Choco-hub-2

- **Grupo 3**
- **Curso escolar: 2024/2025**
- **Asignatura: Evolución y gestión de la configuración**

Miembros del equipo

Miembro del equipo	Implicación[1-10]
Baquero Rodríguez, José María	10
Castelló Sánchez, Fernando	10
Gómez Marín, Jaime	10
Neria Acal, Ángel	10
Santos Dominguez, Pedro Pablo	10
Vélez López, Manuel	10

Enlaces de interés:

- **Repositorio de código**
- **Sistema desplegado**

Documento del Proyecto

Indicadores del proyecto

Miembro del equipo	Horas	Commits	LoC	Test	Issues	Work Item
Baquero Rodríguez, José María	90	17	926	14	9	Download In Different Formats, Fake Nodo
Vélez López, Manuel	100	25	1206	10	10	Register Validation
Santos Dominguez, Pedro Pablo	80	34	1279	16	8	Remember password
Castelló Sánchez, Fernando	70	14	948	18	8	Advanced Filtering
Neria Acal, Ángel	100	21	1052	14	10	Download All Datasets, Create Communities
Gómez Marín, Jaime	65	14	827	8	7	Create Communities, Remember password
TOTAL	505	124	6238	80	48	

Integración con otros equipos

- Chocohub1: Se integra con este equipo para tener un repositorio más completo, contando con más workitems, más workflows, etc. Hemos implementado algunos WIs que tienen interacciones y dependencias entre ellos, pudiendo sacar más partido a lo aprendido en la asignatura. Se ha integrado cada repositorio en el repositorio contrario, usando para ello una branch específica en nuestro caso, además de añadir posteriormente un workflow para automatizar la aplicación de los cambios de un repositorio en el otro.

Resumen ejecutivo

Durante el desarrollo de este proyecto, el objetivo principal fue añadir nuevas funcionalidades y mejorar las capacidades existentes del sistema UVLHub, un proyecto ya establecido. Las mejoras incluyeron la creación de comunidades, una funcionalidad diseñada para fomentar la interacción y colaboración entre los usuarios, permitiéndoles organizarse en grupos específicos y compartir información de manera más eficiente. Otra incorporación clave fue el desarrollo de FakeNodo, una API simulada que reemplaza las llamadas a Zenodo, ofreciendo un entorno controlado para pruebas y desarrollo sin depender de servicios externos. También se implementaron características centradas en la seguridad del usuario, como la verificación de autenticación a través de correo electrónico y la opción de cambio de contraseña, lo que no solo refuerza la protección de las cuentas, sino que mejora la experiencia general del usuario.

Por otro lado, se habilitaron opciones avanzadas para descargar todos los datasets existentes y modelos en diferentes formatos, lo que facilita el acceso a los recursos para los usuarios. Además, se desarrollaron filtros avanzados que mejoran significativamente la búsqueda y selección de datos y modelos dentro del sistema, optimizando el flujo de trabajo y el aprovechamiento de la plataforma por parte de los usuarios.

En paralelo a la implementación de estas funcionalidades, también se trabajó en mejorar la integración continua y el despliegue continuo del sistema. Para ello, se diseñaron y automatizaron diversos workflows. Por ejemplo, se creó un workflow y una plantilla específica para garantizar que los mensajes de commit cumplan con ciertos estándares, asegurando claridad y uniformidad en el historial de cambios. De manera similar, se implementó otro workflow para validar los nombres de las ramas, fomentando una estructura de trabajo más organizada. Además, se desarrolló un flujo automatizado para desplegar el sistema en Render. Este workflow incluye una funcionalidad que, en caso de que el despliegue falle, genera automáticamente una issue en el repositorio, permitiendo al equipo abordar los problemas rápidamente.

Otra mejora importante fue la integración de herramientas para garantizar la calidad del código. Por ejemplo, un workflow dedicado a formatear el código automáticamente utilizando Black asegura que el proyecto cumpla con los estándares de estilo. También se implementó un análisis de calidad del código mediante Codacy, que, si detecta una cobertura de pruebas inferior al 70%, genera automáticamente una issue para que el equipo pueda abordar la falta de cobertura. Además, se desarrolló un workflow que automatiza la creación de pull requests en el repositorio de otro grupo, lo que facilitó la integración con choco-hub-1 y fomentó la colaboración intergrupal.

