Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Automatización de la Corrección de Prácticas Académicas mediante GitHub

Autor: Juan Miguel Rodríguez Santos
Tutor: Fernando Pérez Costoya

Madrid, Junio - 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Automatización de la Corrección de Prácticas Académicas mediante GitHub

Junio - 2025

Autor: Juan Miguel Rodríguez Santos
Tutor: Fernando Pérez Costoya
Departamento de Arquitectura y Tecnología de Sistemas Informáticos (DATSI)
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

Palabras Clave: Corrección automática de prácticas, GitHub, CI/CD en la universidad, DevOps en la enseñanza, Retroalimentación inmediata

En el contexto de la educación universitaria, corregir manualmente prácticas de programación supone un reto importante, especialmente cuando se trata de asignaturas con muchos estudiantes. La ausencia de un sistema automatizado y unificado para gestionar estas prácticas hace que los profesores tengan una gran carga de trabajo, retrasando considerablemente la retroalimentación que reciben los estudiantes. Además, cuando se utilizan plataformas distintas y desconectadas entre sí para gestionar cada entrega, se hace difícil seguir adecuadamente el progreso de los alumnos y promover buenas prácticas de desarrollo.

Este Trabajo de Fin de Grado plantea el diseño y desarrollo de un sistema automatizado para la corrección de prácticas académicas, utilizando GitHub Actions y GitHub Classroom, herramientas ampliamente empleadas tanto en contextos educativos como profesionales. La idea principal es proporcionar un entorno que permita realizar evaluaciones continuas y escalables de las prácticas, aprovechando las ventajas del ecosistema GitHub para introducir a los estudiantes en metodologías de trabajo modernas como las prácticas DevOps.

El sistema planteado facilita la gestión centralizada de prácticas, soporta múltiples lenguajes (se mostrará principalmente Java, por ser el lenguaje vehicular de la Escuela) y contempla desde prácticas sencillas hasta proyectos complejos que involucren arquitecturas completas. Además, permite incluir pruebas automatizadas personalizadas y scripts externos para abordar casos específicos. El flujo de trabajo está diseñado para que cada vez que un estudiante realice una entrega, se ejecuten automáticamente las pruebas definidas, proporcionando un feedback inmediato y detallado.

Para evaluar la viabilidad del sistema propuesto, se ha llevado a cabo una investigación exhaustiva del estado del arte, comparando plataformas existentes como Deliverit, GitLab CI, Jenkins, Gradescope y Codeboard. Los resultados muestran que GitHub Actions, combinado con GitHub Classroom, ofrece una integración más natural con el entorno educativo universitario, fomentando tanto el aprendizaje continuo como el acercamiento a herramientas profesionales del ámbito industrial.

Abstract

Keywords: Automated grading of assignments, GitHub, University CI/CD, DevOps in education, Immediate feedback

Manually grading programming assignments is a major challenge in the context of university education, especially in courses with a large student body. Instructors are overwhelmed and feedback to students is greatly delayed due to the absence of an automated and centralised system to handle these assignments. Furthermore, it is challenging to efficiently monitor student progress and encourage best practices for development when distinct and disjointed platforms are used for every submission.

This Bachelor's Thesis proposes the design and implementation of an automated grading system for academic programming assignments, using GitHub Actions and GitHub Classroom, tools widely adopted in both educational and professional environments. The main goal is to provide an environment that enables continuous and scalable evaluation of programming assignments, leveraging the GitHub ecosystem to introduce students to modern workflows and DevOps methodologies.

The proposed system facilitates centralized management of assignments, supports multiple programming languages (primarily Java, as it is the vehicular language of the school), and covers simple tasks as well as complex projects involving complete architectures. Additionally, it allows for the inclusion of customized automated tests and external scripts to address specific cases. The workflow is designed so that every time a student submits the code, the defined tests are automatically executed, providing immediate and detailed feedback.

To evaluate the feasibility of the proposed system, an exhaustive state-of-the-art analysis was carried out, comparing existing platforms such as Deliverit, GitLab CI, Jenkins, Gradescope, and Codeboard. The results indicate that GitHub Actions, combined with GitHub Classroom, provides a more natural integration within the university educational environment, fostering both continuous learning and the adoption of professional tools from the industrial domain.

Agradecimientos

A mi familia, por estar siempre ahí, por creer en mí, por apoyarme sin condiciones y por darme la oportunidad de llegar hasta aquí. Sin vosotros no habría sido posible.

A mi tutor, por confiar en la idea desde el principio y ayudarme a llevarla adelante.

A todas esas personas con las que he coincidido en este camino, gracias por hacerlo más fácil.

Índice general

Índice de Figuras

Índice de Tablas

Índice de Listings

Capítulo 1

Introducción

Este capítulo introduce el proyecto tratando las razones que lo impulsan desde un punto de vista personal, pedagógico y profesional. Asimismo, sitúa el trabajo dentro del contexto institucional y tecnológico en el que se desarrolla, explicando cuáles son sus objetivos principales y secundarios, para finalmente ofrecer una breve visión de la estructura general del documento.

1.1. Motivación del proyecto

Las razones que sustentan este proyecto pueden agruparse en tres ejes complementarios: personal, pedagógico y profesional. En conjunto, justifican la oportunidad de la solución propuesta.

1.1.1. Motivación personal

Durante los primeros cursos del Grado en Ingeniería Informática, uno de los desafíos recurrentes fue coordinar el trabajo de programación en equipo sin un flujo estructurado. El intercambio de archivos por correo electrónico o mensajería instantánea – con la consabida frase «en mi máquina funciona» – generaba conflictos de versiones y prolongaba tareas simples.

La incorporación de Git introdujo control de versiones y trazabilidad, pero aún quedaban fricciones operativas. Fue con la adopción de GitHub cuando la colaboración distribuida se consolidó realmente, al centralizar los repositorios en la nube y facilitar la revisión de cambios. El progreso culminó con GitHub Actions, que permitió compilar y ejecutar el código automáticamente en un entorno controlado, eliminando discrepancias locales y elevando la calidad del trabajo grupal.

Esta trayectoria personal evidenció el potencial de aplicar la automatización, no solo a la construcción del software, sino también a la evaluación académica de prácticas. La primera prueba experimental, realizada en la asignatura de Administración de Sistemas Informáticos, demostró que un pipeline sencillo era capaz de validar entregas de los estudiantes de forma inmediata. Aquella experiencia fue el germen del presente Trabajo de Fin de Grado: extender la automatización al proceso de corrección para beneficio de toda la comunidad educativa.

1.1.2. Motivación pedagógica

La evaluación de prácticas en asignaturas técnicas suele implicar decenas, cuando no cientos, de entregas que el profesor debe compilar, ejecutar y revisar de forma manual. Este proceso, por naturaleza lento, demora la retroalimentación y reduce su valor formativo [1]: los estudiantes suelen avanzar sobre supuestos incorrectos y olvidan detalles de la práctica cuando reciben finalmente los comentarios.

Al incorporar un sistema de corrección automatizada basado en GitHub Classroom y GitHub Actions, cada entrega desencadena un flujo que compila y ejecuta el código en un entorno controlado y devuelve los resultados en cuestión de minutos. Este feedback inmediato, reconocido en la literatura pedagógica como un factor clave de aprendizaje [2], permite que el alumno detecte y corrija errores antes de que se afiancen, refuerce conceptos fundamentales y adopte un enfoque iterativo similar al de las metodologías ágiles [3].

La naturaleza cuantificable de las pruebas automatizadas favorece, además, la gamificación del aprendizaje [4]. Indicadores visuales simples (por ejemplo, el estado de comprobaciones en GitHub) estimulan la motivación intrínseca del estudiante y fomentan más iteraciones, lo que se traduce en una mejora progresiva de la calidad de sus soluciones.

Desde la perspectiva docente, la automatización libera tiempo que antes se dedicaba a tareas repetitivas y posibilita concentrarse en aspectos cualitativos: estilo, diseño o discusión conceptual durante tutorías. Asimismo, disponer de resultados estructurados facilita detectar patrones de error comunes en el grupo [5] y ajustar de forma proactiva la docencia, favoreciendo una enseñanza más adaptativa.

1.1.3. Motivación profesional y pragmática

La industria del software exige competencias en control de versiones, integración continua y despliegue continuo [6]. Introducir GitHub Classroom y GitHub Actions en la formación universitaria acorta la brecha entre el aula y el entorno laboral, dotando a los estudiantes de habilidades técnicas y metodológicas altamente demandadas.

La familiaridad temprana con flujos de trabajo DevOps no solo mejora la empleabilidad, sino que también desarrolla competencias transversales como comunicación técnica, gestión de tiempos y trabajo colaborativo. Casos de éxito en universidades internacionales, como PUC Minas en Brasil [7], o universidades españolas como es la Universidad de Almería [8], muestran que la adopción de estos entornos incrementa la preparación profesional del alumnado y alinea el currículo con los estándares del sector.

Desde la perspectiva docente, el corrector automático actúa como asistente de evaluación funcional, asegurando objetividad y consistencia [9], [10], mientras el profesor mantiene la última palabra sobre aspectos cualitativos. Así, la solución propuesta conjuga eficiencia operativa con rigor académico, posicionándose como una herramienta estratégica tanto para la enseñanza como para la proyección profesional de los estudiantes.

1.2. Contexto del proyecto

Corregir manualmente decenas o cientos de prácticas en universidades con gran número de estudiantes como, por ejemplo, la Universidad Politécnica de Madrid, consume tiempo y retrasa la retroalimentación. Ante este desafío, GitHub Classroom se ha consolidado como plataforma educativa [11] que facilita la distribución y recogida de tareas; en 2019, más de 20.000 profesores la utilizaban a escala global [11]. No obstante, su sistema de autograding nativo se limita a pruebas unitarias básicas.

El verdadero potencial emerge al integrarlo con GitHub Actions, servicio CI/CD que ejecuta flujos de trabajo en cada commit del estudiante. Esto permite compilar, probar y generar informes automáticamente, ofreciendo retroalimentación inmediata, beneficio documentado en múltiples experiencias universitarias [5], [7], [12], [8].

Pese a la existencia de otras herramientas (Deliverit, GitLab CI, Jenkins, Gradescope), ninguna aúna con la misma facilidad la gestión de repositorios académicos y la automatización Integración Continua (*Continuous Integration*); práctica que ejecuta pruebas y análisis automáticamente en cada *commit* para detectar errores de forma temprana, eje central de los flujos descritos en la memoria. (CI) en la nube. Por ello, este Trabajo de Fin de Grado aborda el diseño de un corrector de prácticas basado en GitHub Actions que pueda usarse en diferentes asignaturas de programación, con el foco inicial en Java, pero extensible a otros lenguajes (C, Python, etc.).

1.3. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es diseñar un sistema automatizado de corrección de prácticas académicas basado en GitHub Actions, integrado en el ecosistema de GitHub Classroom. Se propone una solución que permita a los docentes corregir de manera eficiente las prácticas de programación, brindando a la vez beneficios formativos a los estudiantes mediante retroalimentación temprana y la exposición a herramientas de desarrollo profesionales. El enfoque de este trabajo será eminentemente práctico: se describirá cómo desplegar un sistema de integración continua para la corrección automática en el entorno de GitHub, considerando los requisitos específicos del ámbito académico.

De este objetivo principal se derivan los siguientes objetivos específicos, cuya consecución se verificará de manera clara y objetiva:

- Diseñar un esquema detallado para la gestión de prácticas académicas utilizando GitHub Actions, especificando claramente la organización de repositorios, configuración de tareas y la interacción con los estudiantes.
- Desarrollar e implementar un sistema de corrección automatizado basado en GitHub Actions que permita evaluar automáticamente las entregas realizadas por los alumnos tras cada actualización del código en el repositorio correspondiente. Esta evaluación automática se centrará exclusivamente en la ejecución funcional del código, sin sustituir en ningún momento la valoración cualitativa o manual realizada por el profesorado.
- Documentar de manera completa y rigurosa el diseño, implementación y configuración del sistema propuesto, facilitando así su comprensión, mantenimiento y replicación en futuras aplicaciones.

El sistema se centra en la validación funcional automática (compilación y ejecución de pruebas). Aspectos cualitativos de estilo, diseño o documentación quedan fuera del alcance y serán revisados manualmente por el profesorado.

1.4. Estructura del Documento

Este Trabajo de Fin de Grado se articula en cinco capítulos principales, complementados con bibliografía y anexos, para guiar al lector de la motivación inicial hasta las conclusiones y el trabajo futuro:

- **Capítulo 1: Introducción.** Presenta el porqué del proyecto: las motivaciones personal, pedagógica y profesional; el contexto académico y tecnológico; los objetivos generales y específicos; y una breve panorámica del resto del documento.
- **Capítulo 2: Trabajo relacionado y Estado del Arte.** Revisa la literatura y las soluciones existentes. Comienza describiendo la metodología de búsqueda, expone la necesidad de la corrección automática y repasa los fundamentos técnicos. Después analiza el uso de Git y GitHub en docencia, detalla beneficios y limitaciones, compara plataformas alternativa y cierra con una comparativa crítica que justifica la elección de la propuesta.
- **Capítulo 3: Diseño y Desarrollo.** Expone la arquitectura del sistema y sus componentes, el flujo de eventos y la gestión de elementos. Describe el diseño de los repositorios plantilla, el catálogo de workflows (desde ejercicios básicos hasta proyectos completos), la gestión de errores y la estrategia de escalabilidad y despliegue. Incluye la lógica de presentación de resultados tanto para estudiantes como para docentes.
- **Capítulo 4: Impacto del trabajo.** Analiza las repercusiones educativas y técnicas del sistema, así como su alineación con los Objetivos de Desarrollo Sostenible (ODS). Examina cómo la automatización influye en la carga docente, el aprendizaje del alumnado y la adopción de prácticas DevOps en la universidad.
- **Capítulo 5: Resultados y conclusiones.** Recoge los resultados de los objetivos marcados para el trabajo, los resultados esperados al aplicar el sistema, evalúa sus limitaciones, y extrae conclusiones personales. Se plantean líneas de trabajo futuro, como la incorporación de análisis estático de calidad o la adaptación a casos aún más complejos.
- **Anexos y Bibliografía.** Se aportan materiales adicionales (configuraciones, capturas, scripts) para profundizar aún más y el listado completo de las referencias utilizadas. Además, se incluye un listado de acrónimos y un glosario con los términos más técnicos.

Capítulo 2

Trabajo relacionado y Estado del Arte

Este capítulo detalla, por orden: la metodología de búsqueda de la literatura, la motivación de la corrección automática, los fundamentos técnicos detrás de esta propuesta, su aplicación educativa documentada, otras plataformas y enfoques comparables, una comparativa crítica basada en cinco criterios y las brechas que dan pie a la propuesta del capítulo 3.

2.1. Metodología de investigación

La revisión cubre literatura publicada entre enero de 2010 y abril de 2025. Las búsquedas se realizaron en las bases de datos ACM Digital Library, IEEE Xplore, Scopus, SpringerLink y Google Scholar, complementadas con documentación oficial y reportes de GitHub Education y GitHub Actions. Se utilizaron combinaciones de términos relativos a programación académica, autograding y CI/CD (por ejemplo, “automated grading”, “GitHub Classroom”, “GitHub Actions”, “DevOps education”). Los trabajos identificados proporcionan la base de evidencias que sustenta el análisis desarrollado en las secciones posteriores.

2.2. Necesidad de la corrección automática

La expansión de la matrícula en las titulaciones de Ingeniería Informática ha llevado a cursos con más de 150 estudiantes por grupo, cifra que compromete la capacidad docente para ofrecer una evaluación oportuna. Diversos estudios reportan que, en asignaturas con este volumen, el plazo medio de entrega de feedback ronda las 2-3 semanas cuando la corrección es íntegramente manual [1] [13]. Este retraso no solo ralentiza el ciclo de mejora del alumno, sino que también degrada el valor formativo de la práctica: cuanto mayor es el intervalo entre la entrega y la retroalimentación, menor es la retención de lo aprendido.

Meta-análisis recientes en educación superior confirman que la inmediatez del feedback es uno de los factores con mayor efecto sobre el rendimiento académico (g - 0,70) [14]. En programación, donde la comprensión se consolida por iteración rápida, el desfase temporal bloquea el aprendizaje activo y favorece la adopción de estrategias

superficiales (entregas de última hora, pruebas mínimas). El problema se agrava por la heterogeneidad de los entornos locales: discrepancias de versión, dependencias no declaradas y configuraciones dispares generan discusiones improductivas sobre por qué el código «funciona en mi equipo» pero falla en el del profesor.

La carga de trabajo asociada a compilar, ejecutar y revisar centenares de proyectos alcanza, según Bovel et al. [1], entre 8 y 12 horas de corrección por práctica en cursos de 180 alumnos. Esta presión operativa suele traducirse en feedback simplificado [9] (comentarios genéricos o puntuación numérica) y en la reducción del número de asignaciones prácticas para mantener la viabilidad del curso. En consecuencia, el estudiante recibe menos oportunidades de practicar y refinar competencias clave de programación.

Los sistemas de autocorrección cambian el paradigma al proporcionar retroalimentación casi instantánea y repetible. El estudio de Wisniewski y Kulbacki [2] demostró que pasar de un ciclo de compilación nocturna a uno bajo demanda redujo las entregas tardías del 34 % al 4 %. De forma análoga, la incorporación de un pipeline automático en un CS1 masivo de 400 estudiantes eliminó por completo los retrasos de calificación, mejoró en un 22 % la nota media y duplicó la participación en pruebas voluntarias [8]. Estos resultados se alinean con la revisión sistemática de Messer et al. [15], que destaca la correlación positiva entre feedback inmediato y logro académico.

Al ejecutar las entregas en un contenedor estandarizado dentro de un flujo CI/CD (p.ej., GitHub Actions), se eliminan las discrepancias de entorno, garantizando que la evaluación sea reproducible y objetivamente comparable. Este enfoque minimiza el tiempo dedicado a resolver incidencias de configuración y centra la corrección en la calidad funcional del código. Los beneficios esperados como resultado de implementar este sistema se pueden condensar en los siguientes puntos:

- Formación en prácticas DevOps: la ejecución de pipelines aproxima al alumnado a los flujos de trabajo profesionales, reforzando la empleabilidad.
- Datos para learning analytics: los resultados estructurados (logs, métricas de test, tiempos de ejecución) facilitan el análisis de patrones de error y la adopción de docencia adaptativa [16].
- Motivación y autorregulación: indicadores visuales en la plataforma (checks verdes/rojos) actúan como elementos gamificados que incrementan la frecuencia de iteración y promueven la autoeficacia [4] [2].

