-user-story-2)
$ git checkout main
Switched to branch 'main'
```

Con cherry-pick copiaremos un commit de otra rama y lo aplica como nuevo commit en la rama actual

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/Desar
m-project (main)
$ git cherry-pick 2151ea2
[main d788597] Funcionalidad lista para revisión
 Date: Tue Apr 22 01:20:22 2025 -0500
 1 file changed, 1 insertion(+)
 create mode 100644 feature2.md
```

Pregunta:

¿Cómo ayuda git cherry-pick a mostrar avances de forma selectiva en un sprint review?

git cherry-pick permite mover solo los commits que están listos desde una rama de desarrollo hacia main, sin traer cambios incompletos. Esto ayuda a mostrar avances específicos y funcionales durante el sprint review. Así, el equipo puede presentar solo lo terminado sin afectar la estabilidad del código.

Fase 4:

nos movemos de rama

```
m-project (main)
$ git checkout feature-user-story-1
Switched to branch 'feature-user-story-1'
```

Y creamos un archivo conflicto

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (feature-user-story-1)
$ echo "Cambio en la misma línea desde feature 1" > conflicted-file.md
```

Añadimos y hacemos el commit

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (feature-user-story-1)
$ git add conflicted-file.md
warning: in the working copy of 'conflicted-file.md', LF will be replaced by CRL
F the next time Git touches it

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (feature-user-story-1)
$ git commit -m "Cambio en feature 1"
[feature-user-story-1 2d1dab3] Cambio en feature 1
 1 file changed, 1 insertion(+)
 create mode 100644 conflicted-file.md
```

Cambiamos a la rama feature 2

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop
m-project (feature-user-story-1)
$ git checkout feature-user-story-2
Switched to branch 'feature-user-story-2'
```

Creamos el conflicto:

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/actividad6/scru
m-project (feature-user-story-2)
$ echo "Cambio diferente en la misma línea desde feature 2" > conflicted-file.md
```

Añadimos y realizamos el commit

```
m-project (feature-user-story-2)
$ git add conflicted-file.md
warning: in the working copy of 'conflicted-file.md'
F the next time Git touches it

windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/Desarrol
m-project (feature-user-story-2)
$ git commit -m "Cambio en feature 2"
[feature-user-story-2 63d1918] Cambio en feature 2
 1 file changed, 1 insertion(+)
 create mode 100644 conflicted-file.md
```

Visualizando el conflicto

```
windows10@DESKTOP-UEI1KU9 MINGW64 ~/Desktop/DesarrolloDeSoftware/ac
m-project (main)
$ git merge feature-user-story-2
Auto-merging conflicted-file.md
CONFLICT (add/add): Merge conflict in conflicted-file.md
Automatic merge failed; fix conflicts and then commit the result.
```

Pregunta:

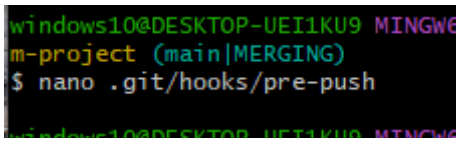
¿Cómo manejas los conflictos de fusión al final de un sprint? ¿Cómo puede el equipo mejorar la comunicación para evitar conflictos grandes?

Al final de un sprint, manejo los conflictos de fusión resolviendo manualmente los archivos en conflicto, validando que el código funcione, y luego haciendo commit. Para reducir conflictos, es clave hacer rebases frecuentes con main

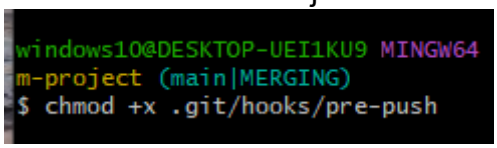
durante el desarrollo. El equipo puede mejorar la comunicación coordinando integraciones y revisando cambios en conjunto antes de hacer merge.

Fase 5:

Creamos el hook, guardamos y salimos luego de poner los comandos solicitados



Realizamos el hook ejecutable:



Preguntas

1. **Ejercicio para git checkout --ours y git checkout --theirs**

Contexto: En un sprint ágil, dos equipos están trabajando en diferentes ramas. Se produce un conflicto de fusión en un archivo de configuración crucial. El equipo A quiere mantener sus cambios mientras el equipo B solo quiere conservar los suyos. El proceso de entrega continua está detenido debido a este conflicto.