Para garantizar que todas estas mejoras funcionen correctamente, se realizaron exhaustivas pruebas en varias categorías. Esto incluyó pruebas unitarias para validar la funcionalidad de módulos individuales, pruebas de integración para garantizar que las nuevas funcionalidades se comporten correctamente en conjunto, pruebas de carga para evaluar el rendimiento bajo diferentes niveles de uso y pruebas funcionales para asegurar que el sistema cumpla con los requisitos definidos.

En conclusión, este trabajo no solo ha ampliado significativamente las capacidades de UVLHub, sino que también ha mejorado su estabilidad, rendimiento y facilidad de uso. Las nuevas funcionalidades aportan un valor significativo al proyecto, mientras que las mejoras en los workflows aseguran que el desarrollo y el mantenimiento futuros sean más eficientes y sostenibles. Durante este proceso, se han adquirido importantes aprendizajes sobre automatización, pruebas y trabajo en equipo, enfrentando y superando retos técnicos y organizativos que enriquecen la experiencia profesional.

Descripción del sistema

El sistema nace a partir del código proporcionado en https://github.com/EGCETSII/uvlhub. Este código es la base de uvlhub.io, que es un repositorio de feature models en formato UVL integrados con Zenodo y flamapy siguiendo los principios de Open Science, y desarrollado por DiversoLab.

A partir de aquí el equipo decidió implementar 7 work items distintos, que fueron los siguientes:

- Download In Different Formats
- Download All Datasets
- Advanced Filtering
- Remember my password
- Register Validation
- Create Communities
- Fake Nodo

Con esto se buscaba mejorar y/o añadir funcionalidades al sistema, y se hicieron además test tanto unitarios como de integración y funcionales de cada uno de ellos para verificar su correcta funcionalidad.

A continuación presentamos una descripción de la implementación de cada workItem:

1. Download In Different Formats

La funcionalidad se integra al backend del sistema permitiendo la exportación de datos en múltiples formatos como .splx, .uvl, .cnf o .JSON. Esto se logra utilizando un controlador de descargas que invoca métodos específicos para transformar los modelos UVL al formato deseado.

Con esto conseguimos mejorar la accesibilidad de los datos al ofrecer compatibilidad con herramientas externas, facilitando el uso de los datos en diversas plataformas.

Esta incluye tests unitarios para verificar la correcta conversión entre formatos y tests funcionales para garantizar que los archivos exportados sean válidos y legibles.

2. Download All Datasets

Esta función permite la descarga masiva de los conjuntos de datos disponibles. Utiliza un sistema de compresión (ZIP) para empaquetar los archivos.

Esta funcionalidad aumenta la eficiencia al evitar descargas individuales repetitivas, ofreciendo a los usuarios un acceso más rápido a grandes volúmenes de datos.

Los tests validan que el archivo ZIP se genere correctamente y contenga todos los datos esperados, además de pruebas de integración para garantizar la compatibilidad con otros módulos.

3. Advanced Filtering

Se añade una capa avanzada de filtrado sobre la base de datos, utilizando parámetros dinámicos. Esto incluye filtros por validación de los archivos UVL de cada dataset, por número de autores (siendo este un select entre 3 opciones (además de la opción any, que viene por defecto y significa el no uso del filtro): proyecto individual (1 autor), colaboración pequeña (2 o 3 autores) o colaboración grande (4 o más autores)), para poder ordenar los datasets por número de descargas de más a menos y por visitas, también de más a menos. Se debe tener en cuenta que para estos dos últimos de cuenta de manera única, es decir, si un usuario ve o descarga un dataset más de una vez, solo se contará como uno a la hora de ordenarlos. Así evitamos el caso en el que un usuario haga un dataset y se lo descargue muchas veces para estar así en el top, entre otros.

Con esta implementación mejoramos la usabilidad de la página explore al permitir a los usuarios encontrar los datasets deseados más fácilmente.

Los tests unitarios verifican que los resultados del filtrado coincidan con los criterios definidos por el usuario. Además, hay pruebas de rendimiento para asegurar la eficiencia de las consultas.