La evidencia empírica converge en que la corrección manual, aunque valiosa para aspectos cualitativos, no escala a las necesidades actuales de las titulaciones multitudinarias. La automatización parcial mediante pipelines reproducibles aporta beneficios duales: reduce drásticamente la carga docente y mejora la experiencia de aprendizaje gracias al feedback inmediato, justificando así la implantación de sistemas de autocorrección como el que se desarrolla en este trabajo.

2.3. Fundamentos técnicos

A continuación, se profundiza en los principales fundamentos técnicos que soportan el sistema de corrección propuesto, con el fin de brindar un mejor conocimiento

de los mismos. Dado que este trabajo incluye numerosos conceptos técnicos, se ha incorporado un listado de acrónimos y un glosario de términos, con la intención de facilitar su interpretación y comprensión.

2.3.1. Git

Git, creado por Linus Torvalds en 2005, almacena la historia del proyecto como un gráfico acíclico dirigido (o DAG): cada commit apunta a su padre (o padres) mediante un hash criptográfico SHA-1/SHA-256 que sella tanto el contenido como la ruta de acceso. A efectos prácticos, un commit no es una versión completa, sino la diferencia respecto a su anterior, aunque Git materializa en una instantánea lógica la reconstrucción de todo el árbol de trabajo. Como características principales encontramos:

- Linealidad navegable. Dado que el DAG preserva el orden temporal, el desarrollador puede avanzar (fast-forward) o retroceder sin pérdida (comandos checkout, revert, reset).
- Ramas y estrategias de integración. Las ramas (branches) son punteros móviles dentro del DAG. Git ofrece merge (historial preservado), rebase (re-escritura para linealidad) y squash merge (condensar múltiples commits en uno) según se prefiera claridad o granularidad del historial.
- Trabajo distribuido. Cada clon de un mismo proyecto contiene la historia completa, permitiendo confirmar cambios sin conexión y garantizar resiliencia del proyecto.
- Etiquetas (tags) y versiones. Las etiquetas nombran puntos estables (v1.0, v1.1-rc) y sirven para releases reproducibles.

Pedagógicamente, Git promueve trazabilidad, revertibilidad y colaboración profesional temprana [1]; su manual Pro Git [17] (2ª ed.) se considera texto canónico.

2.3.2. GitHub

GitHub extiende Git con una plataforma cloud que integra gestión de código, automatización y comunidad. Con más de 100 millones de desarrolladores (2023) y 372 millones de repositorios (Octoverse 2024 [18]), es la plataforma dominante tanto en código abierto (OSS) (-90 % de proyectos OSS alojados) como en entornos empresariales. Entre sus funcionalidades, las principales son:

- Repositorios y pull requests con revisiones, checks automáticos y branch protection.
- Issues / Projects / Discussions para seguimiento ágil y Kanban.
- GitHub Actions — CI/CD nativo; Packages & Container Registry para artefactos: Dependabot & CodeQL para seguridad.
- GitHub Pages despliega sitios estáticos directamente desde la rama gh-pages o instrucciones personalizadas.
- Gists ofrecen snippets versionados y compartibles.
- Organizaciones y equipos. Una organización agrupa repositorios y define permisos a través de equipos (lectura, escritura, administración). Este modelo es

la base para proyectos profesionales y, en el ámbito educativo, para las clases gestionadas por GitHub Classroom.

GitHub ofrece planes gratuitos ilimitados para código abierto y licencias educacionales sin coste; en 2024, 7 millones de usuarios verificados formaban parte de GitHub Education [19] y 450,000 estudiantes realizaron su primer commit público.

2.3.3. GitHub Classroom

GitHub Classroom se apoya en una organización pública o privada creada por el profesor, y que actúa como contenedor de todos los repositorios de la asignatura o de un curso. Su flujo de uso se describe a continuación:

- Configurar la clase. Importar la lista de estudiantes, crear equipos y establecer plantillas de tarea (starter code).
- Publicar la invitación. Un enlace único genera automáticamente un repositorio privado (individual o grupal) vinculado al estudiante o equipo.
- Trabajar en el repositorio. El alumno clona, desarrolla y hace push (comando git) de los cambios; los pull requests quedan sujetos a autograding si se activan pruebas.
- Cuadro de mando / Dashboard docente. Muestra estadísticas de entrega (número de commits, rama activa), registros de Actions, estado de los tests y vista comparativa que permite filtrar por quien aún no pasa las pruebas. Se pueden descargar calificaciones, dejar comentarios de código o generar feedback masivo.

El autograding nativo soporta pruebas de línea de comandos, puntos de salida/entrada o scripts unitarios [5] [11] y limita cada ejecución a 20 minutos y 5.000 logs [20]; sin embargo, para rúbricas complejas (cobertura, static code analysis, contenedores específicos) se recomienda un workflow de GitHub Actions personalizado, descrito con detalle en la Sección 3.3. Classroom también integra Codespaces para disponer de un IDE en la nube y permite la exportación de notas en CSV, facilitando su carga en sistemas de gestión de aprendizaje (LMS) externos.

2.3.4. CI/CD

El concepto de Integración Continua (CI — continuous integration) fue popularizado a principios de los 2000 (Duvall, 2007 [21]) como práctica de integrar cambios pequeños y frecuentes para detectar errores pronto. Posteriormente, la Entrega Continua (CD — continuous delivery), extendida por Humble & Farley [22] (2010), añadió la capacidad de empaquetar y desplegar automáticamente en entornos de producción o preproducción. Ambos procesos se orquestan mediante un pipeline que se dispara ante eventos de control de versiones (p. ej., un push o un pull request).

Un pipeline típico de integración y despliegue continuo se ve así:

1. Build – compilar código y resolver dependencias.
2. Tests unitarios — validar lógica básica.
3. Tests de integración y estáticos -– cobertura, linters, seguridad.
4. Empaquetado -– generar artefacto (JAR, Docker image).

5. Publicación/Deploy — subir a un registro o entorno de ensayo.
6. Notificación — reportar estado a los desarrolladores.

Como beneficios educativos y profesionales, se destacan:

- Reduce la «integración infernal» al detectar conflictos temprano.
- Proporciona entornos homogéneos reproducibles [23], mitigando el clásico "works on my machine".
- Ofrece retroalimentación automática en minutos, alineada con metodologías ágiles y aprendizaje iterativo.

Herramientas como Jenkins (2004), GitLab CI (2014) o Travis CI (2011) fueron los precursores de esta práctica en la industria; GitHub Actions (2019) la integró en el ecosistema de GitHub, eliminando la necesidad de configuraciones externas.

2.3.5. GitHub Actions

GitHub Actions es la funcionalidad nativa de integración y entrega continua (CI/CD) de GitHub. Permite automatizar tareas dentro del ciclo de desarrollo definiendo flujos de trabajo en archivos YAML, almacenados en el directorio `.github/workflows/` del repositorio. Desde su lanzamiento estable en 2019, se ha convertido en una solución popular para la automatización en proyectos de software por su integración directa, flexibilidad y bajo umbral de entrada.

Un flujo de trabajo (workflow) se activa mediante eventos y puede contener uno o varios trabajos (jobs), que a su vez se componen de pasos (steps). Los jobs se ejecutan en entornos efímeros proporcionados por GitHub (runners) que simulan sistemas Linux, Windows o macOS, aunque también es posible registrar runners autohospedados para tareas más exigentes o específicas.

Cada paso puede ejecutar comandos directamente o invocar acciones reutilizables (actions) que encapsulan tareas comunes, como compilar proyectos, ejecutar tests o desplegar artefactos. Estas acciones pueden encontrarse en el GitHub Marketplace, que en 2025 ya supera las 22.000 acciones públicas.

2.4. Git y GitHub en la docencia

GitHub Classroom es hoy la herramienta de referencia para distribuir, recoger y evaluar prácticas de programación directamente en GitHub. Al apoyarse en la infraestructura descrita en la Sección 2.3.3, ofrece al estudiante un entorno idéntico al que encontrará en proyectos profesionales y comunidades de código abierto.

2.4.1. Beneficios pedagógicos

Los estudios más recientes señalan cinco ventajas principales que justifican su adopción:

Aprendizaje activo y feedback inmediato. Cuando Harvard migró CS50 a GitHub Classroom, las comprobaciones de autograding pasaron de ejecutarse una vez al día a hacerlo en cada commit. El 72 % de los 800 estudiantes consultados indicó que ese

bucle de ensayo y error en cuestión de minutos le ayudó a "aprender haciendo" [5]. El resultado fue un aumento del número medio de commits (de 5,8 a 17,4) y una consolidación más rápida de los conceptos básicos.

Autenticidad industrial. Duke University incorporó pull requests y revisiones de código como parte de la calificación. Al cierre del semestre, el 82 % de los alumnos declaró sentirse mejor preparado para prácticas profesionales [24]. Los instructores señalan que la exposición temprana a ramas, revisiones y CI/CD reduce la brecha entre aula e industria y motiva a los estudiantes a seguir buenas prácticas de ingeniería.

Colaboración y trazabilidad. En la Universidad de Almería (UAL), los profesores exigieron el uso de issues y plantillas de PR para documentar el avance semanal. La tasa de contribuciones registradas por equipo creció un 43 % [8], y los conflictos de integración descendieron al disponer de un historial auditable de decisiones.

Learning analytics basadas en datos reales. Chen et al. [16] procesaron 40.000 eventos de Classroom y demostraron que simples métricas —frecuencia de commits, tiempos de respuesta a reviews, etc.— predicen con 0,81 AUC la nota final. Esta capacidad analítica permite detectar a tiempo a los estudiantes rezagados y proponerles tutorías específicas.

Motivación gamificada. En la Universidad de Almería, los indicadores visuales de Classroom (círculos verdes/rojos) incrementaron un 28 % la participación voluntaria antes de la fecha límite[8]. Los alumnos afirmaron que perseguir los "círculos verdes" convierte la práctica en un reto lúdico y disminuye el plagio porque resulta más gratificante depurar que copiar.

En síntesis, la literatura coincide en que la retroalimentación automática integrada en un flujo de trabajo real impulsa el aprendizaje significativo: acelera los ciclos de mejora, fomenta la colaboración estructurada y expone al alumno a herramientas industriales desde los primeros cursos. Estos beneficios se observan tanto en cursos masivos como en grupos reducidos y se sostienen incluso cuando el autograding sólo cubre la dimensión funcional del código.

2.4.2. Desafíos y limitaciones

- Curva de aprendizaje inicial. Conceptos como ramas y pull requests generan ansiedad en estudiantes de primer curso; se recomiendan talleres guiados y repositorio plantilla [19].
- Complejidad para el docente. La primera configuración de YAML y gestión de secrets puede requerir hasta seis horas [11], aunque el esfuerzo se amortiza en cursos sucesivos.
- Evaluación cualitativa incompleta. El código puede pasar los tests pero seguir siendo pobre en estilo; en algunas experiencias se combina linters automatizados con rúbricas humanas [8].

- Escalabilidad de infraestructura. Los 50.000 minutos de ejecución de acciones al mes gratuitos pueden agotarse en cursos con cientos de repositorios; la Universidad de Almería lo solventó con self-hosted runners reciclados [8].
- Privacidad y exposición pública. Se mitiga manteniendo repositorios privados y publicándolos sólo tras la calificación.

2.4.3. Casos de estudio ampliados

A continuación, se entra en más detalle sobre iniciativas de adopción de sistemas con la misma base técnica en otras universidades.

Harvard University. GitHub Classroom se utilizó para distribuir y corregir automáticamente prácticas en cursos multitudinarios. Los estudiantes trabajaban con repositorios individuales generados a partir de plantillas, y al subir sus soluciones se ejecutaban tests de forma automática. La herramienta redujo tareas administrativas y fomentó el uso de Git en el aula [5].

Heriot-Watt University. La integración de GitLab con Canvas permitió escalar la enseñanza de programación a múltiples campus, optimizando el tiempo administrativo en un 16.7 %. La automatización con GitLab CI proporcionó retroalimentación inmediata a los estudiantes, mientras que el análisis de actividad basado en commits y pruebas ayudó a monitorear el compromiso académico. La plataforma centralizó evaluaciones y materiales de aprendizaje en un flujo unificado, aunque se identificaron desafíos como la fragmentación de datos, la carga administrativa manual y la curva de aprendizaje inicial para estudiantes sin experiencia previa [25].

Duke University. El equipo combinó Classroom con NB-Grader y un conjunto de acciones propias. El 95 % de la corrección funcional quedó automatizado y los estudiantes podían consultar un informe HTML con cobertura y lint cada vez que abrían un pull request. Los tutores aprovecharon el histórico de revisiones para discutir decisiones de diseño en las sesiones de laboratorio [24].

Universidad de Almería. En un piloto que abarcó tres asignaturas (Java, C y Python) con más de 600 estudiantes, la facultad desplegó self-hosted runners sobre hardware del laboratorio para sortear los límites de minutos. Esto permitió ejecutar pruebas que requerían bases de datos y Docker-in-Docker. Los datos de uso mostraron que los estudiantes lanzaban de media 8,2 pipelines por práctica, lo que correlacionaba positivamente con la calificación final [8].

Universidad Rey Juan Carlos (URJC) de Madrid. En varias asignaturas de grados como Ingeniería Informática o Ingeniería de Computadores, se utiliza GitHub Classroom para gestionar entregas. Aunque no hay estudios acerca de los beneficios reportados o la metodología seguida en su implantación, es posible ver a través de GitHub el uso que dan a la herramienta.

2.5. Otras herramientas y enfoques

Aunque GitHub Classroom y Actions ofrecen una solución integrada, la literatura recoge más plataformas y servicios con el mismo objetivo: simplificar la entrega y corrección de prácticas o llevar la integración continua al aula. A continuación se revisan las opciones más citadas, subrayando sus puntos fuertes y sus limitaciones, aplicados al ámbito docente.

2.5.1. Plataformas educativas de corrección

Deliverit (UPM). Desarrollada en la ETSIINF-UPM, permite entregas individuales y grupales evaluadas mediante scripts personalizados (JUnit) [26]. Se ejecuta en máquinas virtuales aisladas y se integra con LDAP para autenticar al alumnado. Ventajas: control local de los datos y adaptación específica a los planes de estudio de la UPM. Límites: depende de infraestructura propia, interfaz menos pulida y curva de mantenimiento elevada; carece de un flujo Git completo, por lo que no fomenta ramas ni pull requests [27].

Gradescope (UC Berkeley). Plataforma SaaS utilizada en cursos masivos. Admite autograding con contenedores Docker y combinación con corrección manual en una misma rúbrica. Estudios en Indiana University reportan un 30 % de reducción del tiempo de calificación en asignaturas con más de 400 estudiantes [28]. Desventajas: la retroalimentación llega tras la entrega final, no integra control de versiones y conlleva un coste institucional.

Codeboard. IDE web de código abierto diseñado para la enseñanza de programación en entornos educativos. Ejecuta código directamente en el navegador y proporciona retroalimentación inmediata. Su facilidad de uso lo hace ideal para cursos introductorios; Universidad Carlos III de Madrid lo ha integrado en MOOCs de Java, observando mayor compromiso y actividad en estudiantes registrados [29]. Limitaciones: orientado a ejercicios cortos, sin soporte para proyectos complejos ni trabajo colaborativo con ramas.

Autolab. Plataforma de código abierto de gestión de cursos con evaluación automática de código. Desplegado en Carnegie Mellon University y la Universidad Rey Juan Carlos [30], ha demostrado eficacia en la corrección de prácticas de C, Java y Python, permitiendo pruebas en entornos aislados y configuraciones flexibles de evaluación. Su capacidad para detectar plagio y proporcionar retroalimentación inmediata ha mejorado la eficiencia docente. Limitaciones: requiere administración de servidores y medidas de seguridad para garantizar la integridad de las evaluaciones.

Plug-ins para Moodle. El LMS utilizado en muchas universidades, dispone de módulos de programación (VPL, CodeRunner). Coventry University lo usa en primeros semestres con buen índice de satisfacción [31]. Como ventajas, permite unificar notas en el entorno ya familiar del alumno, a cambio de interfaces menos intuitivas, integración limitada con Git y CI externos.

2.5.2. Alternativas de CI/CD genéricas

GitLab CI/CD. En su integración educativa en Heriot-Watt University [25], la universidad optimizó la enseñanza de programación automatizando la entrega de código y evaluación en GitLab. Señalaron como ventajas el código abierto, escalabilidad en múltiples campus, integración con Canvas, feedback inmediato con GitLab CI y ahorro de tiempo administrativo (hasta 16.7 horas por semestre). La desventaja educativa apunta a la fragmentación de datos entre sistemas, carga administrativa manual, accesibilidad para estudiantes sin experiencia previa, riesgo de exposición de código y plagio. En este grado, en la asignatura Programming Project, apoyada sobre una infraestructura privada de GitLab, se enseña y se practica el uso de esta herramienta.

Jenkins. Pionero de la CI (2004). Su ecosistema de plugins permite encadenar análisis de calidad, pruebas y despliegues. P. Straubinger [32] mostró que Jenkins puede gamificar un curso de testing mediante insignias. Sin embargo, la administración de servidores y actualizaciones consume tiempo docente y la interfaz no está pensada para el alumno.

Travis CI y CircleCI. Servicios en cloud usados históricamente en proyectos de código abierto y docentes. Travis CI ofrece minutos libres para código abierto; CircleCI dispone de orbs pedagógicos. Inconvenientes: límites más severos para organizaciones educativas grandes y ausencia de tablero central de tareas.

Azure Pipelines y GitHub Enterprise Server. Ofrecen runners Windows y despliegues a Azure; útiles para asignaturas de DevOps multiplataforma, pero su complejidad rebasa normalmente las necesidades de un curso de grado.

