Informe de Ambientes

Proyecto: Experiencias Significativas

ÍNDICE

1. Introducción

El proyecto *Experiencias Significativas* implementa una arquitectura de control de versiones basada en múltiples ambientes, con el objetivo de asegurar la estabilidad del código, mejorar el flujo de trabajo del equipo y garantizar que cada funcionalidad pase por un proceso adecuado de revisión, pruebas y validación.

El uso de ambientes separados permite evitar que cambios no probados impacten directamente en la versión estable del sistema. Cada ambiente (dev, qa, staging y main) está diseñado para cumplir una función específica en el ciclo de desarrollo, desde la creación inicial del código hasta su despliegue final.

Este informe explica en detalle cada uno de estos ambientes, los flujos de trabajo utilizados, los procesos aplicados por el equipo, ejemplos visuales del repositorio y problemas detectados durante la ejecución del proyecto.

2. Objetivos del documento

- Describir los ambientes utilizados durante el desarrollo.
- Explicar el flujo de trabajo aplicado en GitHub.
- Documentar el proceso aplicado en frontend, backend y móvil.
- Presentar ejemplos reales con comandos y ramas creadas.
- Proveer una guía clara para futuros desarrolladores del proyecto.

3. Estructura de Ambientes en GitHub

3.1 ¿Qué es un Ambiente de control de versions?

Un ambiente es un espacio lógico que representa una etapa del software dentro del ciclo de vida del Desarrollo. Permite separar Código experimental de Código estable.

3.2 Ambiente utilizados en Experiencias Significativas

Ambiente | Propósito

dev | Desarrollo diario y nuevas funcionalidades.

qa | Validación funcional y pruebas internas.

staging | Simulación de producción antes del despliegue.

main | Versión final estable del Proyecto.

4. Flujo de trabajo por ambientes

4.1 Ambiente dev

Primer entorno de desarrollo donde se crean y prueban nuevas funcionalidades.

Flujo estándar:

- git pull origin dev
- git switch dev
- git switch -c HU-01-dev
- realizar Desarrollo
- crear MR de HU-01-dev → dev
- Nota: El desarrollo se cierra con la rama dev update.

Propósito del ambiente dev:

- Permitir iteraciones rápidas sin afectar las ramas principales.
- Facilitar la integración continua de nuevas funcionalidades.
- Detectar errores tempranos antes de pasar a QA.
- Asegurar que cada cambio esté documentado en commits individuales.

4.2. Ambiente qa

Ambiente para pruebas de calidad, donde se valida que las funcionalidades que vienen de dev cumplen lo requerido.

Flujo estándar:

- git pull origin qa

- git switch qa
- git switch -c HU-01-qa
- realizar desarrollo QA
- crear MR de HU-01-qa → qa
- Nota: El desarrollo se cierra con la rama qa update.

Propósito del ambiente qa:

- Realizar pruebas manuales y automáticas.
- Validar reglas de negocio.
- Asegurar consistencia en el sistema.
- Prevenir que cambios defectuosos lleguen a staging

4.3 Ambiente staging

Ambiente previo a producción. Aquí se valida la integración completa del sistema.

Flujo estándar:

- git pull origin staging
- git switch staging
- git switch -c HU-01-staging
- realizar desarrollo staging
- crear MR de HU-01-staging → staging
- Nota: El desarrollo se cierra con la rama staging update.

Propósito del ambiente staging:

- Simular el comportamiento final del sistema.
- Validar la interacción entre múltiples módulos.
- Verificar la compatibilidad antes del despliegue final.
- Aprobar o rechazar el release temporal.

4.4 Ambiente main

Ambiente principal donde solo se almacenan releases aprobados. No se trabaja directamente sobre esta rama.

Flujo estándar:

- git pull origin main
- git switch main
- git switch -c release.1.1

  HU-01-release.1.1

  HU-02-release.1.1
- crear MR release.1.1 → main

Propósito del ambiente main:

- Representar la versión estable del proyecto.
- Asegurar que el código en main siempre sea funcional.
- Evitar cambios directos y mantener control total con releases.
- Servir como base para las versiones de producción.

5. Uso de ambientes por repositorio

- **Frontend Web** – Implementación completa y correcta del flujo de ambientes.
- **Backend** – Implementación completa, con control adecuado de merges y conflictos.
- **Aplicación móvil** – Flujo implementado con éxito, siguiendo la misma estructura.

Cada repositorio mantiene independencia en su ciclo de versionamiento, pero todos comparten la misma estructura de ambientes para mantener uniformidad y control.

6. Problemas encontrados y soluciones implementadas

Durante el ciclo de desarrollo se presentaron dificultades que permitieron fortalecer el flujo de trabajo.

- ✓ **Conflictos entre ramas**

  Solución:

  - Estandarizar commits.
  - Mantener ramas actualizadas con dev antes de trabajar.