Pregunta:
¿Cómo utilizarías los comandos git checkout --ours y git checkout --theirs para resolver este conflicto de manera rápida y eficiente? Explica cuándo preferirías usar cada uno de estos comandos y cómo impacta en la pipeline de CI/CD. ¿Cómo te asegurarías de que la resolución elegida no comprometa la calidad del código?

Para resolver el conflicto, usamos git checkout --ours para conservar los cambios del equipo A y git checkout --theirs para mantener los del equipo B. Este enfoque acelera la resolución, permitiendo a cada equipo conservar sus cambios específicos. Asegúrate de probar ambos casos en un entorno local para garantizar que no se rompa el flujo de CI/CD.

2. **Ejercicio para git diff**

Contexto: Durante una revisión de código en un entorno ágil, se observa que un pull request tiene una gran cantidad de cambios, muchos de los cuales no están relacionados con la funcionalidad principal. Estos cambios podrían generar conflictos con otras ramas en la pipeline de CI/CD.

Pregunta:
Utilizando el comando git diff, ¿cómo compararías los cambios entre ramas para identificar diferencias específicas en archivos críticos?

Explica cómo podrías utilizar git diff feature-branch..main para detectar posibles conflictos antes de realizar una fusión y cómo esto contribuye a mantener la estabilidad en un entorno ágil con CI/CD.

Utilizamos git diff feature-branch..main para comparar las diferencias entre la rama principal y la rama de características. Esto te permite identificar archivos críticos modificados antes de una fusión, previniendo posibles conflictos en CI/CD. Detectar estas diferencias temprano asegura que la estabilidad del proyecto se mantenga durante el proceso de integración.

3. **Ejercicio para git merge --no-commit --no-ff**

Contexto: En un proyecto ágil con CI/CD, tu equipo quiere simular una fusión entre una rama de desarrollo y la rama principal para ver cómo se comporta el código sin comprometerlo inmediatamente en el repositorio. Esto es útil para identificar posibles problemas antes de completar la fusión.

Pregunta:
Describe cómo usarías el comando git merge --no-commit --no-ff para simular una fusión en tu rama local. ¿Qué ventajas tiene esta práctica en un flujo de trabajo ágil con CI/CD, y cómo ayuda a minimizar errores antes de hacer commits definitivos? ¿Cómo automatizarías este paso dentro de una pipeline CI/CD?

Con git merge --no-commit --no-ff, simulas una fusión sin confirmar los cambios, lo que permite evaluar el impacto antes de hacer un commit. Esta práctica ayuda a identificar conflictos antes de afectar la pipeline de CI/CD. Puedes automatizar este paso en CI/CD ejecutando pruebas previas a la fusión real para minimizar errores.

Ejercicio para git mergetool

Contexto: Tu equipo de desarrollo utiliza herramientas gráficas para resolver conflictos de manera colaborativa. Algunos desarrolladores prefieren herramientas como vimdiff o Visual Studio Code. En medio de un sprint, varios archivos están en conflicto y los desarrolladores prefieren trabajar en un entorno visual para resolverlos.

Pregunta:
Explica cómo configurarías y utilizarías git mergetool en tu equipo para integrar herramientas gráficas que faciliten la resolución de conflictos. ¿Qué impacto tiene el uso de git mergetool en un entorno de trabajo ágil con CI/CD, y cómo aseguras que todos los miembros del equipo mantengan consistencia en las resoluciones?

configuramos git mergetool para usar herramientas gráficas como vimdiff o Visual Studio Code, facilitando la resolución de conflictos. Esto mejora la eficiencia en un entorno ágil, permitiendo visualización clara de los cambios. Asegura consistencia mediante la configuración común de la herramienta en todos los miembros del equipo, manteniendo la calidad del código.

4. **Ejercicio para git reset**

Contexto: En un proyecto ágil, un desarrollador ha hecho un commit que rompe la pipeline de CI/CD. Se debe revertir el commit, pero se necesita hacerlo de manera que se mantenga el código en el directorio de trabajo sin deshacer los cambios.