4. Remember My Password

Implementado en el sistema, esta funcionalidad nos permite darle la opción al usuario de recordar su contraseña siguiendo unos pasos muy sencillos para no tener que escribirla cada vez que inicie sesión. Con esto facilita la experiencia del usuario al mantenerlo autenticado sin necesidad de ingresar repetidamente las credenciales.

Con los tests comprobamos su correcta funcionalidad además de poder ver la persistencia de la contraseña.

5. Register Validation

Esta funcionalidad hace que se envíe un correo de confirmación al correo que se pone en el registro para poder confirmarlo.

Garantiza que los datos ingresados sean correctos y minimiza errores en el registro de usuarios.

Incluye tests unitarios para validar cada campo de entrada y tests de integración para asegurar que los errores se muestren correctamente en la interfaz del usuario.

6. Create Communities

Añade una nueva entidad al modelo de datos para gestionar comunidades de usuarios. Estas comunidades son muy útiles ya que nos abre la puerta a distintas posibilidades: se pueden crear, editar y eliminar, y además como usuario externo puedes tanto entrar como salir de una, y compartir datasets y sacar uno compartido.

Este work item es muy útil porque promueve la colaboración entre usuarios al darles la posibilidad de colaborar entre ellos.

Las pruebas abarcan desde la creación y eliminación de comunidades hasta la correcta asignación de permisos a los usuarios.

7. Fake Nodo

Implementa un nodo simulado para pruebas. Este nodo emula la funcionalidad de un servidor real, permitiendo probar interacciones sin necesidad de una infraestructura completa.

Facilita el desarrollo y las pruebas al ofrecer un entorno controlado para validar funcionalidades.

Se diseñaron tests de integración para verificar que las interacciones con el nodo simulado sean equivalentes a las de un nodo real.

Por otra parte, también se ha trabajado en otros ámbitos del proyecto como en los workflows, con el objetivo de mejorar:

- Eficiencia: Optimizar el uso de recursos y reducir el tiempo necesario para completar tareas.
- Estandarización: Asegurar que los procesos se realicen de manera consistente y conforme a las mejores prácticas.
- Visibilidad: Proporcionar una representación clara de cómo fluyen las tareas y quién es responsable de cada etapa.

- Mejora continua: Facilitar la identificación de cuellos de botella y áreas de mejora en los procesos.

Esto es esencial para la organización y optimización de procesos dentro de una organización, ayudando a alcanzar objetivos de manera más efectiva.

Nuestra aportación en cuanto a nuevos workflows implementados es la siguiente:

- check_branch_name.yml: Comprueba si el nombre de la rama coincide con el patrón deseado.
- commit_message.yml: Comprueba la sintaxis de los commits.
- create_pr_in_choco1.yml: Cuando se realiza un cambio en nuestro repositorio se abre una pull request en el repositorio del otro grupo, para que puedan comprobar nuestros cambios y añadirlos.

Además, este proyecto incluye por supuesto documentación para la gestión de commits, issues, branches y hooks, para entender primero como está este realizado estructuralmente en GitHub y también para informar a aquellos que quieran hacer nuevas aportaciones sobre como hacerlas correctamente.

Como resultado, obtenemos una aplicación mejorada que no solo amplía su funcionalidad y usabilidad, sino que también refuerza la experiencia del usuario al integrar características avanzadas y un entorno optimizado. Estas mejoras, junto con un enfoque centrado en la calidad, pruebas exhaustivas y una gestión eficiente, consolidan el sistema como una herramienta más robusta, accesible y colaborativa para la comunidad.

Visión global del proceso de desarrollo

Debe dar una visión general del proceso que ha seguido enlazándolo con las herramientas que ha utilizado. Ponga un ejemplo de un cambio que se proponga al sistema y cómo abordaría todo el ciclo hasta tener ese cambio en producción. Los detalles de cómo hacer el cambio vendrán en el apartado correspondiente.

Proceso de desarrollo utilizando Git y GitHub

El equipo ha utilizado herramientas de **control de versiones con Git** y un repositorio alojado en **GitHub,** lo que ha facilitado la colaboración eficiente y ha reducido significativamente los problemas al integrar cambios realizados por diferentes integrantes del equipo. A continuación, se detalla cómo hemos estructurado nuestro flujo de trabajo para garantizar un desarrollo ordenado y coordinado.