- ✓ **Fallos de control de versiones en mobile**

  Solución:

  - Documentar flujos.
  - Crear ramas más ordenadas.

- ✓ **Dependencias rotas en backend**

  Solución:

  - Unificación de versiones.
  - Revisión del package.json / dependencias internas.

- ✓ **Ramas abiertas sin cerrar**

  Solución:

  - Política estricta de cierre de ramas.
  - Limpieza mensual del repositorio.

7. Ejemplos con capturas

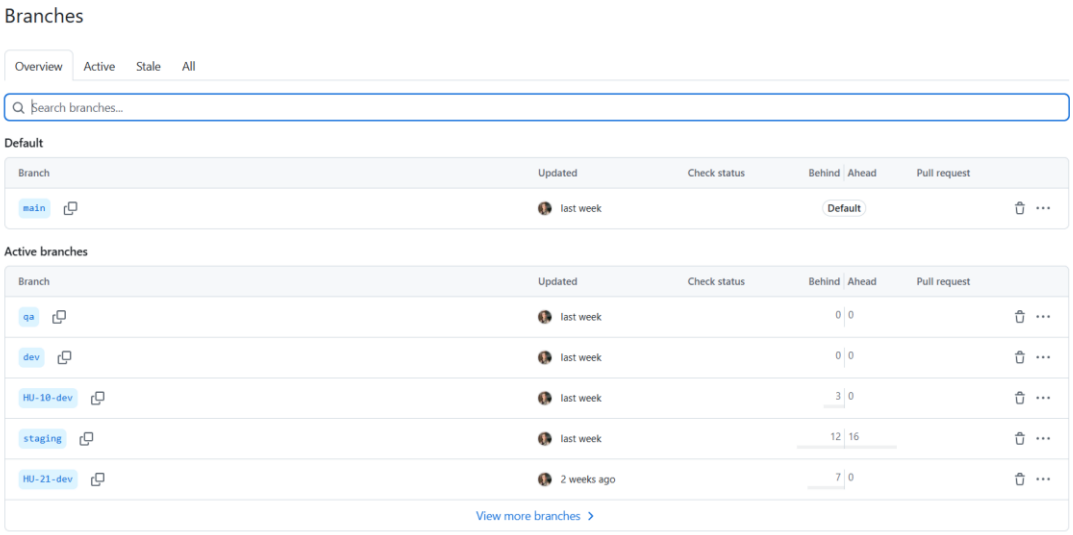
Las siguientes evidencias visuales documentan el uso real del flujo de control de versiones implementado en el proyecto Experiencias Significativas. Cada imagen corresponde a una etapa distinta del ciclo de desarrollo y demuestra la correcta aplicación de buenas prácticas en Git.

El uso de capturas permite validar de forma visual los procesos descritos anteriormente y sirve como respaldo técnico del trabajo realizado por el equipo.

7.1 Ambientes principales configurados en GitHub.

Aqui se muestran las cuatro ramas base del flujo de desarollo:

- **main**: versión estable y publicada (producción).
- **dev**: ambiente destinado al desarrollo activo de funcionalidades.
- **qa**: ambiente donde se realizan pruebas internas de validación.
- **staging**: ambiente de preproducción donde se simula el despliegue final.