2.6. Comparativa crítica

Para evaluar las herramientas descritas se adoptan cinco criterios recurrentes en la literatura de autograding y DevOps educativo [15][32][1]:

Integración con el flujo de desarrollo real. ¿La plataforma refleja el uso de control de versiones y CI tal como se practica en la industria?

Facilidad de adopción y uso. Esfuerzo inicial para docentes y barrera de entrada para estudiantes.

Escalabilidad y mantenimiento. Capacidad de soportar cursos grandes sin sobrecarga técnica.

Flexibilidad en criterios de evaluación. Posibilidad de personalizar pruebas, análisis estático y rúbricas.

Calidad y frecuencia de la retroalimentación. Rapidez, detalle y utilidad pedagógica de la retroalimentación.

2.6.1. Valoración cualitativa por criterio

Las categorías Alta, Media y Baja reflejan el rendimiento relativo de cada herramienta respecto al criterio específico. Valoraciones basadas en los casos de estudio citados

Plataforma	Integración	Adopción	Escalabilidad	Flexibilidad	Feedback
GitHub Classroom + Actions	Alta	Alta	Alta	Alta	Alta
Gradescope	Media	Alta	Alta	Media	Media
Deliverit	Media	Baja	Baja	Media	Media
Autolab	Alta	Baja	Media	Alta	Media
GitLab CI/CD	Alta	Baja	Alta	Alta	Media
Jenkins	Media	Baja	Media	Alta	Baja

Cuadro 2.1: Comparativa de plataformas de corrección automática

(Duke [24], Almería [8]) y en encuestas docentes globales [32] [1] [16].

A continuación, se entra en más detalle sobre cómo se presentan cada una de las alternativas frente a los criterios.

2.6.2. Integración con el flujo de desarrollo real

GitHub Classroom y GitLab CI destacan por alojar el código en repositorios Git reales e integrar pull requests y checks automáticos [24] [32]. Gradescope y Deliverit funcionan como repositorios de archivos, lo que aísla al estudiante del proceso DevOps.

2.6.3. Facilidad de adopción y uso

Gradescope obtiene la mejor nota: su interfaz drag-and-drop y rúbricas visuales permiten empezar en minutos [28]. GitHub Classroom sigue de cerca gracias al wizard de tareas, mientras que Jenkins y GitLab exigen conocimientos de YAML y administración de servidores [32], [1].

2.6.4. Escalabilidad y mantenimiento

Las soluciones cloud (GitHub Actions, Gradescope) escalan automáticamente; Autolab y Jenkins requieren servidores dedicados y actualizaciones periódicas, lo que puede ser inviable en cursos masivos [8], [1].

2.6.5. Flexibilidad en criterios de evaluación

Herramientas CI genéricas (GitLab, Jenkins, Autolab) permiten encadenar análisis de calidad, cobertura y pruebas de rendimiento [32]. Gradescope admite contenedores Docker, pero carece de orquestación multi-fase [28]. GitHub Actions equilibra flexibilidad y simplicidad con su Marketplace de más de 20.000 acciones [19].

2.6.6. Retroalimentación al estudiante

GitHub Classroom permite enviar resultados en cada commit (con menos de 5 minutos de latencia) y puede comentar líneas de código automáticamente. Gradescope entrega reportes detallados, pero sólo tras la fecha límite. Jenkins y GitLab dependen de cómo se configure la notificación, mientras que Deliverit se limita a mostrar los resultados de la ejecución en crudo, además del resumen de pruebas superadas y nota obtenida, sin visualización pedagógica.

2.7. Conclusiones del estado del arte

La comparativa anterior ha puesto de manifiesto que GitHub Classroom y Actions ofrecen, en términos generales, el equilibrio más favorable entre realismo industrial y facilidad de adopción. No obstante, ninguna solución revisada resuelve simultáneamente todos los retos técnicos y pedagógicos detectados en el uso de sistemas de corrección automática en el contexto universitario.

A continuación, se sintetizan las principales brechas técnicas y pedagógicas identificadas:

- Dualidad feedback rápido versus evaluación cualitativa. Gradescope y Deliverit facilitan rúbricas detalladas, pero carecen de devoluciones en tiempo real; GitHub Classroom ofrece comprobaciones instantáneos, pero su autograder nativo no evalúa diseño ni estilo [33] [32] [1].
- Orquestación Git y CI conectada a la docencia. Herramientas CI genéricas (Jenkins, GitLab) replican procesos industriales, pero obligan al profesorado a mantener servidores y a enlazar manualmente con el LMS [25] [19].
- Escalabilidad sin costes de mantenimiento. Las soluciones on-cloud alcanzan, generalmente, el límite cuando se superan los 250 estudiantes. Aquellas on-premise se ajustan a la capacidad necesaria, pero requieren infraestructura y gasto desde el comienzo.
- Análisis de aprendizaje explotable. Sólo estudios aislados han vinculado eventos de autograding con predicción del éxito académico; la mayoría de plataformas no exponen APIs ni métricas listas para *learning analytics*.

Frente a estas limitaciones, el sistema propuesto en este Trabajo de Fin de Grado plantea una solución técnicamente viable y pedagógicamente alineada con las necesidades actuales. El pipeline desarrollado sobre GitHub Actions proporciona feedback funcional automático con una latencia menor a cinco minutos (para la mayoría de los casos), validando los beneficios reportados por experiencias como la Universidad de Almería [8]. Su diseño modular permite extender los workflows con etapas de análisis estático, linters y cobertura de tests, cerrando la brecha cualitativa observada en cursos como el de la URJC. Además, aprovecha los minutos gratuitos que ofrece GitHub para escalar sin costes iniciales, combinables con runners autoalojados para pruebas más exigentes, en un modelo híbrido exitoso en contextos previos. Complementariamente, el sistema permitirá exportar datos de uso y ejecución en formatos interoperables como CSV o JSON, lo que abre la puerta a iniciativas de learning analytics inspiradas en propuestas como la de Quick et al. [4]. Finalmente, para reducir la barrera de entrada, se acompañará de repositorios de inicio, documentación paso a paso y ejemplos funcionales, siguiendo el modelo didáctico de Duke [24].

El capítulo 3 detallará el diseño e implementación de dicho sistema de corrección automática, incluyendo la arquitectura del pipeline, los ficheros YAML generados y los casos de uso.

Capítulo 3

Diseño y Desarrollo

La sección de diseño y desarrollo profundiza en cómo se ha diseñado técnicamente la solución propuesta, abordando desde la arquitectura del sistema hasta la gestión de repositorios, automatización, control de errores y estrategias para garantizar su escalabilidad y mantenimiento.

3.1. Arquitectura general del sistema

El sistema de corrección automática se apoya en la integración nativa entre GitHub Classroom y GitHub Actions para ofrecer un bucle de retroalimentación continuo al alumnado. Una vez que el docente publica la práctica en Classroom, cada estudiante genera un repositorio individual a partir de la plantilla oficial. En el caso de proyectos en equipo, los estudiantes primero se inscriben en un grupo de prácticas en su panel de GitHub Classroom. Según el disparador configurado en la acción, se desencadenará la ejecución de uno o varios workflows que validan el código y devuelven resultados. En casos sencillos, podemos esperar tiempos de menos de un minuto de media. Esto ayuda a afianzar conceptos de Git, gestión de versiones y de proyectos. Más allá, adoptarlo en un conjunto amplio de prácticas mantiene a los estudiantes dentro de un mismo ecosistema.

3.1.1. Componentes principales

Organización. La gestión organizacional del sistema es a través de este componente. Los usuarios de GitHub son capaces de crear y unirse a organizaciones, en las cuales agrupan esfuerzos comunes. Además, permite gestionar equipos, roles, permisos por repositorio y Actions permitidas dentro de la organización, entre otros. Es la base sobre la que se monta el sistema.

GitHub Classroom. Accesible a través de Classroom, tanto alumnos como profesores pueden acceder a las clases. Desde aquí se gestionan los grupos de alumnos, asignaturas y prácticas. Facilita el control de permisos a nivel de organización educativa, aunque se sigue apoyando en una organización previamente creada en GitHub. Todos los repositorios creados por alumnos o profesores estarán disponibles en la organización con los permisos designados (visibilidad del repositorio).

Repositorios plantilla del profesorado. En estos repositorios, los profesores podrán organizar Actions usadas a lo largo de sus asignaturas, documentación acerca de cómo adaptar o generar nuevas Actions, además de esqueletos de código para prácticas y enunciados para las mismas. Facilita la transferencia de conocimiento, tratando de reducir la curva de aprendizaje para nuevos usuarios del sistema.

Repositorios plantilla para alumnos. Para cada práctica propuesta a los alumnos, además de determinar cómo realizar la corrección de estas prácticas, se puede ofrecer un código plantilla. Es común que en ciertos proyectos se ofrezca un código base a partir del cual el alumno tiene que completar (cuando se busca evaluar si el alumno es capaz de resolver un problema sin necesidad de configurar todo lo necesario alrededor). Tradicionalmente, se ha compartido con los alumnos un fichero comprimido con todo el material necesario.

También se puede ver beneficiada una práctica en la que se pide diseñar un sistema completo, por ejemplo, designando Actions para que el alumno pruebe en su entorno local, archivos `.gitignore` ya completados para reducir ruido externo producido por configuraciones comunes. Asegura además la autoría (restringiendo la visibilidad del repositorio a sólo el alumno una vez clonado) y permite un seguimiento personalizado.

GitHub Actions. Motor CI/CD que ejecuta los *workflows* de evaluación. Pueden ser tanto creados por el alumno para realizar sus propias pruebas como los correctores marcados por los profesores para las entregas. Es un servicio cloud administrado que simplifica la infraestructura, dependiente de los Runners que ejecutan estos *workflows*.

Runners. Máquinas que llevan a cabo los *jobs* presentes en un *workflows*. Por defecto, los *runners* se alojan en la nube, provisionados por GitHub (*GitHub Runners*) por defecto, pero también se pueden hospedar. Estos *self-hosted runners* permiten superar límites de tiempo marcados por los planes de GitHub o asignar software específico.

Secretos y variables. Almacenan credenciales y permiten manejar tests privados. Evitan exponer soluciones y protegen datos sensibles. Cobra sentido en el escenario planteado en la sección 3.3.2, donde se quiere tener el corrector distribuido y usarse para corregir las prácticas sin exponerlo al alumnado.

3.1.2. Flujo de eventos y automatización

1. **Creación de la práctica.** El docente prepara el material y corrector para una práctica de una asignatura. Publica un assignment en Classroom, vinculándolo a la plantilla y configura el corrector a medida según las necesidades específicas de la práctica. Adicionalmente, comparte un enlace con los alumnos para que puedan acceder al Classroom y así quedar registrados.
2. **Aceptación por el alumnado.** Al acceder al enlace, el alumno queda registrado en la práctica. En el caso de ser grupos de prácticas, como ya se adelantaba antes, los alumnos se registran en este punto en un grupo de prácticas, a partir del cual se generaría un proyecto único para todos los miembros. Classroom

clona la plantilla en un repositorio privado del estudiante o grupo dentro de la organización a la que está sometida Classroom y concede permisos de escritura.

3. **Desarrollo local y push.** El alumno hace commits con los que avanza en la práctica. Ante el disparador seleccionado para la práctica en el *workflow* YAML correspondiente, se realiza la automatización. El patrón de ejecución desencadenado por hacer push de un commit — commit → validación → feedback inmediato — está recomendado por la propia guía de autograding de Classroom como la forma más efectiva de fomentar la autoevaluación iterativa.

   Es por ello que se recomienda ofrecer al alumno un corrector más básico, con un subconjunto de pruebas a ejecutar durante la corrección, para que el alumno pueda ejecutar éste de manera más continuada y decidir hacer una corrección completa cuando está confiado con sus resultados. Esto trae dos beneficios. Por un lado, se reduce la cantidad de minutos usados por el *worklow* corrector de prácticas, al distribuir gran número de correcciones a aquel *workflow* que tenga el alumno en su repositorio. Por el otro, abre las puertas a que el alumno pueda modificarlo, incluso crearlo en casos sencillos si se considera oportuno, para ampliar sus conocimientos en practicas DevOps dentro del marco educativo.

4. **Ejecución de la evaluación.** GitHub Actions levanta un runner, instala dependencias y lanza las suites de pruebas (IO, JUnit, Maven, RMI, etc.). Por defecto, el alumno podrá ver los resultados de la ejecución como un log (al igual que si ejecutase por terminal y fuese viendo lo que se va imprimiendo).

5. **Generación de resumenes y logs.** Adicionalmente, se pueden configuar resúmenes o resultados parciales que presentar al alumno de una forma más clara. Para cada ejecución de una acción, se puede presentar un resumen, en el que incluir resultados (p.e. en el caso de pruebas, devolver el número de pruebas superadas. Para entrada y salida, ofrecer lo obtenido frente a lo esperado en caso de error. En casos donde la ejecución no sea satisfactoria, mostrar cuánto tiempo ha estado ejecutando). En el caso particular de configurar un *workflow* para ejecutar ante una pull request (p.e. `on: pull-request: branches: main` — *'cuando se hace una pull request a la rama main'*) los commits sólo se evaluarán al estar en la rama que se está intentando hacer *merge* con `main`. Esto nos ofrece como contexto la pull request, lo que nos permite que la Action publique un comentario en la conversación de la pull request. Esta es la forma más clara de ofrecer *feedback* al alumno, ya que por defecto, si un flujo falla, se suele notificar al usuario. Si esto se acompaña de un mensaje donde se resume el punto de error o las condiciones no satisfechas, es más fácil para el alumno entender el error y solucionarlo. Solo los mensajes previstos llegan al alumno para no revelar pruebas sólo visibles para los docentes o claves privadas y secretos, ocultos siempre dentro de GitHub.

6. **Iteración.** El estudiante hace cambios en su código, vuelve a provocar la acción que activa el disparador y observa la nueva puntuación.

7. **Corrección final por el profesor.** Una vez acabado el periodo de presentación de prácticas, el profesor es capaz de exportar los resultados de las correcciones automáticas a cualquier otro sistema de gestión, incluidos los LMS. Esto sólo recoge la puntuación más alta o final de cada repositorio (alumno individual o grupo). Además, al tener acceso a los repositorios, no es necesario que los alum-

nos entreguen nuevamente el código, En este punto, el docente puede realizar una corrección del código, sin tener que preocuparse por los resultados funcionales del código, pudiéndose centrar en otros aspectos como estilo, buenas prácticas y optimización. Además, en proyectos grupales, tiene acceso al historial de commits con el fin de comprobar que todos los integrantes de un grupo han colaborado.

A continuación, se muestra un diagrama de secuencia para describir la interacción de los componentes con el fin de facilitar la comprensión del sistema propuesto para la corrección automática de prácticas.

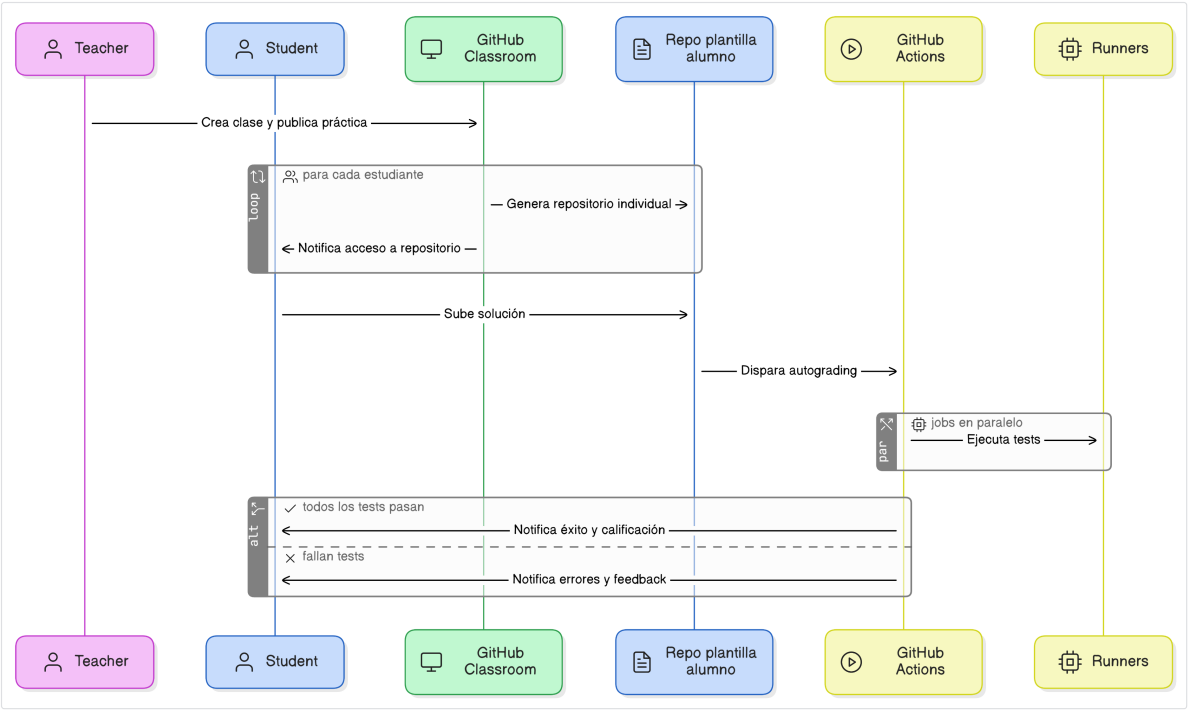


Figura 3.1: Diagrama de secuencia simplificado

En el anexo (Sección A.2), se ofrece un diagrama de secuencia que muestra con más detalle el sistema.

3.1.3. Gestión de roles y permisos

Una asignación granular de privilegios es clave para salvaguardar la integridad académica, así como la protección de datos del alumnado. El modelo adoptado sigue la filosofía de mínimo privilegio que GitHub prescribe para Classroom y Actions, de modo que cada identidad disfruta solo de los permisos estrictamente necesarios para su tarea.