Flujo de trabajo para el desarrollo del proyecto

Escenario 1: Crear las funcionalidades básicas de un **WorkItem**

1. **Abrir una issue en el repositorio**:

Cada integrante debe abrir una **issue** en GitHub para describir la tarea a realizar. Esto se hace siguiendo un **patrón predefinido**:

- Si la tarea consiste en implementar un nuevo **WorkItem**, el nombre de la **issue** debe corresponder al nombre del **WorkItem** en español.

- La descripción de la **issue** debe incluir detalles sobre la funcionalidad requerida, los pasos para implementarla y cualquier otro dato relevante.

2. **Crear una rama específica**:

- Para trabajar en la funcionalidad, se crea una nueva rama en el repositorio con el formato:

`featureNombreDeLaFuncionalidad`.

- Este enfoque permite mantener el código organizado y facilita la identificación de las ramas relacionadas con tareas específicas.

3. **Implementar la funcionalidad**:

- Se desarrollan los cambios necesarios en el código fuente para cumplir con los requisitos de la **issue**.

- Es fundamental realizar **commits atómicos**, es decir, cada commit debe reflejar un cambio específico y autónomo. Esto facilita el seguimiento de los cambios y la resolución de problemas en caso de errores.

4. **Registrar problemas adicionales**:

- Durante el desarrollo, si se detectan **bugs** o se identifican **funcionalidades faltantes**, se debe abrir una nueva **issue** para informar al equipo. Esto asegura la documentación y seguimiento de los problemas.

5. **Subir los cambios y crear una Pull Request (PR)**:

- Los cambios se suben al repositorio remoto mediante comandos de Git.

- Se crea una **Pull Request** para solicitar la revisión de los cambios antes de fusionarlos con la rama principal (`main`).

- Al menos uno o dos integrantes del equipo deben revisar la PR, proporcionar comentarios si es necesario y, tras la aprobación, realizar el **merge** con `main`.

Escenario 2: Crear los tests del **WorkItem**

1. **Abrir una nueva issue**:

- Se crea una nueva **issue** en el repositorio, describiendo los **tipos de tests** a realizar para la funcionalidad implementada. Los tests pueden incluir:

  - **Tests unitarios**: Validan que las funciones individuales se comporten según lo esperado.

  - **Tests de integración**: Verifican que los módulos o componentes interactúen correctamente.

  - **Tests de carga**: Evalúan el rendimiento del sistema bajo condiciones de alta demanda.

  - **Tests de interfaz**: Aseguran que la interfaz de usuario sea funcional y cumpla con los requisitos.

2. **Crear una rama específica para los tests**:

- Se crea una nueva rama en el repositorio con el formato:

  `testNombreDeLaFuncionalidad`.

3. **Implementar los tests**:

- Se desarrollan los tests correspondientes para validar el comportamiento de la funcionalidad.

- Durante el testeo, si se detectan errores, se debe abrir una nueva **issue** para documentarlos y asignarlos a un integrante del equipo.

4. **Subir los cambios y realizar una Pull Request**:

- Los tests desarrollados se suben al repositorio remoto.

- Se crea una Pull Request para fusionar los tests en la rama principal (`main`).

- Al igual que en el desarrollo de funcionalidades, la PR debe ser revisada y aprobada antes de realizar el merge.

Escenario 3: Solucionar errores o implementar funcionalidades faltantes

1. **Identificar y documentar el problema**:

- Los errores encontrados durante el desarrollo o el testeo se registran como nuevas **issues** en el repositorio.

- La descripción de la **issue** debe ser clara y detallada, especificando el problema, su impacto y cualquier información relevante para solucionarlo.

2. **Asignación de issues**:

- Los integrantes del equipo revisan las **issues** abiertas y se asignan aquellas que pueden resolver. Esto asegura que cada problema tenga un responsable.

3. **Crear una rama para la solución**:

- Se crea una nueva rama con un nombre que indique la naturaleza del cambio:

  - Para errores (bugs): `bugfixNombreDelBug`.

  - Para funcionalidades faltantes: `featureNombreDeLaFuncionalidad`.

4. **Realizar los cambios necesarios**:

- Se desarrollan las soluciones correspondientes al problema identificado en la **issue**.

- Se realizan **commits atómicos** que documenten claramente cada cambio realizado.