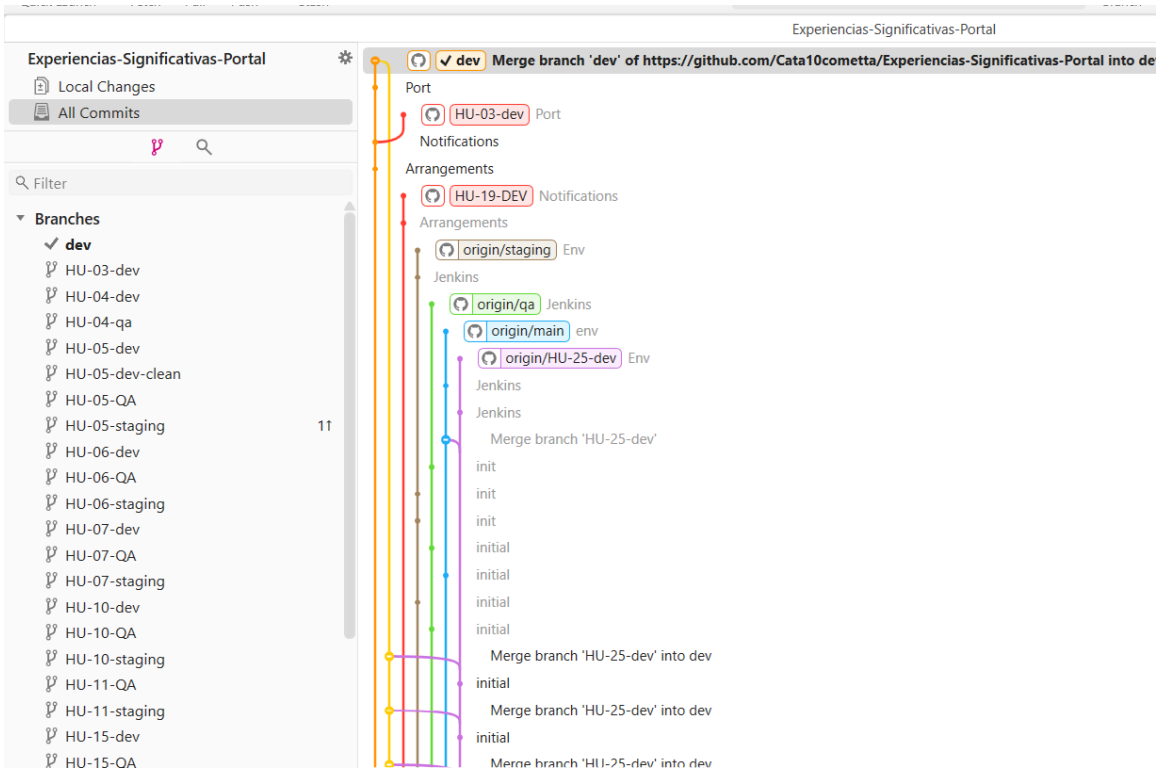
También se visualizan ramas individuales de Historia de Usuario, como HU-10-dev, lo que demuestra la aplicación correcta del flujo GitFlow adaptado al Proyecto.

7.2 Vista de ramas y flujo de commits del Proyecto

En esta imagen se observa la estructura completa de las ramas utilizadas en el proyecto *Experiencias Significativas*, incluyendo las ramas por Historia de Usuario en sus diferentes ambientes (*dev*, *qa*, *staging*).

Esta vista permite visualizar :

- El recorrido de las HU en dev, qa y staging.
- Los cherry-picks realizados para mover cambios puntuales.
- Los merges hacia dev y staging.
- Los commits del equipo organizados por cada funcionalidad.



Además, este es el espacio donde se realizan los ***cherry-picks*** para traer commits específicos desde una rama hacia otra.

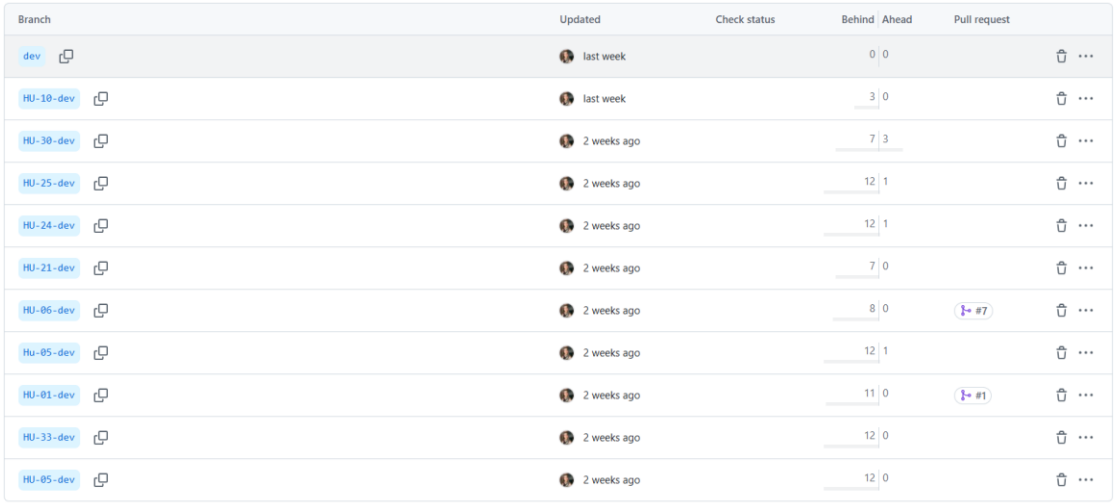
7.3 Ramas de Historias de Usuario en el ambiente dev.

Esta vista muestra todas las ramas utilizadas para el desarrollo activo. Cada rama corresponde a una Historia de Usuario específica y permite trabajar funciones de forma aislada.

- Mensajes de commit descriptivos.

- Separación clara de funcionalidades.

- Registro cronológico del avance del proyecto.