- **Profesorado (*teacher*).** Cada docente actúa como *owner* de la organización Classroom correspondiente. Puede crear o archivar aulas, configurar *workflows*, gestionar *secrets* y volver a ejecutar evaluaciones. Gracias a ese rango, puede acceder a los *logs* completos y a los artefactos generados, lo que le permite corregir prácticas y gestionar incidencias o dudas de alumnos sin exponer datos a

terceros.

- **Administradores de organización.** Este grupo, formado normalmente por personal de soporte o TI, asume tareas transversales: control de facturación, políticas de seguridad y gestión de minutos de CI. GitHub ofrece para ello el rol *billing manager*, que otorga visibilidad de costes sin acceso al código fuente.
- **Alumnado (*student*).** Cada estudiante sólo ve su propio repositorio (o el del grupo) y el resultado de las pruebas y acciones que se ejecutan sobre su repositorio. Los repositorios de otros compañeros tendrán visibilidad espejo. Es decir, los repositorios se crean con los mismos permisos. Si el docente considera que sólo un alumno, o grupo, debe ver su grupo, los alumnos podrán ver únicamente su repositorio. Si por el contrario se decide dar un enfoque más colaborativo y se hacen los repositorios públicos, todos los alumnos tendrán acceso a todos los repositorios de sus compañeros para la práctica en cuestión. De esta forma se previenen plagios directos.
- **Cuentas de servicio para *runners self-hosted*.** Cuando un curso requiera infraestructura propia, los *runners* se registrarán con un *token* limitado que sólo les permite descargar código y actualizar el estado del *check*; no pueden crear ramas ni abrir *pull requests*. De acuerdo con la guía de endurecimiento de Actions, el nodo se etiqueta (`self-hosted, upm-linux`) y se recomienda operar en una VLAN aislada para mitigar riesgos de inyección de código.
- **Bot de GitHub Classroom.** El asistente automatizado que crea repositorios y equipos trabaja con credenciales temporales basadas en OAuth; al finalizar la operación, el *token* expira, reduciendo la superficie de exposición.

Todas las credenciales se almacenan en *Secrets & Variables*, donde las políticas de acceso por entorno impiden su filtrado accidental en los *logs* por defecto.

3.2. Diseño de repositorios y plantillas

3.2.1. Principios de diseño y criterios previos

De acuerdo con lo expuesto en la sección anterior, para la creación de un curso se hará uso de un *template repository* para garantizar que cada práctica parte siempre de la misma arquitectura de carpetas, ramas y ficheros. GitHub describe esta funcionalidad como la forma recomendada de "generar nuevos repositorios con la misma estructura" y limita su activación a usuarios con privilegios de administración del repositorio original. Adoptar plantillas desde el primer día evita errores de configuración (rutas erróneas de compilación o *workflows* ausentes). A mayores, conseguir que todas las asignaturas compartan un esquema muy similar de plantillas facilitaría que el alumnado aprenda una única vez y reaproveche semestre tras semestre. La otra cara de la moneda es que los alumnos no vean alternativas de inicialización de proyectos, acostumbrándolos a este marco común; pero, sin duda, una estructura común que siga buenas prácticas favorecerá el aprendizaje.

Desde la perspectiva pedagógica, una plantilla orientada a la automatización transfiere al estudiante las buenas prácticas de ingeniería profesional: pruebas alojadas en un directorio dedicado, integración continua activa desde el *commit* inicial y documentación viva junto al código. Todo ello coincide con las directrices de autograding

publicadas por GitHub Classroom, que enfatizan el ciclo ya mencionado — commit → validación → feedback inmediato — como pilar de la autoevaluación iterativa. Metodológicamente, esta aproximación se alinea con la estrategia *Prof. CI*; dicho enfoque demostró que el alumnado interioriza antes el desarrollo guiado por pruebas (TDD) cuando trabaja en repositorios personales vinculados a un servicio de CI que ofrece retroalimentación automática [34].

En consecuencia, la plantilla se ha construido bajo tres criterios explícitos:

- **Consistencia transversal**: esquemas válidos que se adecúan a los diferentes escenarios extendidos en las asignaturas y cursos, lo que elimina ambigüedades y facilita la transferencia de conocimiento entre materias.
- **Mínima fricción**: el estudiante clona el repositorio y compila sin pasos adicionales; todas las dependencias están predefinidas en la plantilla, reduciendo incidencias de entorno.
- **Mantenibilidad a largo plazo**: el profesorado actualiza una sola fuente y distribuye los cambios mediante versiones etiquetadas de la plantilla, garantizando trazabilidad y reversibilidad.

3.2.2. Estructura de carpetas estándar

La jerarquía entregada al alumno sigue el principio 'convention over configuration' y se materializa en el listado de la Figura (ejemplo real abreviado):

```
repo-plantilla/
+-- src/
|   +-- main/
|   |   +-- java/            # Codigo fuente Java (p.e., com/example/app/)
|   |   `-- resources/       # Archivos de condiguracion, recursos
   estaticos
|   `-- test/
|       +-- java/            # Tests unitarios
|       `-- resources/       # Recursos especificos para pruebas
+-- README.md                # Archivo de informacion del proyecto
+-- docs/                    # Documentacion
+-- scripts/                 # Scripts adicionales
+-- .github/workflows/       # GitHub Actions workflows (YAML files)
`-- .gitignore               # Archivo de configuracion de Git
```

Listing 3.1: Ejemplo del contenido del repositorio ofrecido al alumno como plantilla

Cada directorio cumple un propósito preciso respaldado por la documentación oficial de GitHub y por las convenciones de la comunidad Java:

`src/` Aloja el código fuente. Se adopta la convención de estructura Maven/Gradle (`src/main/java` y `src/test/java`) que también se puede aplicar en proyectos Java de menor escala. Seguir este estándar evita rutas ad hoc y simplifica el uso de herramientas de análisis estático y cobertura. En proyectos que divergen de esta estructura, como completados de código o clases únicas, se puede prescindir de las subcarpetas para reducir la complejidad cognitiva de la organización.

`tests/` Contiene las pruebas públicas, generalmente creadas por el alumno o, creadas por los docentes pero es sabido que van a ser visibles por el alumno. Dentro, el subdirectorio `resources/` encapsula ficheros de entrada y oráculos de salida. Mantener los *fixtures* públicos en un único punto permite que el estudiante ejecute (en el caso de de proyectos Maven) `mvn test` localmente y obtenga el mismo veredicto que el *pipeline*, reforzando la confiabilidad del ciclo *local* → *CI*. Estas pruebas pueden ser o no las mismas que evalará el corrector automático designado para tal efecto.

`.github/workflows/` Carpeta canónica donde deben residir los ficheros YAML de GitHub Actions. Colocar aquí los *workflows* activos garantiza que cualquier nueva rama o *fork* los herede sin configuración adicional.

`docs/` Repositorio para la documentación generada (p. ej. Javadoc) y para guías de uso redactadas por el estudiante. Incluir aquí el enunciado y material de referencia anticipa las prácticas de Ingeniería del Software donde la calidad de la documentación es evaluable.

`scripts/` Alberga utilidades auxiliares: `build.sh`, despliegues locales o lanzadores de servicios (por ejemplo, el script que lanza varios servicios). Los *workflows* inactivos o experimentales también se almacenan aquí, mientras que los operativos deben vivir obligatoriamente en `.github/workflows/`. Es posible que herramientas adicionales requieran de carpetas propias en las que contener este tipo de archivos. Por ejemplo, en Visual Studio Code los scripts de ejecución de código se deben encontrar en `.vscode/launch.json` y estar debidamente formados.

`resources/` Ficheros binarios de apoyo —imágenes, JSON, datos de prueba voluminosos— mantenidos fuera de `src/` para no contaminar el árbol de código y acelerar las compilaciones incrementales.

`README.md` Documento de cabecera. GitHub muestra automáticamente el `README` en la página del repositorio, por lo que se convierte en el primer punto de contacto con la práctica. Puede usarse para solicitar al alumno o grupo que haga un resumen de las funciones de su proyecto así como cualquier consideración adicional de configuración que se deba tener en cuenta.

Archivos auxiliares `.gitignore`, `.gitattributes`, licencias y badges CI completan la raíz. Ignorar artefactos generados previene repositorios inflados, mientras que los badges comunican el estado del *pipeline* de un vistazo.

Esta organización es la propuesta para un proyecto Java, idealmente usando Maven, seleccionada por ser comprensible. Atendiendo a las necesidades específicas de cada práctica, se puede modificar según convenga.

3.2.3. Guías diferenciadas para estudiantes y docentes

Un solo *template* debe atender a dos perfiles con expectativas muy diferentes: estudiante y docente, sin generar ruido innecesario a ninguno. Para conseguirlo, se sugieren tres recursos prácticos:

Secciones "ocultas" en `README.md`. En caso de usar este archivo, se pueden incluir apartados dirigidos exclusivamente al profesorado (p. ej. criterios de ponderación, rú-

bricas detalladas o instrucciones de mantenimiento). Por ser un archivo MarkDown, admite formateo básico HTML, por lo que se pueden incluir comentarios:

```
<!--teacher-only-start
### Rubrica interna
- Cobertura de las pruebas > 80%  ->  25 % de la nota
- Limpieza del codigo -> 15 % ...
teacher-only-end-->
```

Listing 3.2: Ejemplo de rúbrica interna

GitHub renderiza el bloque como oculto, de modo que la vista por defecto permanece limpia, pero si se ve el archivo en crudo, se dispone de esta información adicional. Esto no se intenta ocultar al alumno, sino contener información útil para el profesor *in situ* sin añadir ruido. Para el estudiante, el `README` se convierte en un espacio vivo donde documentar su práctica: descripción funcional, decisiones de diseño, versiones implementadas y limitaciones. Esto replica el uso real del archivo en proyectos profesionales—ayudar a un tercero a entender si el repositorio le interesa y cómo ejecutarlo.

Tests privados inyectados en tiempo de CI. Las pruebas ocultas residen en un repositorio docente y se clonan durante la ejecución del *workflow* con una clave SSH de solo lectura almacenada como *repository secret*. El flujo está diseñado para que el estudiante reciba solo la porción de feedback que el profesor considere útil (típicamente la nota global y un breve resumen de por qué ha fallado alguna prueba) sin exponer el material de corrección. Cuando los tests privados se inyectan durante el CI, estos se descargan dentro del runner, se ejecutan y se descartan en el mismo contenedor efímero; el alumno nunca obtiene el archivo .jar ni las clases fuente. En la salida que GitHub muestra por defecto, el alumno puede llegar a ver metadatos inevitables, como el nombre de la clase JUnit que se ha lanzado o el identificador de un caso de prueba, pero no tiene acceso a la lógica interna ni a los datos de los asserts, a menos que el docente lo imprima explícitamente en consola. Los artefactos que se publican (un reporte de cobertura, un fichero result.txt con el veredicto, etc.) permanecen disponibles solo el tiempo configurado y contienen información filtrada: puntuaciones, mensajes de error controlados y, si procede, una traza resumida. Este patrón mantiene la integridad pedagógica de la evaluación: el alumno sabe qué mejorar, pero no puede copiar la solución ni deducir casos ocultos, y el profesorado conserva la libertad de publicar los tests más adelante si desea transformarlos en material de estudio.

***Issue template* para la entrega final.** Al abrir la *pull request* definitiva, aparece un cuadro de verificación automático:

```
- [ ] Compila sin warnings
- [ ] Pasa `mvn test` local
- [ ] README actualizado
```

Listing 3.3: Ejemplo de plantilla de comprobaciones previas a una entrega

Esta lista de comprobación actúa como recordatorio de buenas prácticas y reduce el número de correcciones triviales: el alumno valida su código antes de solicitar la calificación y el profesorado recibe entregas de mayor calidad desde el primer intento.

3.2.4. Mantenibilidad y evolución de la plantilla

Para garantizar la transferencia de conocimiento entre cursos y profesorado, se recomienda conservar todas las plantillas, junto con su documentación técnica, que se conservarán en un repositorio privado. Este repositorio, accesible sólo para el equipo docente, estará contenido en la organización. Así se evita que el know-how quede disperso en múltiples ramas de proyectos pasados y se facilita la incorporación de nuevos profesores.

Políticas de evolución controlada

- **Versionado semántico.** Cualquier cambio incompatible incrementa la *major*: `template-v2.0.0`. Las promociones antiguas siguen trabajando con la versión previa, mientras que los nuevos cursos arrancan directamente con la plantilla actualizada.
- **Ramas dedicadas.** Cada serie `template-vX` contiene sólo los ficheros estables. El trabajo experimental vive en `dev` y no se fusiona hasta superar todas las pruebas. El objetivo es evitar que YAMLs inestables aterricen en producción, con el significado en este contexto de ser donde se corrigen las prácticas.
- **Pull requests con CI.** Toda modificación se somete al mismo pipeline que usa el alumnado. De este modo garantizamos que los ejemplos siguen compilando y que los cambios no rompen la experiencia del estudiante.
- **Registro de cambios.** El archivo `CHANGELOG.md` sirve para documentar nuevas funciones y cambios; desde la pestaña *Releases* se enlaza directamente para que otros docentes sincronicen sus scripts con un vistazo rápido.
- **Actualización automática de dependencias.** Dependabot abre PRs semanales. Los flujos de Actions comprueban que subir `JUnit`, `Maven Surefire` o cualquier otra dependencia, incluyendo dependencias en otras Actions, no rompe la suite de pruebas.
- **Refactorización controlada de carpetas.** Si una asignatura requiere, por ejemplo, un intérprete de Python, se añade un subdirectorio `languages/python` en lugar de mezclarlo dentro de Java. Así, se reduce la complejidad.

Medición de impacto. Previo y posterior a cada *release* se monitorizan los minutos de ejecución y la tasa de fallos del pipeline. Estas métricas permiten detectar regresiones tempranas y ajustar la plantilla antes de que el curso comience.

3.3. Catálogo de workflows

El sistema define tres *workflows* "tipo" que cubren los siguientes escenarios docentes: un corrector por *Entrada / Salida* (E/S) para ejercicios introductorios, una variante con *tests privados* embebidos para evaluar conceptos avanzados sin exponer el código al alumnado, y un pipeline completo para proyectos Maven de envergadura media. Cada uno se apoya en la acción oficial `actions/checkout` (descarga de repositorios), `actions/setup-java` (instalación JDK y caché) y las mejores prácticas de caché, `timeout` y `concurrency`. En las siguientes secciones se profundiza en mayor

detalle, con el objetivo de facilitar la comprensión de la implementación propuesta y el funcionamiento práctico del sistema desarrollado.

3.3.1. Entrada / Salida básico

Este flujo verifica la viabilidad de los ejercicios introductorios enfocados a asignaturas como Programación I y II, donde se solicita al alumno crear funciones o clases que superen casos para afianzar sus conocimientos. Esto se suele hacer comparando la salida en `stdout` con la esperada.

Como componentes del repositorio del alumno, esperamos encontrar la clase Main.java o similar en la que el alumno complete o cree la lógica que soluciona el problema que se le ha planteado. Además, en estos cursos introductorios es común que se ofrezcan las pruebas directamente, por lo que encontraremos también los casos de prueba en documentos `.txt` para que el alumno pueda practicar. La ejecución constará de la compilación del código, superar los tests y mostrar los resultados.

A continuación, se describen en orden los fragmentos de código que compondrían el flujo para esta casuística.

Timeout y concurrencia. Inicio del flujo.

```
name: IO-basic
on: [push, pull_request]

jobs:
  io-basic:
    runs-on: ubuntu-latest
    timeout-minutes: 3
    concurrency:
      group: ${{ github.ref }}
      cancel-in-progress: true

    steps:
```

Listing 3.4: Configuración de concurrencia y límites de tiempo en un flujo de GitHub Actions

Se designa un nombre para el flujo, los disparadores (en este caso, al subir código al repositorio y al abrir un 'pull request') y los trabajos a realizar. Se nombra el trabajo y se describe la máquina a usar. Se establece un límite de tres minutos y cancelación automática de ejecuciones solapadas en la misma rama. A continuación, se declararán los pasos del trabajo.

Checkout del código del alumno. Clona el repositorio del alumno en el *runner*.

```
    - uses: actions/checkout@v4
```

Listing 3.5: Clonación del repositorio en el runner en un flujo de GitHub Actions

Instalación de Java (sin caché). Configura el entorno para disponer de Java en futuros pasos.

```
- name: Configurar Java y cache
  uses: actions/setup-java@v4
  with:
    distribution: temurin
    java-version: 21
```

Listing 3.6: Instalación de Java sin caché en un flujo de GitHub Actions

Además añade el detalle de que a un paso se le puede declarar un nombre. Se elige una distribución de Java (en este caso Termurin) así como una versión del JDK. En la documentación de la Action suelen mostrarse todos los valores posibles para todos los argumentos de configuración. Para este flujo no se designa caché, ya que no se usará ni Maven ni Gradle, y no se considera necesario.

Compilación con Java. Compilación Java básica. Equivalente a llamadas usando una terminal.

```
- name: Compilar con javac
  run: |
    mkdir -p out
    javac -encoding UTF-8 -d out $(git ls-files '*.java')
```

Listing 3.7: Compilación de archivos Java en un flujo de GitHub Actions

Compila todos los `.java` y deposita los `.class` en `out/`. Esto es para seguir estándares, pero se podría compilar in situ.

Corrección. Evaluación de la salida Para la corrección se presentan tres alternativas.