5. **Subir los cambios y realizar una Pull Request**:

- Se suben los cambios al repositorio remoto.

- Se crea una Pull Request para fusionar los cambios en la rama principal (`main`).

- La PR debe ser revisada, aprobada y fusionada siguiendo el mismo procedimiento descrito anteriormente.

Estados de las Issues

-**ToDo:** La **issue** está abierta y pendiente de resolución.

- **In Progress**: La **issue** está siendo trabajada por un integrante del equipo.

- **Completed**: La **issue** se marca como completada cuando los cambios han sido aprobados y fusionados en la rama `main`.

Beneficios del flujo de trabajo

Este enfoque estructurado asegura:

- **Colaboración eficiente y coordinada entre los miembros del equipo**: Al dividir las tareas mediante *issues* y ramas específicas, cada integrante sabe exactamente qué debe hacer y cómo encaja su trabajo en el contexto general del proyecto. Esto reduce el riesgo de solapamiento de tareas y evita conflictos innecesarios en el código.
- **Documentación clara y centralizada de las tareas y problemas**: Las *issues* actúan como un registro detallado de todo lo que se ha trabajado, incluyendo qué problemas surgieron y cómo se resolvieron. Esto no solo ayuda a mantener el orden durante el desarrollo, sino que también proporciona una valiosa referencia para el futuro.
- **Gestión ordenada de los cambios y versiones del código**: El uso de ramas específicas para cada funcionalidad, test o corrección de errores garantiza que los cambios estén aislados y organizados. Esto facilita la revisión de código y la integración de nuevas funcionalidades sin afectar la estabilidad del sistema.
- **Seguimiento transparente del progreso del proyecto**: Al actualizar el estado de las *issues* y gestionar las Pull Requests, todos los integrantes pueden visualizar el avance del proyecto en tiempo real. Esto fomenta una mayor coordinación, permite identificar cuellos de botella y asegura que las tareas se completen según lo planeado.
- **Fomento de buenas prácticas de desarrollo**: Al exigir commits atómicos, revisiones de código y pruebas exhaustivas, este flujo de trabajo promueve

estándares altos de calidad en el desarrollo. Esto no solo beneficia al proyecto actual, sino que también mejora las habilidades de los desarrolladores involucrados.
- **Reducción de errores y problemas inesperados**: Al integrar pruebas desde el inicio y abordar errores de manera estructurada, se minimizan los problemas que podrían surgir durante la implementación. Esto asegura un producto final más robusto y confiable.

Entorno de desarrollo

Debe explicar cuál es el entorno de desarrollo que ha usado, cuáles son las versiones usadas y qué pasos hay que seguir para instalar tanto su sistema como los subsistemas relacionados para hacer funcionar el sistema al completo. Si se han usado distintos entornos de desarrollo por parte de distintos miembros del grupo, también debe referenciarlo aquí.

En el desarrollo de nuestro proyecto se ha utilizado un conjunto de herramientas. A continuación, se describen con detalle las herramientas empleadas, las versiones utilizadas, así como los pasos necesarios para instalar, configurar y desplegar el sistema junto con los subsistemas relacionados.

En primer lugar, el desarrollo del sistema se ha llevado a cabo en Ubuntu, un sistema con compatibilidad con herramientas como Docker, Python y Git permitiendo a todo el equipo mantener un entorno homogéneo.

Se ha utilizado Visual Studio Code como herramienta principal de edición de código junto con extensiones como Python, Docker, para gestionar contenedores directamente desde el editor, y GitHub, para integración con el repositorio y manejo de ramas y pull requests, lo que permitió la ejecución de las tareas y colaboración entre los miembros del equipo.

Para gestionar el código fuente y facilitar la colaboración, se utilizó Git junto con un repositorio alojado en GitHub, permitiendo asegurar la trazabilidad de los cambios y una buena integración continua.

También se han utilizado hooks de git para garantizar que los mensajes de commit sigan un formato predefinido y consistente, asegurando la claridad y trazabilidad de los cambios implementados. Para instalar los hook se siguen los siguientes pasos:

- Concede permisos de ejecución al script: **chmod +x setup-hooks.sh**
- Ejecuta el script de instalación: **./setup-hooks.sh**

Esto copia los hooks al directorio .git/hooks, automatizando las validaciones en cada commit

El lenguaje principal del sistema es Python, destacando el uso de las siguientes librerías:

- **Flask**: Framework ligero para el desarrollo del backend
- **Flask-RESTful:** Para la creación y gestión de endpoints de la API.
- **Flask-SQLAlchemy:** Para la integración y manipulación de bases de datos.