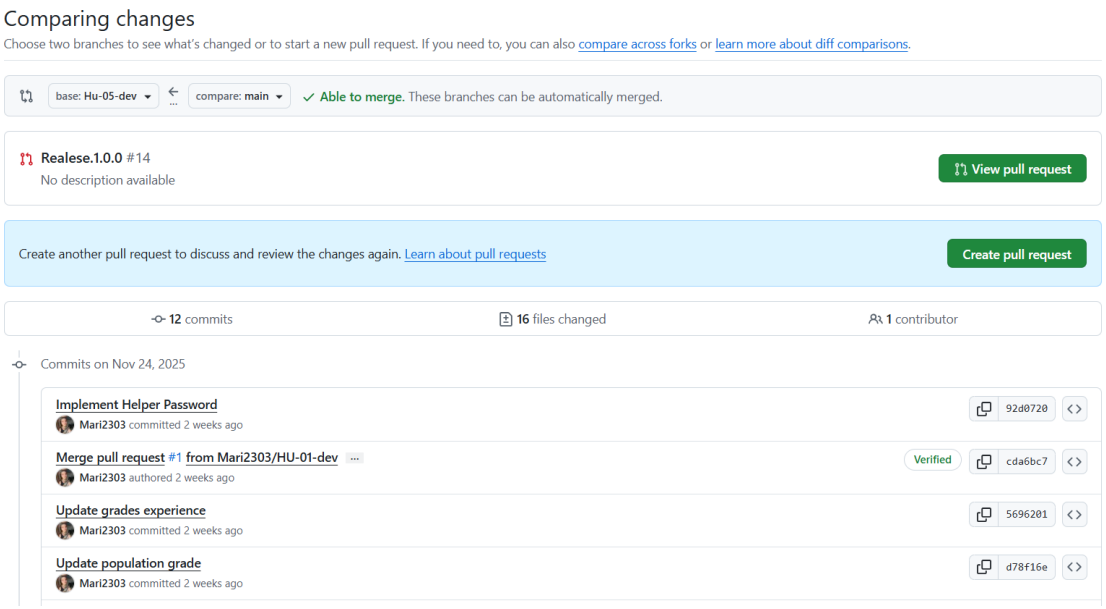


Branch	Updated	Check status	Behind	Ahead	Pull request
dev	last week		0	0	
HU-10-dev	last week		3	0	
HU-30-dev	2 weeks ago		7	3	
HU-25-dev	2 weeks ago		12	1	
HU-24-dev	2 weeks ago		12	1	
HU-21-dev	2 weeks ago		7	0	
HU-06-dev	2 weeks ago		8	0	#7
Hu-05-dev	2 weeks ago		12	1	
HU-01-dev	2 weeks ago		11	0	#1
HU-33-dev	2 weeks ago		12	0	
HU-05-dev	2 weeks ago		12	0	

7.4 Merge Request desde HU-XX-dev hacia dev.

El Merge Request es un paso obligatorio para integrar código.

Aquí se revisa la calidad del código y se validan los cambios antes de aprobarlos.



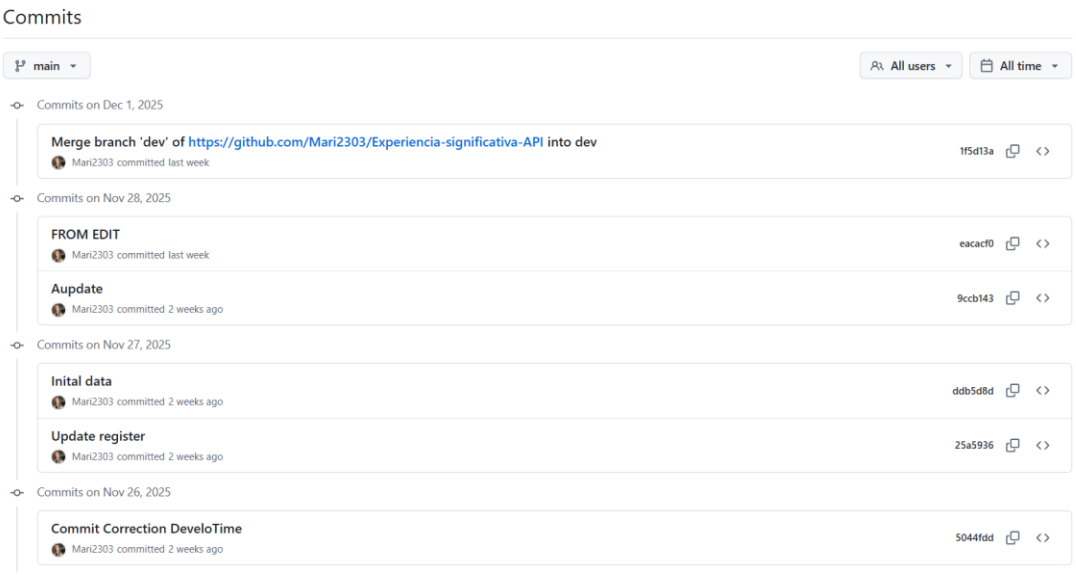
7.5 Historial de Commits del Proyecto

En esta imagen se visualiza el registro completo de todos los commits realizados.