- **Auto-grading con GitHub Classroom.** Ejecuta el contenedor oficial de Classroom, lee la configuración `.github/classroom/autograding.json`, redirige la entrada pública y genera anotaciones verdes/rojas; la calificación aparece en la pestaña *Checks*.

```
- name: Auto-grading (I/O)
  uses: education/autograding@v1
```

Listing 3.8: Corrección usando Autograding en un flujo de GitHub Actions

- **Corrector clásico con `diff`.** Lanza el programa con la entrada de prueba, guarda la salida en un fichero temporal y ejecuta `diff -q` que devuelve `0` si no hay diferencias y `1` si las hay; es la forma más transparente de ver lo que ocurre. Parte de la existencia de los archivos mencionados en el repositorio del alumno. Como se ha comentado, la orientación es hacia cursos introductorios, por lo que es más común que el alumno tenga acceso a los casos para hacer sus prueba.

```
- name: Ejecutar & comparar
  run: |
    java -cp out Main < tests/public/input01.txt > alumno.out
    diff -q alumno.out tests/public/output01.txt
```

Listing 3.9: Corrección básica con Java comparando la salida de ejecutar el programa contra la salida esperada en un flujo de GitHub Actions

- **Acción compuesta propia de corrección.** Encapsula la lógica anterior en una acción reutilizable; permite cambiar los criterios de evaluación sin modificar cada workflow. Es algo más opaca pero solo en apariencia, al tener todo el código fuente en otro repositorio. Al ser más configurable (es una Action en sí misma) se puede procurar generar una gestión de errores o mensajes para el alumno más detallados. En el ejemplo, se dispone de una Action llamada 'corrector-io', dentro de la organización, que gestiona toda la lógica, y admite los campos designados.

```
- name: Corrector I/O (accion propia)
  uses: org/corrector-io@v1
  with:
    input: tests/public/input01.txt
    expected: tests/public/output01.txt
    main-class: Main
```

Listing 3.10: Corrección usando una Action de la organización en un flujo de GitHub Actions

Resumen en el Job Summary

- **Bash incrustado en el workflow.** Los flujos permiten directamente manipular código usando Bash en la acción. Por ejemplo, se puede escribir directamente en `$GITHUB_STEP_SUMMARY`; mostrando un resumen corto a partir del restultado del paso anterior.

```
- name: Resumen (bash)
  run: |
    echo '### Caso `input01`' >> $GITHUB_STEP_SUMMARY
    if [ $? -eq 0 ]; then
      echo 'Todo OK' >> $GITHUB_STEP_SUMMARY
    else
      echo 'La salida no coincide' >> $GITHUB_STEP_SUMMARY
    fi
```

Listing 3.11: Ejemplo de producción de resumen del flujo de GitHub Actions usado scripting tradicional

- **JavaScript con `actions/github-script`.** Alternativamente, se puede usar la API oficial mediante código JavaScript; permite tablas, enlaces e incluso análisis de artefactos sin abandonar el workflow. Dependiendo de la intención, puede resultar más interesante que la anterior. Con fines demostrativos, se presenta cómo quedaría la secuencia de comandos anterior con esta alternativa. Cabe mencionar que se pueden usar para cualquier lógica dentro del flujo, pero aquí se muestran sólo para la presentación de resutlados.

```
- name: Resumen (github-script)
  uses: actions/github-script@v7
  with:
    script: |
      const ok = process.exitCode === 0
      await core.summary
        .addHeading('Caso `input01`')
        .addRaw(ok ? 'Todo OK' : 'La salida no coincide')
        .write()
```

Listing 3.12: Ejemplo de producción de resumen del flujo de GitHub Actions usado scripting por Action (JavaScript)

- **Script externo desacoplado.** El workflow delega la generación del resumen a un script contenido en un fichero interno del repositorio (o externo gestionando su importación, al igual que ocurre con la gestión de pruebas privadas en la siguiente sección (3.3.2), facilitando reutilizar la misma lógica en múltiples proyectos. Análogamente a lo que se mostraba para la correción, en caso de querer usar lógica distribuida, puede ser interesante considerar la creacion de Actions más allá de scripts aislados para facilitar su utilización dentro de la organización.

```
1      - name: Resumen desde script
2        run: scripts/post-summary.sh ${{ steps.evaluacion.outcome }}
```

Listing 3.13: Ejemplo de producción de resumen del flujo de GitHub Actions usado un script externo

3.3.2. Tests JUnit privados

En muchas asignaturas de la escuela, como son 'Programación II', 'Algoritmos y Estructura de Datos', 'Concurrencia', etc., se usa un conjunto de pruebas privadas que determinan la nota del alumno. Con este flujo se pretende mostrar cuál sería el equivalente integrado en GitHub. Es una extensión del flujo anterior para evaluar casos ocultos mediante pruebas JUnit privadas. Por comodidad, se muestra una versión basada en un proyecto Maven para aprovechar la estructura de ficheros así como la gestión de dependencias, pero se podría modificar para no requerir de Maven haciendo uso de una configuración algo más detallada, pero en cualquier caso posible.

Nexo común. Al igual que en el anterior ejemplo, se parte de la definición de trabajos, políticas y pasos. Para evitar redundancias, se evita repetir los fragmentos comunes.

Importación segura de los tests. Desde un repositorio privado de la organización, que contiene los tests.

```
1      - name: Checkout tests privados
2        uses: actions/checkout@v4
3        with:
4          repository: org/tests-privados-p2
5          ssh-key: ${{ secrets.TEST_REPO_KEY }}
6          path: tests_privados
7          persist-credentials: false
```

Listing 3.14: Clonar el repositorio con las pruebas rivadas en el flujo de GitHub Actions

A diferencia del caso anterior, además de clonar el repositorio del alumno en la máquina, se clonan las pruebas.

La clave pública se añade como *deploy key* en el repositorio de tests; la clave privada se guarda cifrada como *secret* en la organización o el repositorio plantilla. GitHub inyecta el secreto únicamente en tiempo de ejecución, lo oculta en los registros y lo borra al finalizar, de modo que el alumnado nunca puede acceder al código de las pruebas ni a la clave.

Setup Java con caché Maven. Ahora, se añade el gestor de dependencias.

```
- uses: actions/setup-java@v4
  with:
    distribution: temurin
    java-version: 21
    cache: maven
```

Listing 3.15: Instalación de Java con caché Maven en el flujo de GitHub Actions

Indicando el tipo de caché usado, en este caso Maven, se instala además esta herramienta. La caché es de interés aquí según lo explicado en secciones anteriores para acelerar descargas de dependencias, y Maven facilita la construcción del proyecto.

Compilación del proyecto. Al establecer Maven, ahora la compilación se realiza usando esta herramienta.

```
- name: Compilar con Maven
  run: mvn -B package -DskipTests
```

Listing 3.16: Compilación con Maven, sin ejecutar tests, en el flujo de GitHub Actions

Se exige un `pom.xml`; si la asignatura no usa Maven, basta con alterar en este punto la lógica de compilación. En este paso sólo se verifica el código del alumno. Si no compila, no es necesario comprobar si supera las pruebas.

Inyección de las pruebas. Copiar las pruebas importadas del repositorio privado a la carpeta de tests para construirlo todo.

```
- name: Copiar pruebas
  run: cp -R tests_privados/*.java src/test/java/
```

Listing 3.17: Copia de las pruebas privadas para su ejecución en el flujo de GitHub Actions

De nuevo, si se desea no usar Maven, sería necesario cambiar este fragmento. En cualquier caso, hay que notificar a los alumnos los nombres reservados de clases para evitar conflictos.

Ejecución de los tests. Ejecutar las pruebas, incluyendo las privadas.

```
- name: Ejecutar pruebas
  run: mvn -B test
```

Listing 3.18: Ejecutar todos los tests usando Maven en el flujo de GitHub Actions

En este punto, se ejecutan las pruebas, tanto las que haya creado el alumno para hacer sus propios casos de prueba, como las que se inyectan desde el repositorio de pruebas de los profesores. Se entiende que a los alumnos se les ha requerido crear ciertas clases con ciertos métodos expuestos para que las pruebas verifiquen su funcionamiento.

Informe Surefire. Dependencia añadida para simplificar la generación de resultados.

```
1     - name: Informe Surefire
2       uses: scacap/action-surefire-report@v1
3       with:
4         github_token: ${{ secrets.GITHUB_TOKEN }}
```

Listing 3.19: Generación de resultados de tests usaando Surefire y Action del Marketplace en el flujo de GitHub Actions

Aprovechando las capacidades de Maven para dependencias, se usa Surefire para generar un informe de las pruebas superadas. Mediante esta acción publicada en el Marketplace de GitHub, se publica el número de tests superados/fallidos en la pestaña *Checks*; y puede ampliarse con un resumen Markdown si se desea.

En este caso se ha prescindido de un resumen personalizado, pero se ha usado Surefire para mostrar de una forma profesional los resultados de los tests. Adicionalmente, sirve como demostración del uso de acciones externas dentro de un flujo, más allá de las oficiales.

3.3.3. Proyecto Maven completo

Para asignaturas que piden una aplicación integral, como 'Programming Project', 'Sistemas Inteligentes', 'Sistemas Orientados a Servicios' e incluso 'Ingeniería del Software II', este flujo asegura que el proyecto compila, ejecuta sus tests con cobertura y genera documentación.

Disparadores. Reducir el número de activaciones del flujo.

```
1 on:
2   push:
3     paths-ignore:
4       - "**/*.md"
5   pull_request:
```

Listing 3.20: Ejemplo de configuración granular de disparadores en el flujo de GitHub Actions

Se vuelve al inicio para demostrar una llamada al pipeline más granular. Al ser proyectos completos, es posible que se realicen cambios sólo a la documentación (comúnmente en lenguaje Markdown), por lo que no queremos que se compruebe la compilación del proyecto completo en estos casos.

Compilación, tests y JaCoCo. Comprobación funcional, a la que se suma la cobertura de las pruebas.

```
- name: Compilar + tests + cobertura
  run: mvn -B verify jacoco:report
```

Listing 3.21: Compilación Maven completa, produciendo un informe de JaCoCo en el flujo de GitHub Actions

Usando Maven y contando con JaCoCo como dependencia, generamos un informe en primera instancia para guardar los resultados de cobertura en `target/site/jacoco`.

Generación de Javadoc. Documentación del código.

```
- name: Generar Javadoc
  run: mvn -B javadoc:javadoc
```

Listing 3.22: Compilación Maven completa, produciendo Javadoc en el flujo de GitHub Actions

En este punto, se genera la documentación del código Java usando Javadoc, que genera un HTML con toda la documentación sobre métodos expuestos. Esto es de especial utilidad cuando se crean librerías de código, pero en general es una buena práctica.

Subida de artefactos. Subir los archivos generados con políticas de retención.

```
- name: Subir artefactos
  uses: actions/upload-artifact@v4
  with:
    name: build-output
    path: |
      target/*.jar
      target/site/jacoco
      target/site/apidocs
    retention-days: 30
```

Listing 3.23: Subir artefactos generados en el flujo de GitHub Actions

Guardar el ejecutable, el informe de cobertura y la Javadoc durante treinta días para revisión posterior. Esto permite tanto a alumnos como a docentes ver los resultados durante 30 días.

Con estos tres flujos se cubre el grueso del espectro de la evaluación docente: desde programas sencillos de E/S, pasando por tests ocultos, hasta la compilación y verificación integral de un proyecto Maven. En el anexo (Sección A.3), se ofrecen los tres workflows completos. Esto se complementa con el repositorio del TFG, también referenciado en el anexo (Sección A.5), donde se muestra una implementación simplificada a modo de prueba de concepto.

3.4. Gestión de errores y robustez

La fiabilidad de un sistema de autograding no se mide solo por los aciertos que detecta, sino por cómo maneja los fallos y cómo comunica esas incidencias a los distintos

actores implicados. La presente sección describe la estrategia adoptada para clasificar errores, contener los recursos, separar la salida estándar del alumno de la del corrector y presentar retroalimentación accionable tanto al estudiante como al profesor. Todos los mecanismos se implementan únicamente con herramientas nativas de GitHub Classroom y GitHub Actions, de modo que el sistema conserva la portabilidad y reduce dependencias externas.

3.4.1. Gobierno de la salida estándar y separación de mensajes

En las primeras prácticas es habitual que el alumnado inunde la salida estándar con `System.out.println` para depurar. Si esos *prints* se mezclan con la salida que el corrector analiza, la comparación falla aun cuando la lógica sea correcta. Para evitar este tipo de falsos negativos se pueden adoptar, entre infinidad de soluciones, dos salvaguardas técnicas más una convención de estilo:

1. **Prefijo reservado `[CORRECTOR].`** Todo mensaje generado por el propio script de evaluación comienza con ese marcador. Antes de comparar resultados, el *workflow* filtra la salida del programa del alumno con:

```
grep -v '^\[CORRECTOR\]' alumno.raw > alumno.clean
```

Listing 3.24: Ejemplo para la diferenciación de salidas estándar programa <>alumno en el flujo de GitHub Actions

   Así sólo se contrastan cadenas emitidas por el código del estudiante. Si el profesorado necesita imprimir algo extra (*logging* interno), debe anteponer también `[CORRECTOR]` para diferenciarlo. En este punto, se pueden hacer más diferenciaciones de estilo, pero se recoge ésta como muestra.

2. **Redirección de flujos a ficheros.** La ejecución se lanza como:

```
java -jar app.jar < input.txt  > alumno.out 2> alumno.err
```

Listing 3.25: Ejemplo de redirección E/S del programa en el flujo de GitHub Actions

   `alumno.out` contiene la salida que se compara con la esperada, mientras que los mensajes de depuración quedan aislados en `alumno.err`. Ambos ficheros se suben como artefactos para que el estudiante los inspeccione si lo desea, sin interferir en la corrección.

3. **Resumen con *Step Summary*.** GitHub Actions permite escribir un informe Markdown visible en la pestaña *Summary*, mediante comandos en el script. En el siguente supuesto, establecemos que se devuelve al flujo un resultado booleano indicando éxito o fracaso de la ejecución (como verdadero o falso) y el tiempo como número. Estos se almacenan en las variables `passes` y `time`, respectivamente. Al mostrar los resultados, se incluyen estas variables. El resultado producido mostrará el resumen con los datos mencionados.

```
echo '## Tests publicos' >> $GITHUB_STEP_SUMMARY
echo "| Caso    | Estado  | Tiempo |" >> $GITHUB_STEP_SUMMARY
echo "|---------|---------|--------|" >> $GITHUB_STEP_SUMMARY
echo "| input01 | $passes | $time  |" >> $GITHUB_STEP_SUMMARY
```

Listing 3.26: Ejemplo de generación de resumen de ejecución usando variables en el flujo de GitHub Actions

También se puede volcar el *Maven Summary* o un contador de pruebas superadas, ofreciendo al alumno una vista clara sin tener que examinar la consola en crudo. Quizás es la técnica más útil, ya que si la correción devuelve cualquier resultado, con una lógica implementada en los *workflows* se puede devolver tanto al alumno como al docente información comprensible

3.4.2. Tipología de fallos y canales de detección

Para que la retroalimentación sea útil, el *pipeline* y corrector podrían clasificar los errores en cuatro tipologías, cada una asociada a un "sensor" distinto dentro de GitHub Actions. Esta segmentación permite tomar decisiones específicas (reintento, cancelación, aviso al profesor) y, sobre todo, mostrar al alumno un diagnóstico que señala el foco real del problema.

Errores de compilación. Si la salida es distinta de `0` al invocar el comando encargado de compilar, se puede abortar inmediatamente y el *Step Summary* recupera las primeras veinte líneas para no saturar la vista. En casos donde el error provenga de `javac`, un `grep` con la etiqueta seleccionada para mensajes de corrector elimina los posibles mensajes internos antes de mostrar el resultado, si se ha seguido esta estrategia para escribir mensajes por consola.

Excepciones en tiempo de ejecución. La captura del `$?` devuelto por la JVM es la vía principal para detectar *NullPointerException*, bucles infinitos o salidas anómalas. No obstante, el sistema puede contemplar otros entornos más allá de Java: en contenedores Docker ejecutados sobre *self-hosted runners* puede activarse la opción `–init` para interceptar `SIGSEGV` y `SIGABRT` de procesos C/C++, o bien redirigir la salida de `pytest` en proyectos Python mediante `pytest –exitfirst`. En todos los casos, los mensajes internos del framework de pruebas llevan prefijos reservados, como se ha visto en el apartado anterior, de modo que el alumno no los confunda con *prints* propios. En el resumen de ejecución, se puede mostrar la clase o binario implicado, la línea de la traza y un *hint* genérico ("revisa índices de arrays / manejo de nulos") para guiar sin revelar el test exacto.

Desajustes E/S. De nuevo, para evitar errores relacionados con la salida, tras ejecutar `java -jar app.jar < input.txt > alumno.out`, el *diff* se realiza sobre `alumno.out`. Si los mensajes de salida del alumno se muestran en `alumno.err` y se cargan como artefacto aparte, se evita que el *diff* evalúe texto añadido por `System.out.println`.

Consumo excesivo de recursos. Los *runners* de GitHub imponen de fábrica un límite genérico (aproximadamente 7 GB de RAM y 14 GB de espacio en disco, que provoca `OOM Killed` o un `GC overhead limit` cuando la JVM –o cualquier otro runtime– rebasa la cuota. En el caso de los *self-hosted* podemos endurecer todavía más el contenedor que ejecuta la práctica: basta con declarar en el `job` un bloque `container` y pasar opcionalmente `options: "–memory 512m"`; GitHub lo traslada

al atributo `memory.max` de *cgroup2*, de modo que la aplicación se termina tan pronto como sobrepasa ese tope. El mismo mecanismo admite `–pids-limit` para frenar *fork-bombs* (`pids.max`) y `–memory-swap` para evitar que el alumno colapse el nodo saturando la *swap*. Si se trata de abusos de E/S (p. ej. escritura ilimitada de ficheros temporales) se puede montar el directorio de trabajo en un *tmpfs* con tamaño fijo o aplicar `–device-write-bps` en runners Docker. Cuando cualquiera de estos límites salta, GitHub marca el paso como `Cancelled`; el corrector captura el evento y añade al *Step Summary* un mensaje del tipo "Memoria o PIDs máximos alcanzados, revisa las estructuras de datos o procesos en segundo plano", de forma que el estudiante identifique rápidamente que el problema es de consumo de recursos y no de lógica.

3.4.3. Contención de recursos: límites, reintentos y cancelaciones

La bolsa de minutos gratuita que GitHub Education concede a la organización es finita. Para evitar que un *commit* defectuoso o un bucle infinito, agote la cuota o bloquee los *runners*, cada *workflow* incorpora una capa de autocontrol sustentada únicamente en la sintaxis oficial de Actions:

Límites de tiempo (`timeout-minutes`). Cada *job* declara un tope de tiempo para compilación y para pruebas. Cuando el tiempo expira, GitHub detiene el contenedor y marca el paso como `timeout`. De esta forma, se liberan más rápido *runners* ante entregas fallidas.