Pregunta:
Explica las diferencias entre git reset --soft, git reset --mixed y git reset --hard. ¿En qué escenarios dentro de un flujo de trabajo ágil con CI/CD utilizarías cada uno? Describe un caso en el que usarías git reset --mixed para corregir un commit sin perder los cambios no commiteados y cómo afecta esto a la pipeline.

git reset --soft conserva los cambios no confirmados, git reset --mixed mantiene los cambios en el directorio de trabajo y git reset --hard elimina todo. En un flujo ágil, git reset --mixed es útil cuando necesitas revertir un commit sin perder cambios. Esto ayuda a resolver problemas sin afectar la integridad de la pipeline.

5. **Ejercicio para git revert**

Contexto: En un entorno de CI/CD, tu equipo ha desplegado una característica a producción, pero se ha detectado un bug crítico. La rama principal debe revertirse para restaurar la estabilidad, pero no puedes modificar el historial de commits debido a las políticas del equipo.

Pregunta:
Explica cómo utilizarías git revert para deshacer los cambios sin modificar el historial de commits. ¿Cómo te aseguras de que esta acción no afecte la pipeline de CI/CD y permita una rápida recuperación del sistema? Proporciona un ejemplo detallado de cómo revertirías varios commits consecutivos.

Usa git revert para deshacer un commit sin modificar el historial, creando un nuevo commit que deshace los cambios. Esto asegura que la estabilidad de la pipeline de CI/CD no se vea comprometida. Para revertir varios commits consecutivos, ejecuta git revert para cada uno, garantizando que los cambios se reviertan de manera controlada.

6. **Ejercicio para git stash**

Contexto: En un entorno ágil, tu equipo está trabajando en una corrección de errores urgente mientras tienes cambios no guardados en tu directorio de trabajo que aún no están listos para ser committeados. Sin embargo, necesitas cambiar rápidamente a una rama de hotfix para trabajar en la corrección.

Pregunta:
Explica cómo utilizarías git stash para guardar temporalmente tus cambios y volver a ellos después de haber terminado el hotfix. ¿Qué impacto tiene el uso de git stash en un flujo de trabajo ágil con CI/CD

cuando trabajas en múltiples tareas? ¿Cómo podrías automatizar el proceso de *stashing* dentro de una pipeline CI/CD?

Utiliza git stash para guardar temporalmente los cambios y cambiar de rama sin perder el trabajo pendiente. Esto te permite trabajar en tareas urgentes, como un hotfix, sin comprometer cambios incompletos. Automatizar el stashing en CI/CD ayuda a gestionar múltiples tareas sin interferir en el flujo de trabajo general del equipo.

7. **Ejercicio para .gitignore**

Contexto: Tu equipo de desarrollo ágil está trabajando en varios entornos locales con configuraciones diferentes (archivos de logs, configuraciones personales). Estos archivos no deberían ser parte del control de versiones para evitar confusiones en la pipeline de CI/CD.

Pregunta:
Diseña un archivo .gitignore que excluya archivos innecesarios en un entorno ágil de desarrollo. Explica por qué es importante mantener este archivo actualizado en un equipo colaborativo que utiliza CI/CD y cómo afecta la calidad y limpieza del código compartido en el repositorio.

Diseña un archivo .gitignore para excluir archivos como logs y configuraciones locales que no deben ser versionados. Mantenerlo actualizado es crucial para evitar que archivos no deseados lleguen al repositorio y rompan la pipeline de CI/CD. Este control asegura que solo se versionen los archivos relevantes para el desarrollo del proyecto.

Ejercicios adicionales

Ejercicio 1: Resolución de conflictos en un entorno ágil

Contexto:
Estás trabajando en un proyecto ágil donde múltiples desarrolladores están enviando cambios a la rama principal cada día. Durante una integración continua, se detectan conflictos de fusión entre las ramas de dos equipos que están trabajando en dos funcionalidades críticas. Ambos equipos han modificado el mismo archivo de configuración del proyecto.

Pregunta:

- **¿Cómo gestionarías la resolución de este conflicto de manera eficiente utilizando Git y manteniendo la entrega continua sin interrupciones? ¿Qué pasos seguirías para minimizar el impacto en la CI/CD y asegurar que el código final sea estable?**

  Para gestionar el conflicto eficientemente, primero identificaría qué cambios específicos generaron el conflicto y trabajaría con ambos equipos para acordar la mejor solución. Utilizaría una rama temporal para resolver el conflicto, ejecutaría pruebas locales y de integración, y luego fusionaría

nuevamente a la rama principal. Finalmente, garantizaría que la pipeline de CI/CD corra sin interrupciones al hacer commits frecuentes y pequeños.