Permite identificar quién realizó cada cambio, cuándo y en qué rama.

Se analizan:

- Mensajes descriptivos
- Secuencia de trabajo
- Frecuencia de Desarrollo
- Calidad de los commits



7.6 Conflicto generado durante un cherry-pick realizado desde Fork, visualizado en Merge Changes dentro de Visual Studio Code.

En esta captura se observa un escenario real ocurrido durante el desarrollo del proyecto Experiencias Significativas, en el cual se realizó un cherry-pick utilizando la herramienta gráfica Fork. El objetivo del cherry-pick era trasladar un commit específico desde una rama de Historia de Usuario hacia otra rama correspondiente a un ambiente diferente, sin realizar un merge completo.

Durante este proceso, Fork notificó que el cherry-pick no pudo completarse de forma automática debido a la presencia de conflictos de código. Estos conflictos se producen cuando el commit que se intenta aplicar modifica líneas de código que ya han sido alteradas previamente en la rama destino, generando ambigüedad para Git al momento de decidir

qué versión conservar.

Posteriormente, al abrir el proyecto en Visual Studio Code, los archivos involucrados en el conflicto fueron listados automáticamente dentro de la sección “Merge Changes”, lo que indica que Git ha detenido el proceso y requiere intervención manual del desarrollador.

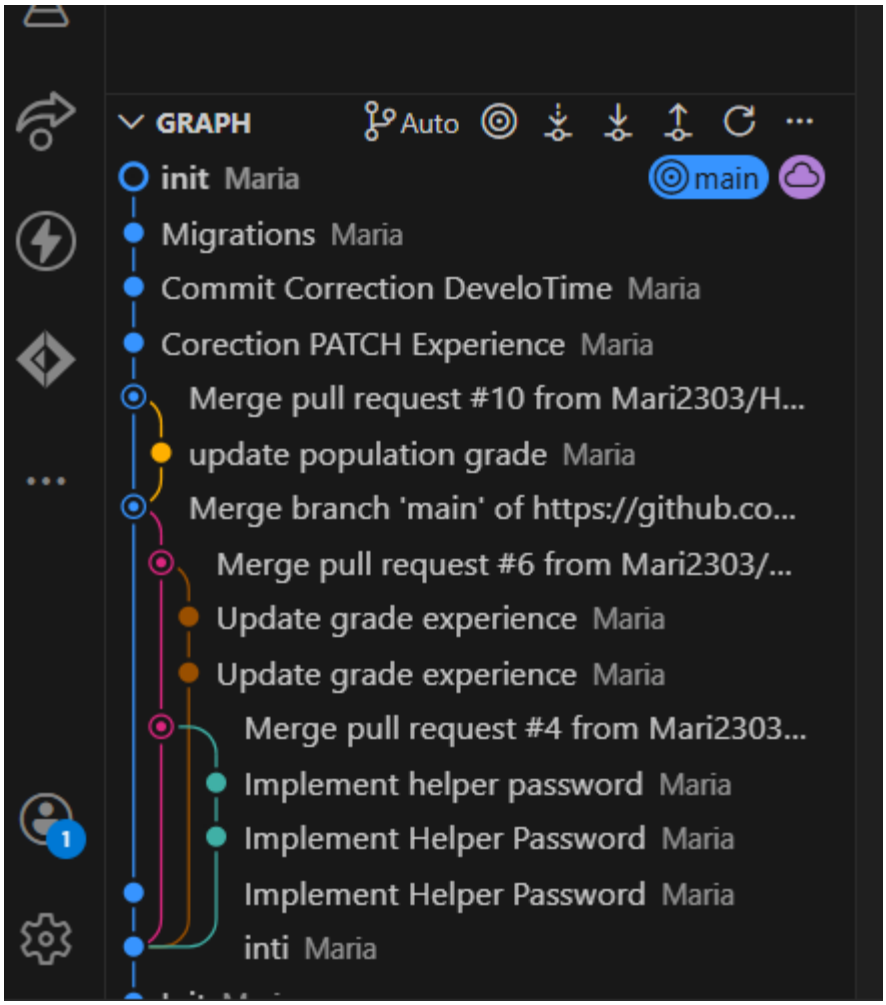
Este escenario es común cuando se trabaja con varios ambientes (dev, qa, staging) y múltiples ramas activas, ya que los cambios pueden haberse modificado de manera diferente en cada contexto.

7.7 Visualización del gráfico de commits (GRAPH) que muestra el historial de ramas y movimientos del proyecto.

En esta imagen se observa la representación gráfica del flujo de trabajo Git utilizado en el proyecto *Experiencias Significativas*.