Reintentos acotados. Para fallos transitorios (descargas de dependencias, límites de uso de API) se aplica:

```
1 if: failure() && github.run_attempt < 3
```

Listing 3.27: Estrategias de reintento en el flujo de GitHub Actions

De esta forma, el paso se reejecuta un máximo de dos veces adicionales. La variable `run_attempt`, propia de GitHub y que no requiere configuración adicional, puede ayudar a prevenir bucles infinitos de este modo.

Concurrencia y cancelación de ejecuciones obsoletas. Previene malgastar minutos de ejecución.

```
1 concurrency:
2   group: ${{ github.ref }}
3   cancel-in-progress: true
```

Listing 3.28: Gestión de concurrecia de disparadores en el flujo de GitHub Actions

La directiva anterior aborta el *workflow* anterior si llega un *push* nuevo a la misma rama, liberando minutos y *runners*. En repositorios antiguos que carecen de esta opción, se emplea la acción comunitaria *Cancel Previous Workflow Runs*, por lo que ambas alternativas pueden ser implementadas.

Cuotas organizativas. El panel *Settings* → *Actions* → *Usage* muestra minutos y almacenamiento consumidos. Se puede establecer un *cron job* diario que consulta la API REST y, si una práctica supera el 80 % de la bolsa asignada, envía aviso al coordinador del curso para que revise la configuración y cachés.

Con este cuádruple cinturón de seguridad se mantiene el consumo mensual estable y se evitan incidencias puntuales que puedan comprometer el servicio para el resto del alumnado.

3.4.4. Presentación de resultados al alumno

El alumnado no tiene por qué repasar un *log* de 300 líneas para localizar un error. El sistema puede destilar la retroalimentación en tres niveles crecientes de profundidad:

Vista "Checks". Cada *job* aparece con un ícono verde o rojo y un enlace directo al paso fallido. Es la primera alerta visual que GitHub proporciona de serie. Esto aporta información general acerca de compilación y pasos grandes, como resultados ternarios (pasa, falla o abortado), por lo que esto abre las puertas a casos más triviales de corrector. Aquellos en los que se busca una compilación usando Maven o situaciones en las que el alumno quiere comprobar que todos sus tests de prueba pasan, la información producida en este paso será útil.

Resumen en *Step Summary*. El corrector puede generar una tabla Markdown accesible desde la pestaña *Summary*, sin necesidad de abrir la consola. El ejemplo mostrado en la Sección 3.4.1 ilustra una casuística que aplica aquí. Una más práctica, pero elaborada, consistiría en hacer que el corrector devuelva como resultado un documento Markdown, de forma que su contenido se pueda redirigir a `GITHUB_STEP_SUMMARY` y, con ello, tener un resumen más completo y detallado para el alumno.

Comentario en la *Pull Request*. Cuando la asignatura exige PR final, un *step* extra ejecuta:

```
gh pr comment $PR_URL \
  --body "2/8 tests fallaron -> revisa Summary."
```

Listing 3.29: Ejemplo de generación de mensajes en una Pull Request durante el flujo de GitHub Actions

Un estudio certifica que el *feedback* incrustado en la conversación de la PR acelera la corrección en el siguiente *commit* [35]. De esta forma, además de ofrecer el log completo y un resumen de los resultados, se notifica al alumno de estos resultados.

Artefactos descargables. Los ficheros `alumno.out`, `alumno.err` e informes de cobertura, como los generados por Jacoco en (`site/jacoco/index.html`) si se usa este plugin, se publican como artefactos y permanecen 30 días:

```
- name: Publicar artefactos
  uses: actions/upload-artifact@v4
  with:
    name: salida-y-cobertura
```

```
5     path: |
6       alumno.out
7       alumno.err
8       target/site/jacoco/index.html
9     retention-days: 30
```

Listing 3.30: Muestra de subida de artefactos personalizados en el flujo de GitHub Actions

Así, el estudiante puede revisar diferencias con calma, incluso sin conexión.

3.4.5. Síntesis y monitorización para el profesorado

Para el equipo docente es vital disponer de un cuadro de mando que resuma el estado del curso sin perder la posibilidad de inspeccionar un caso concreto al instante. La estrategia se apoya en tres pilares complementarios:

Panel integrado de GitHub Classroom. La pestaña *Assignments* muestra, alumno por alumno, la puntuación del autograding y un enlace directo a su repositorio. Con un clic en *Download CSV* obtenemos un fichero con identificador, fecha de entrega y nota. Ese CSV se importa en Google Sheets y un pequeño script de *Apps Script* colorea cada fila según el número de fallos o el porcentaje de tests superados.

Pull Requests como canal de soporte contextual. Los beneficios de usar pull requests se van acumulando según lo descrito en las secciones anteriores. Un beneficio adicional, al haber una conversación dentro del contexto de la pull request, es el permitir que un alumno consulte dudas a un profesor mencionándolo (`@profesor`) cuando necesite ayuda. El docente, con acceso completo al repositorio, revisa el código exacto que originó la duda—sin correos, sin adjuntos desactualizados y con todo el historial de *commits* a la vista. Esta conversación queda archivada en la propia PR, de manera que el feedback futuro parte del mismo contexto técnico.

Digest diario automático. Un *workflow* programado puede consultar la API REST de ejecución de *workflows* y enviar al profesorado un correo Markdown con el balance del día: *"Hoy: 42 entregas nuevas · 10 sin fallos · 5 con errores de compilación."* Así se detectan picos de incidencias o prácticas atascadas y se priorizan las tutorías. Si bien ejecutar esto diariamente puede carecer de interés salvo para controlar el uso, con planificaciones más complejas, se pueden llegar a generar reportes o estadísticas de uso al finalizar una tarea del Classroom.

Esta triple capa reduce la fatiga de inspeccionar logs uno por uno y permite focalizar los esfuerzos donde el alumnado realmente los necesita.

3.5. Escalabilidad y despliegue

Partimos de una proyección puramente hipotética (todavía no se ha medido en cursos reales) basada en 120 estudiantes por asignatura y una media de 6 a 8 commits significativos por práctica. Además, aunque falta validar con datos reales, se espera un comportamiento típico de calendario académico: la mayor parte de los commits llega en las dos semanas previas a la entrega.

3.5.1. Topologías de runners: cloud vs. on-premise

GitHub Actions distingue dos tipos de ejecutores, cada cual con sus pros y contras:

GitHub runners. Contenedores efímeros (Ubuntu, macOS, Windows) que GitHub aprovisiona en segundos. Ventajas: arranque casi instantáneo, mantenimiento cero y bolsa de minutos gratuita para cuentas Education. Limitaciones: no se pueden instalar *drivers* específicos (GPU, FPGA) y la organización depende al 100 % de la cuota de minutos.

Self-hosted *runners*. Máquinas físicas o virtuales que gestiona la propia universidad, registradas mediante token y etiquetadas para su selección en los YAML. Ventajas: control pleno sobre hardware y sistema, acceso a red interna y posibilidad de montar cachés locales persistentes. Inconvenientes: mantenimiento, seguridad y escalado manual.

En la práctica y siempre sobre estimaciones preliminares para *glsworkloads* ligeros (-10 min por commit) la opción *cloud* suele salir más barata, ya que el consumo queda absorbido dentro del *tier* gratuito.

La mayoría de las asignaturas emplean Java como lenguaje vehicular y construyen con Maven o Gradle en los casos más complejos. Ambas herramientas se benefician de la acción oficial `setup-java`, que prepara el entorno y cachea dependencias por defecto, de modo que la parte pesada (descarga de artefactos) se paga una sola vez:

```
- name: Configurar Java y Maven cacheado
  uses: actions/setup-java@v4
  with:
    distribution: 'temurin'
    java-version: '21'
    cache: 'maven'   # o 'gradle'
```

Listing 3.31: Instalación de Java con caché para acelerar ejecuciones futuras del flujo de GitHub Actions

Con esta configuración, las ejecuciones típicas —compilación, tests y análisis— rondan el minuto, salvo que aparezcan tiempos de espera (*timeout*) por errores de programación. Para la mayoría de los casos de uso, la infraestructura GitHub-hosted cubre las necesidades sin coste adicional; los *self-hosted* quedan inicialmente reservados a prácticas que requieran hardware específico o tiempos de ejecución prolongados.

3.5.2. Optimización de minutos y almacenamiento

Escalar no implica despreocuparse de la eficiencia. Se destacan cuatro capas de optimización.

Caching de dependencias. Uso de la Action actions/cache con claves derivadas de pom.xml permite evitar descargas repetidas; GitHub estima reducciones del 60 % en tiempo de compilación. En las versiones más actualizadas de las acciones de configuración oficiales, ya se hace el caching de dependencias por defecto.

Concurrencia y cancelación de runs obsoletos. Se puede configurar cómo debe actuar el flujo ante entregas subidas casi de manera simultánea. Esta directiva, presentada durante la descripción de los flujos y en la Sección 3.4.3, es configurable en

los flujos, aborta el workflow anterior cuando el alumno hace un nuevo push. Esto ahorra minutos y evita colas fantasma. Anteriormente, se hacía uso de la acción oficial 'Cancel Previous Workflow Runs', que utiliza la API descrita en 'Canceling a workflow', pero al comenzar desde cero, se pueden seleccionar las configuraciones más modernas.

TTL de artefactos. El tiempo de vida (TTL) de los artefactos se puede definir. Siempre por debajo del máximo (400 días), con un TTL recomendado de 14 días, libera el almacenamiento tras dos semanas, período suficiente para la revisión.

Matrix builds con *fail-fast*. Cuando la misma práctica debe verificarse en varias versiones de JDK (por ejemplo 8 y 21), en distintos sistemas operativos o con y sin *flags* de optimización, puede usarse la estrategia `matrix`:

```
strategy:
  fail-fast: true
  matrix:
    java: [ '8', '21' ]
    os:   [ ubuntu-latest ]
```

Listing 3.32: Estrategia de matriz para ejecución simultánea en varios entornos dentro del mismo flujo de GitHub Actions

GitHub ejecuta los ejes en paralelo y, si uno falla, la opción `fail-fast:true` cancela el resto para no desperdiciar minutos. En prácticas donde el enunciado fija una versión concreta del toolchain, la matriz carece de sentido; no obstante, resulta valiosa en asignaturas avanzadas que comparan portabilidad entre entornos o miden regresiones multi-JDK. Además, el alumno visualiza de un vistazo en la pestaña *Checks* qué combinaciones han pasado o caído, reforzando la mentalidad de pruebas cruzadas.

Capítulo 4

Impacto del trabajo

La automatización de la corrección de prácticas con GitHub Classroom + Actions genera un impacto positivo neto en los ámbitos personal, social, empresarial, económico y medioambiental, con efectos culturales limitados. Contribuye directamente a los ODS 4 (Educación de calidad), 9 (Industria, innovación e infraestructura), 12 (Producción y consumo responsables) y 13 (Acción por el clima), y de forma secundaria al 8 (Trabajo decente y crecimiento económico). Las decisiones de diseño (código abierto, flujos CI/CD, retroalimentación inmediata y repositorios sin papel) se justifican justamente por maximizar estos impactos y mitigar riesgos.

4.1. Impacto general

El impacto global de este Trabajo Fin de Grado se manifiesta, en primer lugar, en la **esfera personal**: la retroalimentación casi instantánea que ofrece el sistema fomenta un aprendizaje más profundo y autónomo de los estudiantes, al mismo tiempo que libera al profesorado de tareas repetitivas y le permite centrarse en la mentoría. Esta combinación refuerza la motivación de ambos colectivos, pues introduce prácticas propias del mundo profesional -control de versiones, integración continua y revisiones de código— directamente en el aula.

En el **plano social**, la homogeneidad y la inmediatez del feedback elevan la equidad docente porque eliminan tres focos clásicos de desigualdad. Primero, la variabilidad entre correctores: en un experimento con 28 evaluadores de prácticas de Java, la fiabilidad inter-rater reliability cayó a $\alpha \approx 0{,}2$ y las mismas entregas llegaron a diferir casi dos puntos según quién las calificara, prueba de que las rúbricas no bastan para garantizar coherencia. Segundo, los sesgos cognitivos del evaluador —el halo effect—, donde la impresión previa sobre un estudiante (por ejemplo, su rendimiento pasado o su perfil) contamina la nota de trabajos posteriores; los estudios recomiendan el anonimato precisamente para neutralizar este sesgo. Y tercero, las disparidades entre grupos en la rapidez y profundidad de la retroalimentación: los sistemas automáticos entregan resultados idénticos y casi instantáneos a todos los estudiantes, evitando las diferencias que surgen cuando cada sección depende del estilo o la disponibilidad de su ayudante. De este modo, esta infraestructura de corrección automatizada no solo agiliza el aprendizaje, sino que nivela en cierta medida el terreno de juego para todo el alumnado.

Aunque el **impacto empresarial** no es el foco principal, resulta significativo: las empresas se benefician de egresados que ya dominan flujos de trabajo DevOps, lo que reduce la curva de incorporación y mejora la productividad temprana. Además, el proyecto impulsa el ecosistema de herramientas abiertas —acciones de GitHub, linters y analizadores estáticos— generando oportunidades de consultoría y servicios especializados.

Desde una **perspectiva económica**, la automatización disminuye el coste asociado a la corrección manual de prácticas, evita la posible contratación de ayudantes para tareas rutinarias y, a medio plazo, eleva la eficiencia global del sistema educativo. A medida que se generalice su adopción, se producirá un retorno derivado de una fuerza laboral mejor preparada en competencias digitales avanzadas.

El **impacto medioambiental**, aunque a veces se pase por alto, es tangible: al eliminar entregas impresas y trasladar toda la interacción a repositorios en línea, se reduce de forma drástica el consumo de papel y la logística asociada. Asimismo, la ejecución de los flujos de integración continua en la nube permite escalar los recursos bajo demanda y evita infraestructuras locales sobredimensionadas, con la consiguiente mejora en eficiencia energética.

Por último, el **componente cultural** es más limitado: el trabajo promueve la cultura y el uso de software libre dentro del entorno académico, pero su influencia más allá de la comunidad universitaria queda en segundo plano frente a los ámbitos anteriores.

Todas las decisiones de diseño —desde optar por repositorios individuales y pruebas automáticas, hasta exigir entregas exclusivamente digitales y publicar plantillas abiertas— se han tomado con estos impactos en mente: maximizar los beneficios formativos y sociales, reducir costos y huella ecológica, y transferir competencias reales al alumnado.

4.2. Objetivos de Desarrollo Sostenible

La Agenda 2030 de Naciones Unidas articula 17 Objetivos de Desarrollo Sostenible (ODS) para mejorar en lo social, económico y ambiental. Este TFG, al digitalizar la evaluación de prácticas y transferir competencias DevOps al aula, contribuye de forma tangible a varios de esos objetivos. A continuación se describen los ODS más relevantes y la manera en que la solución los refuerza.

ODS 4 – Educación de calidad. Sustituir la corrección manual por baterías de tests automatizados ofrece retroalimentación inmediata y objetiva, elevando la calidad del aprendizaje y reduciendo diferencias entre grupos. La práctica habitual con Git y CI/CD en las asignaturas acerca la formación universitaria a las demandas del sector.

ODS 8 – Trabajo decente y crecimiento económico. Familiarizar a los estudiantes con flujos DevOps desde el aula aumenta su empleabilidad en un mercado donde la demanda de estos perfiles sigue creciendo de forma sostenida. El resultado es una incorporación más rápida y productiva al grueso empresarial.

ODS 9 – Industria, innovación e infraestructura. La integración continua y los repositorios distribuidos refuerzan la infraestructura digital de la universidad y fo-

mentan la innovación docente, alineándose con la meta de construir infraestructuras resilientes y de promover la industrialización sostenible. Liberar las plantillas como software libre anima a la comunidad a reutilizar y mejorar la solución, creando un ciclo virtuoso de innovación.

ODS 12 – Producción y consumo responsables. Al exigir entregas exclusivamente digitales, el proyecto elimina el uso de papel y la logística asociada, reduciendo residuos y huella material. La reutilización de plantillas y workflows comunes, además, evita la duplicación de recursos.

ODS 13 – Acción por el clima. Desmaterializar las entregas y ejecutar la compilación en la nube —escalando recursos solo cuando se necesitan— evita emisiones derivadas del transporte y del sobreaprovisionamiento de hardware local. Al mismo tiempo, sensibiliza al alumnado sobre la relación entre prácticas digitales y sostenibilidad climática.

Capítulo 5

Resultados y conclusiones

5.1. Resultados

A continuación se presentan los resultados obtenidos tras el diseño, implementación y validación del sistema propuesto. Se evalúa el grado de cumplimiento de los objetivos planteados inicialmente, así como el impacto funcional y pedagógico del prototipo desarrollado. Aunque no se ha desplegado en entornos reales, se han reproducido condiciones operativas mediante entornos simulados que permiten extraer conclusiones sólidas sobre su viabilidad técnica y utilidad docente.

5.1.1. Evaluación de los objetivos planteados

En cuanto a los resultados obtenidos, el proyecto ha cumplido de forma satisfactoria los objetivos formulados en la sección 1.3. El primer objetivo, definir un esquema de gestión de prácticas sustentado en GitHub y GitHub Classroom, se materializa en una estructura de repositorios y plantillas que ya pueden adoptarse sin modificaciones para los casos estudiados. La organización propuesta, junto con la creación de guías de uso para docentes y alumnado a partir de los conocimientos que se pueden extraer de este Trabajo de Fin de Grado, resuelve los problemas iniciales de dispersión de entregas y facilita un flujo homogéneo desde la creación de la práctica hasta la recepción de la solución.

El segundo objetivo, centrado en el diseño e implementación de un sistema de corrección automatizada soportado por pipelines CI/CD, también se ha alcanzado plenamente. Los workflows desarrollados ejecutan de forma idempotente baterías de pruebas y generación de informes de resultados, todo ello en un tiempo estimado de ejecución inferior a dos minutos por entrega, para la mayoría de los casos. La configuración paramétrica de GitHub Actions, con matrices de versiones de JDK, gestión de secrets y aprovechamiento de cachés, impulsa el uso de estas herramientas en un nivel de robustez que normalmente se asocia a proyectos industriales, acercando al alumno a buenas prácticas profesionales.