- **Flask-Migrate:** Para gestionar las migraciones de base de datos.
- **Flask-Cors:** Para manejar el intercambio de recursos entre dominios.

En cuanto a gestión de base de datos, mencionar que se ha utilizado MariaDB.

A continuación, se detalla un resumen de las versiones principales empleadas en el desarrollo:

- Python: 3.11.x
- Flask: 3.0.3
- SQLAlchemy: 2.0.31
- Docker: 24.x
- MariaDB: 10.11.x

El archivo requirements.txt incluye todas las dependencias necesarias para el correcto funcionamiento del sistema.

Además en el archivo .env se encuentran todas las configuraciones del entorno necesarias para ejecutar la aplicación.El contenido del mismo será el siguiente:

```
FLASK_APP_NAME="UVLHUB.IO(dev)"

FLASK_ENV=development

DOMAIN=localhost:5000

MARIADB_HOSTNAME=localhost

MARIADB_PORT=3306

MARIADB_DATABASE=uvlhubdb

MARIADB_TEST_DATABASE=uvlhubdb_test

MARIADB_USER=uvlhubdb_user

MARIADB_PASSWORD=uvlhubdb_password

MARIADB_ROOT_PASSWORD=uvlhubdb_root_password

WORKING_DIR=""
```

Ya conocidos los aspectos generales referentes al desarrollo y gestión de los elementos del sistema, los pasos para la instalación y configuración del mismo son los siguientes:

- Primero tener en cuenta los siguientes requisitos previos:
  - Tener instalado **Docker** y **Docker Compose.**
  - Contar con **Python 3.11** o superior.
  - Instalar **Git** para clonar el repositorio.
  - Configurar un entorno virtual con **venv** de nombre para evitar conflictos de dependencias.

- Comenzamos clonando el repositorio de GitHub: **https://github.com/ch0cohub/choco-hub-2.git**
- Instalar las dependencias: **pip install -r requirements.txt**
- Crear un archivo .**env** en la raíz del proyecto y copiar el contenido del ejemplo proporcionado anteriormente
- Construir y desplegar los contenedores para levantar tanto la aplicación Flask como la base de datos MariaDB, según lo especificado en el archivo docker-compose.yml: **docker-compose up --build**
- Ejecutar las migraciones de la base de datos: **flask db upgrade**
- Finalmente, iniciar el servidor Flask: **flask run**
- **El sistema estará accesible en http://localhost:5000**

Finalmente encontramos los siguientes subsistemas:

- **Render**: Permite el despliegue en la nube. Se incluye un pipeline de CI/CD para desplegar automáticamente los cambios realizados en el repositorio.
- **Files.io**: Se utiliza como servicio de almacenamiento para la base de datos y archivos relacionados con la aplicación.

Ejercicio de Propuesta de Cambio: Añadir un Nuevo Filtro Avanzado

Paso 1: Crear una Issue para el cambio

1. Accede al repositorio del proyecto en GitHub.
2. Ve a la sección **Issues** y selecciona "New Issue".
3. Escribe un título descriptivo, como: *"Añadir filtro avanzado para búsqueda por tamaño de dataset"*.
4. Describe los detalles de la funcionalidad que deseas implementar, incluyendo el objetivo del cambio y cómo beneficiará a los usuarios.
5. Asigna la issue a un desarrollador y selecciona las etiquetas correspondientes, como "enhancement" o "feature".

Paso 2: Crear una Rama para el Cambio

1. En la terminal o herramienta de control de versiones, asegúrate de estar en la rama principal:
   - `git checkout main`
   - `git pull origin main`
2. Crea una nueva rama para el cambio:
   - `git checkout -b featureAddSizeFilter`
3. Implementa el cambio en el código. Por ejemplo, añade el filtro avanzado en el archivo correspondiente, asegurándote de seguir el estilo y la estructura del proyecto.