El GRAPH permite visualizar:

- La línea de tiempo de todos los commits realizados.
- La separación entre ramas como *dev*, *qa*, *staging* y *main*.
- Las ramas individuales creadas para cada Historia de Usuario (HU-XX-dev, HU-XX-qa, HU-XX-staging).
- Los merges entre ramas y el recorrido que siguen las funcionalidades.
- Los puntos específicos donde se realizan **cherry-picks**, permitiendo traer commits puntuales desde otras ramas sin necesidad de hacer un merge completo.
- Se aprecia la estabilidad en main al no recibir cambios directos.
-



Esta vista es fundamental para entender cómo evoluciona el proyecto y cómo se integran los cambios entre los diferentes ambientes, asegurando trazabilidad y orden en el ciclo de desarrollo.

8. Buenas prácticas en la gestión de ambientes

El uso adecuado de ambientes dentro de un proyecto de desarrollo de software requiere la aplicación de buenas prácticas que garanticen orden, estabilidad y calidad en el código. En el proyecto *Experiencias Significativas*, se implementaron diversas prácticas que permitieron un manejo correcto del control de versiones y una mejor colaboración entre los integrantes del equipo.

Una de las principales buenas prácticas fue la **prohibición de trabajar directamente sobre las ramas principales** como *main*, *staging*, *qa* y *dev*. Todo cambio debía realizarse mediante ramas específicas de Historias de Usuario, lo que evitó errores críticos y sobrescritura de código.

Asimismo, se estableció el uso obligatorio de **Merge Requests**, los cuales permitieron revisar los cambios antes de integrarlos a un ambiente superior. Esta práctica ayudó a detectar errores tempranamente y a mantener la calidad del código.

Otra buena práctica importante fue la **estandarización de nombres de ramas**, utilizando una nomenclatura clara como HU-XX-dev, HU-XX-qa y HU-XX-staging. Esto facilitó la identificación rápida del estado de cada funcionalidad dentro del proyecto.

Finalmente, se promovió el uso de **commits descriptivos**, donde cada mensaje reflejara claramente el cambio realizado y la Historia de Usuario relacionada, mejorando la trazabilidad del proyecto.

9. Roles del equipo en el manejo de ambientes

El correcto manejo de los ambientes no solo depende de la herramienta utilizada, sino también de la correcta asignación de roles dentro del equipo de trabajo. Aunque el proyecto contó con un equipo reducido, se identificaron roles clave que facilitaron la gestión del control de versiones.

El **desarrollador** fue responsable de crear las ramas de Historias de Usuario, realizar commits y ejecutar pruebas iniciales en el ambiente *dev*.

El **rol de QA** se encargó de validar el correcto funcionamiento de las funcionalidades en el ambiente *qa*.

El **integrador** fue quien consolidó los cambios en *staging* y preparó los releases para *main*.

Finalmente, el **documentador** registró el proceso, generó evidencias visuales y documentó los flujos aplicados.

Esta división de responsabilidades permitió mantener un proceso ordenado y eficiente.

10.Impacto de los Ambientes en la Calidad del Proyecto

El uso de múltiples ambientes mejora considerablemente la calidad del software.

10.1. Ventajas en la gestión de la calidad

El principal impacto de trabajar con distintos ambientes es que se minimiza el riesgo de errores en producción. Gracias a los ambientes de prueba y preproducción:

- **Reducción de riesgos**: Los errores se detectan y corrigen antes de llegar a producción.
- **Validación de funcionalidad**: Las funcionalidades se validan en entornos específicos antes de su integración.
- **Pruebas en entornos controlados**: Permiten realizar pruebas de integración, de carga y de usuario sin afectar el producto final.

10.2. Desventajas de la Implementación de Múltiples Ambientes

Si bien el uso de múltiples ambientes ofrece ventajas sustanciales, también puede conllevar algunos desafíos.

10.3. Complejidad en la gestión de ambientes

El principal inconveniente es la **complejidad** de gestionar múltiples ambientes, especialmente si no se tiene una automatización adecuada. Los errores de sincronización entre ambientes pueden generar inconsistencias que dificultan el trabajo del equipo.

10.4. Recursos adicionales necesarios

Mantener varios ambientes activos requiere mayor infraestructura, lo que implica un gasto adicional en recursos y herramientas de monitoreo.

11. Uso de Cherry-Pick en la herramienta Fork

El **cherry-pick** es una funcionalidad de Git que permite seleccionar un commit específico de una rama y aplicarlo en otra, sin necesidad de realizar un merge completo entre ramas. En el proyecto *Experiencias Significativas*, esta técnica fue utilizada principalmente para trasladar cambios puntuales entre diferentes ambientes, especialmente cuando una corrección o mejora desarrollada en una rama debía replicarse en otra sin incluir modificaciones adicionales.