Respecto al tercer objetivo, documentar y validar el sistema, se optó por una validación en entorno simulado, decisión estratégica que antepuso la seguridad del alumnado a la exposición temprana de un prototipo. El banco de pruebas sintético reproduce las casuísticas previstas (compilación, fallo lógico, fuga de memoria,

timeout), ofreciendo métricas representativas. Esta elección garantiza que la primera implantación real se realice con una versión pulida, evitando interferencias en la docencia vigente sin comprometer la rigurosidad de la evaluación.

5.1.2. Resultados funcionales

A nivel funcional, el sistema proporciona corrección instantánea tras cada push, almacenamiento de artefactos (logs, informes JUnit, cobertura) y comentarios automáticos en la pull request. La arquitectura se ha construido en módulos claramente aislados: plantillas, lógica de evaluación y orquestación. Esto se ha hecho para facilitar la extensión a otros lenguajes de programación o a prácticas con requisitos especiales. Asimismo, las plantillas ofrecen puntos de anclaje versionados, de modo que un docente puede fijar su asignatura a una versión concreta mientras el repositorio principal evoluciona.

5.1.3. Beneficios esperados

Los beneficios esperados se concretan en tres ejes: (1) reducción drástica de la carga docente al delegar las comprobaciones repetitivas en el pipeline; (2) retroalimentación casi inmediata para el estudiante, que puede iterar sobre su solución en cuestión de minutos; y (3) adopción temprana de buenas prácticas profesionales, como el control de versiones, pruebas automatizadas y despliegue continuo, integradas de forma natural en la experiencia académica. En conjunto, los resultados posicionan este trabajo como una base sólida para su implantación futura y un punto de partida para futuras líneas de mejora descritas en el capítulo 5.

5.2. Conclusiones personales

5.2.1. Aprendizajes técnicos

Uno de los aprendizajes más importantes ha sido dominar las posibilidades de personalización y automatización que ofrece GitHub Actions, lo que me ha permitido construir *workflows* adaptados al contexto educativo con precisión y robustez. Este dominio no se limita a la mera edición de archivos YAML: implica comprender el ciclo completo de autenticación mediante *tokens* y *secrets*, optimizar tiempos de ejecución y desplegar matrices de prueba que repliquen entornos heterogéneos. Al afinar parámetros como `concurrency` o `timeout`, he interiorizado conceptos de tolerancia a fallos y *rate limiting* que rara vez se ven con tanta claridad en prácticas estándar.

En paralelo, integrar los tests ha reforzado la disciplina de escribir pruebas idempotentes y reproducibles. Pocas veces un proyecto académico me había dado la oportunidad de hilar, en un mismo hilo conductor, conocimientos de Programación, Sistemas Operativos y Distribuidos con técnicas de ingeniería de software profesional.

5.2.2. Aprendizajes profesionales y de buenas prácticas

Este trabajo ha requerido una aproximación práctica al diseño de soluciones útiles para múltiples perfiles (docente y alumno). Documentar para otros ha dejado de ser un trámite para convertirse en parte esencial de la experiencia: tanto la memoria como cada ejemplo de plantilla debían servir para que un futuro profesor comprendiera

el sistema sin necesidad de intérprete. Este ejercicio de empatía técnica me ha forzado a pulir la comunicación escrita y a priorizar la claridad sobre la exhaustividad.

5.3. Trabajo futuro

Con las piezas descritas a continuación, se pretende que este Trabajo no se quede en la foto fija de hoy, sino que se prepare el terreno para prácticas más complejas, datos de uso precisos y un ecosistema que se actualiza solo cuando las dependencias lo exigen.

5.3.1. Reutilización y modularidad de plantillas

Para abrir la puerta a nuevas asignaturas y a prácticas técnicamente más exigentes, el repositorio de plantillas para los docentes se organiza bajo un directorio `src/`. La idea es simple: el núcleo común a todas las prácticas (*setup*, `concurrency`, etiquetado de *runners*, etc.) se mantiene en plantillas para facilitar su reutilización, mientras que lógicas individuales de lenguajes, como archivos necesarios para la compilación, se mantienen agrupadas por lenguajes. Esto pretende facilitar la creación de nuevas plantillas para prácticas de cualquier índole. Un árbol de la jerarquía quedaría tal que así:

```
plantillas/
+-- src/        # Contiene scripts por lenguaje y funcionalidades extra
|   +-- bash/
|   |   +-- script.sh
|   |   `-- bash.yml
|   +-- c/
|   |   +-- Makefile
|   |   `-- c.yml
|   +-- python/
|   |   +-- requirements.txt
|   |   `-- py.yml
|   +-- java/             # Bases de plantilla java en distintos lenguajes
|   |   +-- iojava.yml
|   |   +-- mvnjava.yml
|   |   `-- testsjava.yml
|   `-- common/
|       `-- template1.yml
|       `-- template2.yml
+-- docs/                 # Documentacion
+-- used/                 # Historico de flujos utilizados
|   +-- practica-aed-2425.yml
|   +-- practica-concurrencia-2425.yml
|   `-- ...
`-- README.md             # Archivo de informacion del repositorio
```

Como podemos observar, el árbol contiene lo descrito, incorporando una carpeta para documentación interna así como una carpeta `used` donde guardar el histórico de acciones usadas. A medida que este historial aumente, se debe considerar clasificarlas, idealmente por asignatura para conservar el contexto.

Además, conviene versionar las plantillas empleando *tags* semánticos (`vX.Y.Z`) y un

fichero `CHANGELOG.md`. De esta forma, un docente puede "anclar" su práctica a la versión exacta que validó en su momento, mientras el repositorio principal avanza sin riesgo de romper entregas antiguas. Cuando sea necesario un cambio incompatible (por ejemplo, de JUnit 5.10 a 5.12), se libera una versión mayor, dejando constancia del cambio.

5.3.2. Casos más complejos y privilegios elevados

A partir de tercero surgen prácticas que exigen capacidades distintas del *pipeline* estándar: necesitan privilegios de `root`, manipulan `cgroups` o requieren varios procesos cooperando. Para estos escenarios se recomienda recurrir a **runners self-hosted privilegiados**, etiquetados `privileged=true`; los GitHub-hosted quedan descartados porque no permiten lanzar contenedores con `–privileged`.

A continuación, se presentan dos casos de prácticas de asignaturas de grado, cuyas prácticas (entre 2021 y 2025) requieren de trabajo adicional para poder ser corregidas con el sistema desarrollado en el Capítulo 3.

Administración de Sistemas. El alumno desarrolla un intérprete de comandos en C, a modo de mini-shell que crea procesos, encadena tuberías y gestiona la memoria. Para evaluar la entrega con GitHub Actions, se aconseja seguir un escenario como el propuesto a continuación:

1. Desplegar un contenedor efímero con `docker run –privileged –memory 1g –pids-limit 512`.
2. Compilar con `gcc` y ejecutar la batería de pruebas bajo `valgrind`; las fugas se vuelcan a `valgrind-report.txt`.
3. Añadir un `grep -c 'definitely lost'` que puntúe la sección "memoria".
4. Mantener `timeout-minutes: 4` y `concurrency: cancel-in-progress:true` para evitar runners bloqueados.

Sistemas Distribuidos. Java RMI o sockets C. Se trata de una práctica en la que se construye y estudia una arquitectura cliente–servidor con Java RMI o con sockets tradicionales en C. En esta práctica, el alumnado debe reproducir todo el ciclo distribuido: compilar el servidor, registrarlo en el *registry* RMI (o abrir un `listen socket`); lanzar el cliente, localizar el servicio remoto e invocar métodos; capturar excepciones remotas y desconexiones; cerrar la sesión de forma ordenada. Para validar esos requisitos, se recomienda orquestar la entrega con `docker-compose`:

1. **Definición de servicios.** El `docker-compose.yml` declara dos contenedores, `server` y `client`, basados en una imagen Java 21 o un `gcc:bullseye` para sockets C.
2. **Arranque controlado.** El servicio `server` incluye un `healthcheck` que sondea `tcp://localhost:1099` (RMI) o el puerto que el alumno publica mediante variable de entorno. El *workflow* espera a que dicho *healthcheck* devuelva `healthy` antes de iniciar el contenedor `client`; así se evitan fallos.

3. **Captura de trazas independientes.** Ambos procesos dirigen su `stdout/stderr` (salida de error) a `/logs/server.log` y `/logs/client.log`; el corrector sube esos ficheros como artefactos para que el estudiante los consulte si algo va mal.
4. **Análisis automático.** Finalizada la ejecución, el script examina los códigos de salida:
   - `0/0`: éxito total.
   - `1/0`: «cliente abortó → revisa manejo de excepciones».
   - `0/1`: «cliente OK, servidor falló en . . . »

   El resumen se vuelca al *Step Summary* y, si la práctica exige *pull request*, se añade un comentario con el mismo diagnóstico.
5. **Contención en el uso de recursos.** Se mantienen los límites `timeout-minutes:` y `concurrency:{cancel-in-progress:true}` descritos más arriba: ningún runner queda bloqueado aunque el servidor entre en un bucle, y el uso de minutos permanece dentro de la bolsa gratuita.

Cabe destacar que los escenarios descritos anteriormente no han sido aún validados mediante pruebas reales. Por tanto, las estrategias y recomendaciones propuestas para implementar el sistema de corrección automática tienen un carácter hipotético e ilustrativo, y podrían requerir ajustes significativos al trasladarlas a un contexto real. El objetivo principal es proporcionar una base conceptual sobre el funcionamiento previsto y señalar posibles vías para futuras validaciones experimentales.

Contención y métricas. Se aconseja lanzar los contenedores con los límites heredados del runner: `memory.max = 1 GiB`, `pids.max = 512`, `io.max = 50 MB/s`. Si se supera alguno, el contenedor finaliza con el código 137 y el *workflow* lo etiqueta como *resource-exceeded*, adjuntando `resource-report.txt` con `memory.current`, `pids.current` y uso de disco.

Aceleradores de tiempo. Para reducir minutos de ejecución se sugiere cachear dependencias mediante `actions/cache` y utilizar imágenes base preconstruidas (por ejemplo, `ghcr.io/etsinf/so-base:latest`) que ya incorporen herramientas y bibliotecas comunes como `openjdk`, `valgrind`, o entornos específicos para C o Java. Esta estrategia evita tiempos repetidos de instalación y descarga, acelerando considerablemente el tiempo de ejecución de los workflows.

Instrumentación ligera. Se recomienda capturar consumo de CPU/RAM y número de `syscalls`; los valores se publican en el *Step Summary*. Así, el docente dispone de pruebas objetivas para penalizar soluciones con *busy wait* o fugas de recursos, sin inspeccionar manualmente cada log.

5.3.3. Análisis estático y calidad extra

Siguiendo la filosofía de la asignatura *Programming Project*, se puede añadir un paso opcional de JaCoCo o SonarQube que no afecta a la nota funcional, pero sí devuelve una foto de cobertura, deuda técnica y vulnerabilidades. El alumno, si quiere ir más allá de pasar las pruebas, o se instaura como requerimiento de alguna práctica la

necesidad de pasar la cobertura o calidad marcada, obtiene datos para pulir su código a estándar industrial.

Para los proyectos en C/C++, `cppcheck` o `clang-tidy` pueden integrarse de forma análoga, generando informes en formato SARIF que GitHub muestra como anotaciones en la propia *pull request*. De esta forma, un estudiante detecta `undefined behaviour` o fugas de memoria sin abandonar la plataforma.

En Java, se puede complementar JaCoCo con `SpotBugs` y su extensión *sbom* para producir un "bill of materials" que documente dependencias de terceros.

5.3.4. Diagnóstico de uso y dimensionamiento futuro

Aunque el panel de GitHub muestra los minutos consumidos, se puede buscar algo más fino: un script que llame a la API de *usage* y genere un CSV por práctica. Con ese histórico podremos decidir cuántos *runners* adicionales (o de qué tipo: Windows, macOS, GPU) hay que añadir en próximas iteraciones de uso.

Dependabot de apoyo. Activar Dependabot sobre los *workflows* y los scripts de soporte ayuda a cerrar vulnerabilidades en cuanto aparecen, sin que tengamos que vigilar manualmente versiones de JUnit, Maven Surefire, `actions/setup-java`, etc. Eso sí, cuando una práctica depende de una versión concreta (por ejemplo, JDK 21 y JUnit 5.10) se fija en el `pom.xml` para que los cambios automáticos no rompan la corrección.

Para la parte de dimensionamiento, se contempla la opción de **runners autohospedados** en el Centro de Cálculo. Un *daemon* de Kubernetes escala los pods de runners en función del número de jobs en cola, manteniendo un mínimo de dos instancias en horas valle y ampliando en picos de entrega. Esta arquitectura híbrida — gratuitos de GitHub + propios — garantiza costos controlados y tiempos de espera razonables.

Bibliografía

[1] M Nurminen y H Järvinen. *Having It All: Autograders Reduce Workload yet Increase the Quantity and Quality of Feedback*. 2021. URL: https://cris.tuni.fi/ws/portalfiles/portal/62016214/SEFI_2021_Having_it_all_final.pdf (visitado 11-03-2025).

[2] Samiha Marwan et al. «Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science». En: (ago. de 2020), págs. 194-203. DOI: 10.1145/3372782.3406264. URL: https://dl.acm.org/doi/10.1145/3372782.3406264 (visitado 18-03-2025).

[3] Davide Fucci et al. «A Longitudinal Cohort Study on the Retainment of Test-Driven Development». En: *arXiv* 1 (2022). URL: https://arxiv.org/pdf/1807.02971.pdf (visitado 11-03-2025).

[4] Francisco Calles-Esteban et al. «Influence of Gamification on the Commitment of the Students of a Programming Course: A Case Study». En: *Applied Sciences* 14 (abr. de 2024), págs. 3475-3475. DOI: 10.3390/app14083475. URL: https://www.mdpi.com/2076-3417/14/8/3475 (visitado 20-03-2025).

[5] Ryan Hecht et al. «Distributing, Collecting, and Autograding Assignments with GitHub Classroom». En: (mar. de 2023), págs. 1179-1179. DOI: 10.1145/3545947.3569627. URL: https://dl.acm.org/doi/10.1145/3545947.3569627 (visitado 10-03-2025).

[6] Indrabudhi Lokaadinugroho y Burhanudin Burhanudin. «DevOps CICD in Higher Education». En: (mayo de 2022). DOI: 10.21203/rs.3.rs-1500352/v2. URL: https://www.researchgate.net/publication/360381031_DevOps_CICD_in_Higher_Education (visitado 18-03-2025).

[7] Maria Augusta Nelson y Lesandro Ponciano. *CSDL | IEEE Computer Society*. Computer.org, 2021. URL: https://www.computer.org/csdl/proceedings-article/seeng/2021/319600a031/1v56fVuFna0 (visitado 15-03-2025).

[8] Manel Mena et al. «Applying GitHub Services to Support Teaching-learning Strategies in Computer Science Courses». En: (ene. de 2022), págs. 289-296. DOI: 10.5220/0011071000003182. URL: https://www.researchgate.net/publication/360325468_Applying_GitHub_Services_to_Support_Teaching-learning_Strategies_in_Computer_Science_Courses (visitado 25-02-2025).

[9] Marcus Messer et al. *How Consistent Are Humans When Grading Programming Assignments?* arXiv:2409.12967 [cs.HC]. 2024. arXiv: 2409.12967. URL: https://arxiv.org/abs/2409.12967 (visitado 11-03-2025).

[10] John M. Malouff, Ashley J. Emmerton y Nicola S. Schutte. «The Risk of a Halo Bias as a Reason to Keep Students Anonymous During Grading». En: *Teaching of Psychology* 40 (mayo de 2013), págs. 233-237. DOI: 10.1177/

0098628313487425. URL: https://journals.sagepub.com/doi/10.1177/0098628313487425 (visitado 11-03-2025).

[11] Nathaniel Woodthorpe. *GitHub Classroom Gets a Reliability and Performance Boost*. The GitHub Blog, ago. de 2019. URL: https://github.blog/news-insights/github-classroom-gets-a-reliability-and-performance-boost/ (visitado 14-03-2025).

[12] Pedro Alves. *Automated Assessment in Mobile Programming Courses: Leveraging GitHub Classroom and Flutter for Enhanced Student Outcomes*. Arxiv.org, abr. de 2019. URL: https://arxiv.org/html/2504.04230v1 (visitado 14-03-2025).

[13] Chris Wilcox. «The Role of Automation in Undergraduate Computer Science Education». En: (feb. de 2015), págs. 90-95. DOI: 10.1145/2676723.2677226. URL: https://dl.acm.org/doi/10.1145/2676723.2677226 (visitado 10-03-2025).

[14] Benedikt Wisniewski, Klaus Zierer y John Hattie. «The Power of Feedback Revisited: A Meta-Analysis of Educational Feedback Research». En: *Frontiers in Psychology* 10 (ene. de 2020). DOI: 10.3389/fpsyg.2019.03087. URL: https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2019.03087/full (visitado 21-03-2025).

[15] Marcus Messer et al. «Automated Grading and Feedback Tools for Programming Education: A Systematic Review». En: *ACM Transactions on Computing Education* 24 (dic. de 2023), págs. 1-43. DOI: 10.1145/3636515. URL: https://dl.acm.org/doi/10.1145/3636515 (visitado 21-03-2025).

[16] Huanyi Chen y Paul Ward. *Predicting student performance using data from an Auto-grading system*. 2019. URL: https://arxiv.org/pdf/2102.01270 (visitado 21-03-2025).

[17] Scott Chacon y Ben Straub. *Pro Git*. Apress, 2014. URL: https://git-scm.com/book/en/v2.

[18] GitHub Staff. *Octoverse: AI leads Python to top language as the number of global developers surges*. The GitHub Blog, oct. de 2024. URL: https://github.blog/news-insights/octoverse/octoverse-2024/ (visitado 07-04-2025).

[19] Courtney Hsing. *Improve student success and increase teacher time with autograding*. The GitHub Blog, mar. de 2020. URL: https://github.blog/developer-skills/github-education/improve-student-success-and-increase-teacher-time-with-autograding/ (visitado 11-03-2025).