Paso 3: Commit del Cambio

1. Una vez realizados los cambios, añade los archivos modificados al índice:

    - `git add .`
2. Realiza un commit con un mensaje descriptivo:
    - `git commit -m "feat: Añadir filtro de tamaño, se han añadido los cambios necesarios para implementar la funcionalidad requerida"`

Paso 4: Crear una Pull Request (PR) para la rama de feature

1. Sube la rama de feature al repositorio remoto:
    - `git push origin featureAddSizeFilter`
2. En GitHub, selecciona la rama principal y haz clic en "Compare & pull request".
3. Completa los detalles de la PR, mencionando que incluye la funcionalidad correspondientes y asignándola a un miembro del equipo.
4. Dicho miembro del equipo debe comprobar la PR y mergearla en caso de que no haya problemas.
5. Cerrar Issue asociada al cambio solicitado.

Paso 5: Crear y Asociar una Issue para los Tests

1. Crea una nueva issue con el título: *"testing *tipoDeTest* añadir filtro avanzado para búsqueda por tamaño de dataset"*.
2. Detalla en la descripción que se desarrollarán las pruebas necesarias para la funcionalidad.

Paso 6: Crear Rama para los Tests

1. Crea una nueva rama para las pruebas:
    - `git checkout main`
    - `git pull origin main`
    - `git checkout -b testAddSizeFilter`
2. Escribe las pruebas necesarias en el archivo correspondiente.

Paso 7: Commit de los Tests

1. Añade y realiza el commit de los tests:
    - `git add .`
    - `git commit -m "test: Añadir filtro por tamaño, se han añadido los tests unitarios de esta funcionalidad"`

Paso 8: Crear una Pull Request (PR) para la rama de test

1. Sube la rama de feature al repositorio remoto:
    - `git push origin testAddSizeFilter`
2. En GitHub, selecciona la rama principal y haz clic en "Compare & pull request".
3. Completa los detalles de la PR, mencionando que incluye los tests correspondientes y asignándola a un miembro del equipo.
4. Dicho miembro del equipo debe comprobar la PR y mergearla en caso de que no haya problemas.

5. Cerrar Issue asociada al test solicitado.

Conclusiones y trabajo futuro

El desarrollo llevado a cabo en este proyecto ha logrado fortalecer de manera notable las funcionalidades y capacidades de UVLHub, dotando a los usuarios de herramientas más completas y mejorando significativamente la experiencia general de uso. Asimismo, se ha dado prioridad a la calidad del código, acompañado de pruebas exhaustivas y la automatización mediante workflows, lo que establece una base sólida para garantizar la sostenibilidad y el progreso continuo del sistema.

Sin embargo, uno de los mayores aprendizajes de este proyecto ha sido la importancia de la automatización en el desarrollo de software. Si bien se han logrado avances significativos, como la creación de workflows para despliegue, formateo de código y análisis de calidad, el tiempo disponible no permitió explorar todas las posibilidades de automatización. Este aspecto representa un área clave para el desarrollo futuro del sistema.

En futuras iteraciones del proyecto, se recomienda profundizar en la automatización mediante workflows. Entre las posibles mejoras se encuentran:

- **Ampliar la cobertura de los workflows existentes:** Por ejemplo, automatizar más procesos relacionados con la gestión de issues, revisiones de código y despliegues en diferentes entornos.
- **Explorar herramientas avanzadas para la integración continua:** Investigar nuevas plataformas o funcionalidades que complementen Codacy y Render, para optimizar la eficiencia y el control del ciclo de desarrollo.
- **Incorporar validaciones más completas en los workflows:** Como análisis de seguridad del código, validaciones específicas para configuraciones y mejoras en la monitorización del estado del sistema en tiempo real.
- **Automatización en la integración de terceros:** Mejorar y expandir los workflows relacionados con la colaboración entre repositorios, facilitando la sincronización y el trabajo conjunto entre equipos.

Finalmente, aunque el enfoque principal del proyecto fue consolidar las funcionalidades existentes, en el futuro sería interesante añadir capacidades más avanzadas en términos de inteligencia artificial o análisis de datos para UVLHub, como modelos preentrenados que ayuden a los usuarios en tareas específicas. Esto, junto con un perfeccionamiento continuo de la automatización, garantizará que el proyecto evolucione para cubrir nuevas necesidades y desafíos.