Para la ejecución de los cherry-picks se utilizó la herramienta gráfica **Fork**, la cual facilita la visualización del historial de commits y la selección precisa del cambio que se desea trasladar. Fork permite trabajar de manera visual con Git, reduciendo errores comunes y haciendo más intuitivo el manejo de ramas y commits, especialmente en proyectos con múltiples ambientes como *dev*, *qa* y *staging*.

El proceso general seguido para realizar un cherry-pick en Fork fue el siguiente:

1. Seleccionar la rama destino correspondiente al ambiente donde se necesita aplicar el cambio.
2. Ubicar en el historial el commit específico que se desea trasladar.
3. Ejecutar la opción **Cherry-pick commit** desde la interfaz de Fork.
4. Validar si el commit puede aplicarse de forma automática o si se presentan conflictos.

En varios casos, el cherry-pick se aplicó correctamente sin intervención adicional, lo que permitió corregir errores o sincronizar funcionalidades de manera rápida y eficiente. Sin

embargo, también se presentaron escenarios en los que Fork notificó la existencia de **conflictos**, indicando que el commit no podía integrarse automáticamente debido a diferencias entre el código de la rama origen y la rama destino.

Cuando se detectó un conflicto durante el cherry-pick, Fork detuvo el proceso y mostró una alerta indicando que la integración requería resolución manual. En estos casos, el proyecto fue abierto en **Visual Studio Code**, donde los archivos conflictivos aparecieron listados en la sección **"Merge Changes"**. Esta sección permitió identificar con claridad los archivos afectados y las líneas de código que generaron el conflicto.

Los conflictos se produjeron principalmente porque:

- El mismo archivo había sido modificado previamente en la rama destino.
- La estructura del código había cambiado entre ambientes.
- Dos Historias de Usuario distintas afectaban una misma sección del código.

Visual Studio Code facilitó la resolución de estos conflictos mediante herramientas visuales que permiten comparar los cambios actuales con los cambios provenientes del cherry-pick. El desarrollador pudo decidir qué versión conservar o cómo combinar ambas, asegurando que la funcionalidad final fuera correcta y coherente con el estado del proyecto.

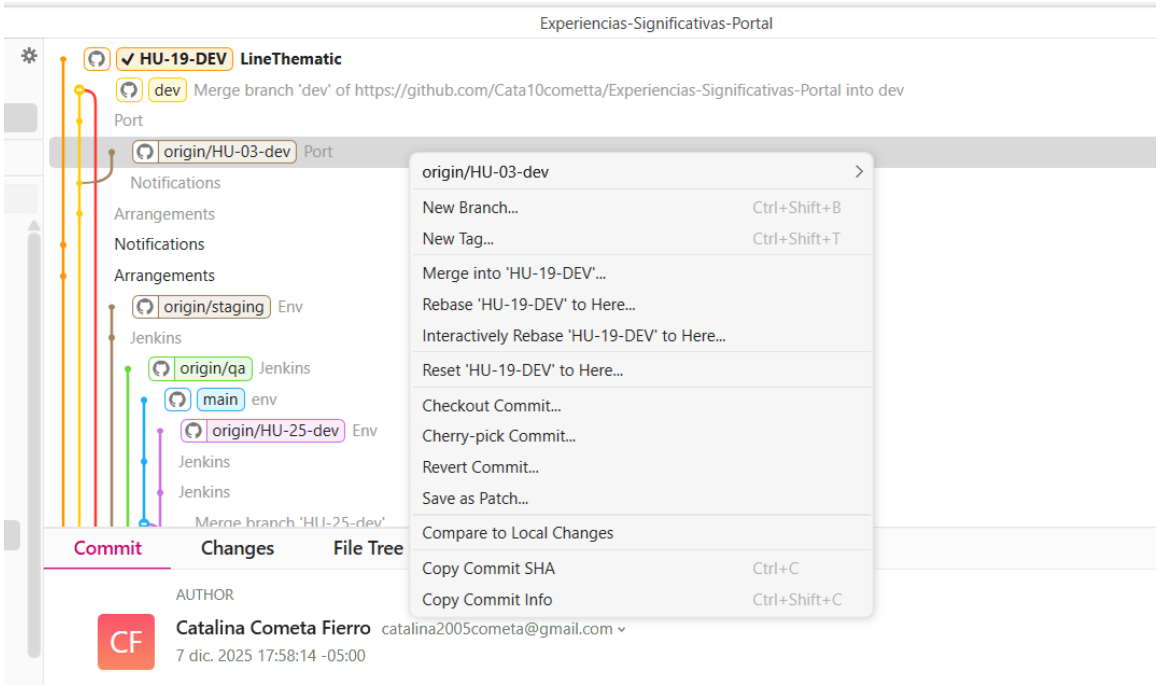
Una vez resueltos los conflictos, el proceso de cherry-pick se completó exitosamente, permitiendo continuar con el flujo normal de trabajo. Esta práctica demostró ser especialmente útil para mantener sincronizados los ambientes sin comprometer la estabilidad del proyecto ni introducir cambios innecesarios.