Ejercicio 2: Rebase vs. Merge en integraciones ágiles

Contexto:
En tu equipo de desarrollo ágil, cada sprint incluye la integración de varias ramas de características. Algunos miembros del equipo prefieren realizar merge para mantener el historial completo de commits, mientras que otros prefieren rebase para mantener un historial lineal.

Pregunta:

- **¿Qué ventajas y desventajas presenta cada enfoque (merge vs. rebase) en el contexto de la metodología ágil? ¿Cómo impacta esto en la revisión de código, CI/CD, y en la identificación rápida de errores?**

El merge preserva un historial completo de cambios, lo que facilita el seguimiento, pero puede generar un historial menos limpio y más complejo. El rebase mantiene un historial lineal, facilitando la revisión del código y la identificación de errores, pero puede complicar la gestión de conflictos. En CI/CD, ambos enfoques son válidos, pero el rebase puede hacer que la integración continua sea más sencilla y menos propensa a conflictos.

Ejercicio 3: Git Hooks en un flujo de trabajo CI/CD ágil

Contexto:
Tu equipo está utilizando Git y una pipeline de CI/CD que incluye tests unitarios, integración continua y despliegues automatizados. Sin embargo, algunos desarrolladores accidentalmente comiten código que no pasa los tests locales o no sigue las convenciones de estilo definidas por el equipo.

Pregunta:

- **Diseña un conjunto de Git Hooks que ayudaría a mitigar estos problemas, integrando validaciones de estilo y tests automáticos antes de permitir los commits. Explica qué tipo de validaciones implementarías y cómo se relaciona esto con la calidad del código y la entrega continua en un entorno ágil.**

Implementaría hooks pre-commit para validar el estilo de código y pre-push para ejecutar pruebas unitarias, evitando commits o pushes que no pasen las validaciones. Esto aseguraría que solo se suba código que pase las pruebas y siga las convenciones del equipo, mejorando la calidad y reduciendo errores. Al integrarlo con CI/CD, se evitarían interrupciones y se mantendría la estabilidad del proyecto.

Ejercicio 4: Estrategias de branching en metodologías ágiles

Contexto:
Tu equipo de desarrollo sigue una metodología ágil y está utilizando Git Flow

para gestionar el ciclo de vida de las ramas. Sin embargo, a medida que el equipo ha crecido, la gestión de las ramas se ha vuelto más compleja, lo que ha provocado retrasos en la integración y conflictos de fusión frecuentes.

Pregunta:

- **Explica cómo adaptarías o modificarías la estrategia de branching para optimizar el flujo de trabajo del equipo en un entorno ágil y con integración continua. Considera cómo podrías integrar feature branches, release branches y hotfix branches de manera que apoyen la entrega continua y minimicen conflictos.**

Optimizaría el flujo de trabajo utilizando ramas de características pequeñas, con una estrategia de integración frecuente (por ejemplo, integración diaria) para reducir conflictos. Integraría ramas de hotfix y release de manera que se fusionen rápidamente a la rama principal tras pruebas exhaustivas. La clave es mantener un ciclo continuo de integración, con merges frecuentes y bien gestionados.

Ejercicio 5: Automatización de reversiones con git en CI/CD

Contexto:
Durante una integración continua en tu pipeline de CI/CD, se detecta un bug crítico después de haber fusionado varios commits a la rama principal. El equipo necesita revertir los cambios rápidamente para mantener la estabilidad del sistema.

Pregunta:

- **¿Cómo diseñarías un proceso automatizado con Git y CI/CD que permita revertir cambios de manera eficiente y segura? Describe cómo podrías integrar comandos como git revert o git reset en la pipeline y cuáles serían los pasos para garantizar que los bugs se reviertan sin afectar el desarrollo en curso.**

Para revertir cambios rápidamente, implementaría un paso automático en la pipeline de CI/CD que ejecute git revert para deshacer commits defectuosos o git reset para situaciones más críticas. Aseguraría que el proceso esté probado en un entorno de staging antes de hacer el rollback en producción, garantizando que el sistema permanezca estable y no afecte otros desarrollos en curso.