El uso del cherry-pick en Fork aportó múltiples beneficios al proyecto, entre ellos:

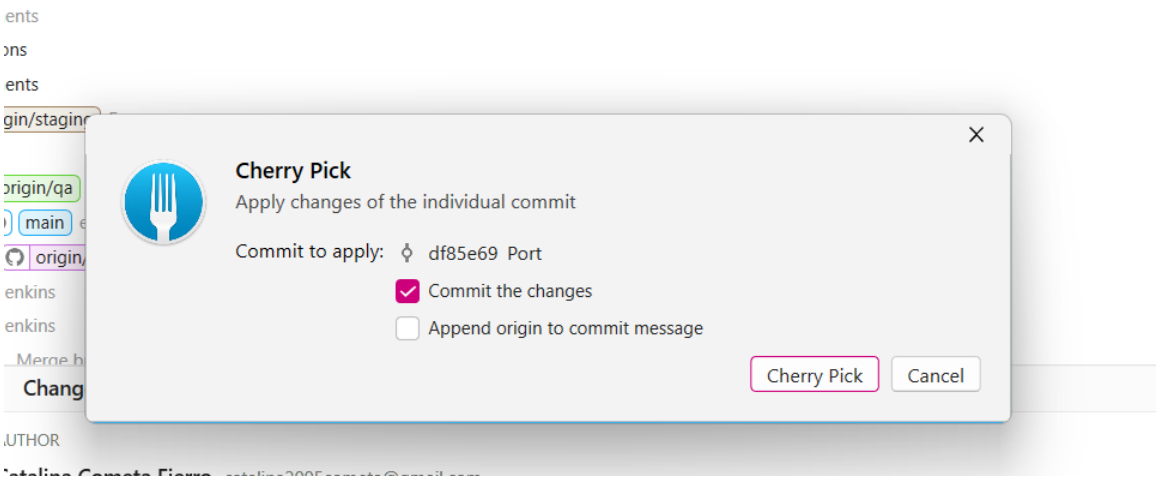
- Mayor control sobre los cambios que se integran entre ambientes.
- Reducción de riesgos al evitar merges completos innecesarios.
- Mejor trazabilidad de los commits.
- Agilidad en la corrección de errores.
- Aprendizaje práctico del manejo avanzado de Git.

En conclusión, la aplicación del cherry-pick mediante la herramienta Fork fue una estrategia clave en el proyecto *Experiencias Significativas*, ya que permitió una integración selectiva de cambios, fortaleció el control de versiones y evidenció el manejo de escenarios reales propios de proyectos de desarrollo colaborativo.

Seleccionamos la rama a la que en la que queremos hacer el Cherry Pick



Seleccionamos y damos clic derecho al commit que queremos pasar a la rama seleccionada, damos clic donde dice **Cherry-pick Commit…**



Damos clic donde dice Cherry Pick y listo!

12. Comparación entre el trabajo con y sin ambientes

Trabajar con múltiples ambientes representa una diferencia significativa frente a un desarrollo realizado en una sola rama. En un escenario sin ambientes, todos los cambios se realizan directamente sobre una rama principal, lo que incrementa el riesgo de errores, conflictos y fallos en producción.

En contraste, el uso de ambientes como *dev*, *qa*, *staging* y *main* permite separar claramente cada etapa del desarrollo. En *dev* se realizan pruebas iniciales, en *qa* se valida la funcionalidad, en *staging* se simula producción y en *main* se publica la versión final estable.

En el proyecto *Experiencias Significativas*, esta separación permitió detectar errores en etapas tempranas, evitar que código inestable llegara a producción y mantener un historial claro de cambios. Además, facilitó el trabajo colaborativo y permitió que varios desarrolladores trabajaran en paralelo sin interferencias.

13. Aprendizajes obtenidos del uso de ambientes

El uso de ambientes permitió adquirir aprendizajes clave tanto técnicos como organizativos. Se fortaleció el conocimiento en Git, la resolución de conflictos, el manejo de cherry-picks y el trabajo colaborativo.

Además, se desarrolló una mayor disciplina en el uso del control de versiones, entendiendo la importancia de seguir flujos establecidos y documentar cada cambio realizado.

Estos aprendizajes resultan fundamentales para futuros proyectos académicos y profesionales.

14. Conclusiones

- El manejo de ambientes permitió mantener un flujo organizado.
- El frontend fue el repositorio con implementación completa.
- Backend y móvil aplicaron el flujo parcialmente.
- El esquema garantiza calidad, orden y trazabilidad.