[20] Arelia Jones. *Introducing Autograding for GitHub Classroom and the GitHub Teacher Toolbox*. The GitHub Blog, mar. de 2020. URL: https://github.blog/developer-skills/github/github-teacher-toolbox-and-classroom-with-autograding/ (visitado 14-03-2025).

[21] Paul M Duvall, Steve Matyas y Andrew Glover. *Continuous Integration*. Pearson Education, jun. de 2007.

[22] Jez Humble y David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2011.

[23] Jessica Fender. *How DevOps is Transforming Higher Education? - Kovair Blog*. Kovair Blog, abr. de 2022. URL: https://www.kovair.com/blog/how-devops-is-transforming-higher-education/ (visitado 20-03-2025).

[24] Maria Augusta Nelson y Lesandro Ponciano. «Experiences and insights from using Github Classroom to support Project-Based Courses». En: *arXiv (Cornell University)* (mayo de 2021), págs. 31-35. DOI: 10.1109/seeng53126.2021.00013. URL: https://www.researchgate.net/publication/353118069_

Experiences_and_insights_from_using_Github_Classroom_to_support_Project-Based_Courses (visitado 25-03-2025).

[25] Laura Schauer, Robert Stewart y Manuel Maarek. «Integrating Canvas and GitLab to Enrich Learning Processes». En: (abr. de 2024), págs. 180-190. DOI: 10.1145/3639474.3640056. URL: https://dl.acm.org/doi/10.1145/3639474.3640056 (visitado 24-03-2025).

[26] Trifan Ionel Constantin. «Deliverit 1.0: mejoras en los entornos de ejecución de las prácticas | Archivo Digital UPM». En: *Oa.upm.es* (jun. de 2021). DOI: https://oa.upm.es/68533/. URL: https://oa.upm.es/68533/ (visitado 07-04-2025).

[27] Carambia Corbella Nicola. «DeliverIt: extensión para la corrección de prácticas | Archivo Digital UPM». En: *Oa.upm.es* (mayo de 2024). DOI: https://oa.upm.es/82497/. URL: https://oa.upm.es/82497/ (visitado 07-04-2025).

[28] Arjun Singh et al. «Gradescope». En: (abr. de 2017), págs. 81-88. DOI: 10.1145/3051457.3051466. URL: https://dl.acm.org/doi/10.1145/3051457.3051466 (visitado 20-03-2025).

[29] Jesús Manuel Gallego-Romero et al. «Analyzing learners' engagement and behavior in MOOCs on programming with the Codeboard IDE». En: *Educational Technology Research and Development* 68 (abr. de 2020), págs. 2505-2528. DOI: 10.1007/s11423-020-09773-6. URL: https://link.springer.com/article/10.1007/s11423-020-09773-6 (visitado 29-04-2025).

[30] Jake Zimmerman. *Autolab: Autograding for All*. Github.io, 2015. URL: https://autolab.github.io/2015/03/autolab-autograding-for-all/ (visitado 22-04-2025).

[31] David Croft y Matthew England. «Computing with CodeRunner at Coventry University». En: *arXiv (Cornell University)* (ene. de 2020), págs. 1-4. DOI: 10.1145/3372356.3372357. URL: https://dl.acm.org/doi/10.1145/3372356.3372357 (visitado 23-03-2025).

[32] Philipp Straubinger y Gordon Fraser. «Gamifying a Software Testing Course with Continuous Integration». En: *arXiv (Cornell University)* (abr. de 2024), págs. 34-45. DOI: 10.1145/3639474.3640054. URL: https://dl.acm.org/doi/10.1145/3639474.3640054 (visitado 22-03-2025).

[33] Carrie A. Hansel et al. «Gradescope in Large Lecture Classrooms: A Case Study at Indiana University». En: *Journal of Teaching and Learning with Technology* 13 (dic. de 2024). DOI: 10.14434/jotlt.v13i1.38519. URL: https://scholarworks.iu.edu/journals/index.php/jotlt/article/view/38519 (visitado 27-03-2025).

[34] Christoph Matthies, Arian Treffer y Matthias Uflacker. «Prof. CI: Employing continuous integration services and Github workflows to teach test-driven development». En: *2021 IEEE Frontiers in Education Conference (FIE)* (oct. de 2017). DOI: 10.1109/fie.2017.8190589. URL: https://dl.acm.org/doi/10.1109/FIE.2017.8190589?utm_source=chatgpt.com (visitado 13-04-2025).

[35] Chris Brown y Chris Parnin. «Understanding the impact of GitHub suggested changes on recommendations between developers». En: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (nov. de 2020), págs. 1065-1076. DOI: 10.1145/3368089.3409722. URL: https://dl.acm.org/doi/10.1145/3368089.3409722 (visitado 15-05-2025).

Listado de acrónimos

API REST Interfaz de Programación de Aplicaciones (*Application Programming Interface*) basada en el estilo arquitectónico *Representational State Transfer*; expone recursos mediante peticiones HTTP con verbos estándar.. 35

Bash «Bourne Again SHell»; intérprete de órdenes y lenguaje de *scripting* omnipresente en sistemas Unix/Linux.. 27

CD Entrega Continua (*Continuous Delivery*); automatiza el paso de código verificado a entornos de producción o pre-producción, cerrando el ciclo tras la CI.. 6, 55, 57

CI Integración Continua (*Continuous Integration*); práctica que ejecuta pruebas y análisis automáticamente en cada *commit* para detectar errores de forma temprana, eje central de los flujos descritos en la memoria.. 3, 51, 54, 55, 57

CodeQL «Code Query Language»; motor de análisis estático de GitHub que describe patrones de fallo en una base de datos semántica del código.. 7, 57

CSV *Comma-Separated Values*; formato de texto tabular donde cada línea representa un registro y los campos se separan por comas. Útil para intercambiar grandes volúmenes de datos estructurados.. 8

DAG «Directed Acyclic Graph» (grafo dirigido y acíclico); estructura de nodos y aristas sin ciclos, usada por Git para modelar el historial de *commits*.. 7

HTML «HyperText Markup Language»; lenguaje de marcado que estructura documentos web mediante etiquetas como `<h1>` o `<p>`.. 11

JaCoCo Java Code Coverage; herramienta que mide qué líneas de un programa Java se han ejecutado durante las pruebas.. 30

JDK Java Development Kit; kit oficial que incluye el compilador `javac`, la máquina virtual y utilidades para desarrollar en Java.. 24

JSON JavaScript Object Notation; formato ligero de intercambio de datos basado en pares clave–valor.. 15

LMS Learning Management System; plataforma que gestiona actividades, materiales y calificaciones de un curso (p. ej. Moodle).. 8

OAuth «Open Authorization»; protocolo que permite delegar permisos entre servicios sin compartir la contraseña del usuario, mediante el intercambio de *tokens*.. 20

ODS Objetivos de Desarrollo Sostenible; agenda de la ONU compuesta por 17 metas globales para erradicar la pobreza, proteger el planeta y promover la prosperidad antes de 2030.. 39

OOM Out Of Memory; condición en la que un proceso supera los límites de memoria asignados por el sistema operativo.. 33

SHA-1 «Secure Hash Algorithm 1»; función de huella criptográfica de 160 bits usada para identificar cada commit en Git.. 7

SHA-256 «Secure Hash Algorithm 256»; versión de 256 bits que GitHub está adoptando para sustituir los identificadores SHA-1.. 7

TTL Time To Live; parámetro que determina la vida útil máxima de un recurso (p. ej. un artefacto de CI, una entrada de caché o un contenedor efímero) antes de que el sistema lo purgue automáticamente.. 38

YAML «YAML Ain't Markup Language»; formato legible de serialización de datos (clave:valor) empleado para definir workflows.. 10

Glosario de términos

fail-fast Estrategia que aborta la ejecución del pipeline en cuanto surge el primer error, ahorrando tiempo y recursos. Evita esperar a pasos posteriores si algún paso ha fallado y detiene otras ejecuciones concurrentes, como las de una matriz de entornos.. 38

`timeout-minutes` Parámetro de GitHub Actions que cancela un job si excede el tiempo máximo definido.. 34

artefacto Archivo generado por un flujo de integración continua (p. ej. un `.jar`, un PDF o un informe de cobertura) que el sistema almacena como *artifact* para que el alumnado lo descargue y revise.. 7

autograding Corrección y calificación automática de entregas de código mediante la ejecución de pruebas en un entorno controlado.. 3, 8

billing manager Rol de GitHub que puede gestionar la facturación de una organización (planes, métodos de pago, etc.) sin acceder a los repositorios de la organización.. 20

branch Rama de desarrollo en Git; una línea independiente de *commits*. Comúnmente, se trabaja con una rama `main` o `master` princial y, opcionalmente, en ramas de adicionales, como `dev` o `feature/feature2` sobre las cuales se desarrolla código hasta su madurez, cuando se introducen los commits en la principial mediante las estrategias de intregación de ramas.. 7

branch protection Conjunto de reglas que impiden modificar una rama (p. ej. `main`) sin cumplir requisitos como pasar los tests o contar con revisiones. Se usa para garantizar calidad y fomentar buenas prácticas.. 7

build Dentro de CI/CD, Proceso automatizado que compila el código y genera artefactos; suele incluir pruebas y análisis antes de producir un artefacto.. 8

cache Almacén intermedio de datos de acceso rápido; en CI se usa para guardar dependencias y acelerar construcciones sucesivas.. 37

cgroup2 Segunda versión de *control groups* de Linux que aísla y limita recursos (CPU, memoria, procesos) en los contenedores.. 34

checkout Comando Git que conmuta la rama activa o restaura archivos. En GitHub Actions el paso `actions/checkout` clona el repositorio para los jobs.. 7

checks Conjunto de resultados (tests, linter, cobertura) que GitHub muestra en cada *pull request*.. 6

CircleCI Plataforma externa de CI/CD basada en contenedores comparada en la memoria como alternativa educativa a GitHub Actions.. 13

CodeRunner Plugin para Moodle que ejecuta código enviado por el alumnado y comprueba la salida con tests, función análoga al *autograding* descrito.. 12

commit Instantánea inmutable del estado del proyecto en Git, identificada por un hash; base de las ramas y del sistema de calificación automática.. 3

concurrencia Ejecución simultánea de varios procesos; en Actions permite lanzar jobs en paralelo o limitar a uno mediante grupos de concurrencia.. 37

container Registry Servicio para almacenar y versionar imágenes de contenedor, p. ej. GitHub Container Registry, desde donde los runners descargan entornos.. 7

contenedor Entorno ligero, portable y aislado (habitualmente Docker) que empaqueta aplicación y dependencias, garantizando pruebas reproducibles.. 6

cuadro de mando / Dashboard Vista gráfica que agrupa indicadores clave (tasas de éxito, entregas atrasadas, etc.) y orienta la toma de decisiones.. 8

Dependabot Servicio integrado en GitHub que rastrea dependencias y abre *pull requests* automáticas cuando hay actualizaciones o vulnerabilidades, garantizando entornos seguros en los repositorios.. 7

deploy Acción de poner en marcha una versión del software en un entorno (pruebas, producción, aula virtual). Puede encadenarse tras la CI para publicar el contenido si ha superado la integración.. 9

DevOps Conjunto de prácticas que integran el desarrollo de software (*Development*) y las operaciones de TI (*Operations*), con el objetivo de automatizar y mejorar el proceso de construcción, prueba e implementación de aplicaciones. Favorece la colaboración entre equipos, la entrega continua y la infraestructura como código.. 2

digest diario Correo o mensaje resumido que, una vez al día, lista nuevas entregas, fallos de tests o métricas. Se genera mediante un *cron job*.. 36

Docker Plataforma de contenedores que empaqueta software con sus dependencias.. 8

documentación Conjunto organizado de guías, ejemplos y referencias que explica cómo usar y mantener el proyecto. Facilita la transferibilidad del código.. 17

evento Suceso que inicia automáticamente un flujo de Integración Continua en GitHub Actions (p. ej. `push`, `schedule` o `workflow_dispatch`). Permite que las pruebas se ejecuten justo cuando ocurre la acción relevante.. 9

feedback inmediato Respuesta automática generada segundos después de realizar una entrega; muestra la nota provisional y fomenta el aprendizaje iterativo.. 2, 6

fork En el contexto Unix, llamada al sistema que permite a un proceso crear una copia de sí mismo. En el contexo de Git, copia de un repositorio bajo otra cuenta u organización.. 34

gamificación Aplicación de mecánicas de juego (puntos, niveles, insignias) al proceso de aprendizaje para aumentar la motivación.. 2

GitHub Plataforma de alojamiento de repositorios Git con servicios de colaboración, revisión de código y automatización; es la columna vertebral del flujo docente descrito en la memoria.. 1

GitHub Actions Infraestructura de Integración/Entrega Continua nativa de GitHub que ejecuta *workflows* declarados en YAML. Se usa para autograding, generación de artefactos y publicación de resultados.. 1

GitHub Classroom Extensión de GitHub que crea repositorios individuales o por equipo, reparte plantillas, lanza CI y centraliza la calificación del alumnado.. 3

GitHub Enterprise Server Versión autohospedada de GitHub que permite mantener los datos dentro de la infraestructura de la universidad cuando la normativa lo exige.. 13

GitHub Pages Sistema de hosting estático que despliega automáticamente el sitio web generado (documentación o demo) desde la rama `gh-pages` o mediante una GitHub Action.. 7

GitLab Plataforma de repositorios Git con funcionalidades similares a GitHub y su propia «GitLab CI/CD». Usada en la asignatura Programming Project para la gestión de proyectos de prácticas.. 11

halo effect Sesgo cognitivo en el que una impresión previa (p. ej., rendimiento histórico o percepción personal) influye en la evaluación posterior, pudiendo inflar o penalizar la nota de trabajos subsecuentes y comprometer la objetividad.. 39

inter-rater reliability Medida de consistencia estadística entre distintos correctores que califican una misma tarea.. 39

issue Ticket de GitHub que registra una tarea, duda o bug. Puede enlazarse a *pull requests*, asignarse y relacionarse con tableros de proyecto para un seguimiento claro.. 7

Jenkins Servidor CI/CD de código abierto basado en plugins; se compara con GitHub Actions como opción para la evaluación automática.. 3

JUnit Framework de pruebas unitarias para Java.. 12

kanban Método de gestión visual del trabajo mediante tarjetas en columnas (pendiente, en curso, hecho); GitHub Projects lo implementa.. 7

learning analytics Análisis de datos educativos (participación, entregas, notas) para mejorar la docencia; los *logs* de CI pueden alimentar estas métricas.. 6

linter Programa que revisa el estilo y posibles errores estáticos del código.. 8

logs Registros cronológicos que detallan la ejecución de pruebas y builds; permiten diagnosticar fallos.. 6

merge Operación Git que combina el historial de dos ramas; suele requerir que todos los *checks* estén verdes.. 7

organización Entidad de GitHub que agrupa repositorios, equipos y permisos bajo un espacio común, ideal para separar la docencia por asignaturas.. 7

pipeline Secuencia ordenada de pasos (build, test, deploy) que GitHub Actions ejecuta de forma automática en cada cambio del repositorio.. 1

plantilla En este contexto, repositorio base con código esqueleto y configuración de CI que el profesorado clona para cada alumno vía GitHub Classroom.. 16

rebase Reescritura de una rama sobre otra, aplicando los commits en un nuevo punto, lo que mantiene un historial lineal.. 7

refactorización Reestructuración interna del código sin alterar su funcionalidad, mejorando legibilidad y mantenibilidad.. 24

release Versión estable etiquetada del software, acompañada de notas y artefactos descargables.. 24

repositorio Estructura que almacena el historial Git de un proyecto, incluyendo ramas, tags y issues.. 11

repositorio privado Repositorio de código cuyo contenido no es accesible públicamente; solo los usuarios con permisos explícitos pueden visualizarlo o colaborar en él. 8

reset Comando Git que mueve la referencia de rama a otro commit, opcionalmente modificando el área de trabajo.. 7

revert Comando Git que crea un nuevo commit que deshace los cambios introducidos por otro commit.. 7

runner Máquina (virtual o física) que ejecuta los jobs de GitHub Actions para el autograding.. 9

rúbrica Tabla de criterios que detalla cómo se asignan los puntos de la calificación, ofreciendo transparencia al estudiante.. 12

script Archivo de instrucciones (Bash, `sh`) que automatiza tareas. En los pasos de CI se encarga de tarieas como preparar dependencias o lanzar pruebas. Un script puramente hablando puede estar escrito en cualquier lenguaje de programación y tener cualquier función.. 22

secret Valor confidencial (token, contraseña) que GitHub cifra y suministra a los workflows sin exponerlo en los logs del autograding.. 10

self-hosted runner Máquina gestionada por la universidad que ejecuta los jobs de CI cuando los runners de GitHub no cubren requisitos especiales (GPU, software licenciado).. 11

squash merge Fusión que aplasta todos los commits de una rama en uno solo, dejando un historial lineal y limpio.. 7

static code analysis Examen automático del código sin ejecutarlo (linters, CodeQL) para detectar errores de estilo, seguridad o rendimiento.. 8

step Acción individual dentro de un job de GitHub Actions; cada step ejecuta un comando o acción.. 9

tag Etiqueta inmutable que señala un commit concreto, habitualmente para marcar una *release*.. 7

teacher Rol docente con permisos de administración en GitHub Classroom y responsabilidad de evaluar.. 19

test de integración Prueba que verifica la comunicación entre varios componentes (base de datos, API) de la aplicación.. 8

test privado Caso de prueba oculto al estudiante que se ejecuta tras la entrega para evitar soluciones sobre-ajustadas.. 17

test unitario Prueba que comprueba el comportamiento de la unidad más pequeña de código (método o función).. 8

timeout Límite máximo de tiempo de ejecución asignado a un proceso o *job*. Si la tarea no finaliza antes de dicho límite, el sistema la cancela para evitar bloqueos y liberar recursos.. 37

toolchain Conjunto de herramientas de desarrollo—compilador, gestor de dependencias, linter, framework de pruebas, etc.—que, encadenadas, permiten construir, probar y desplegar software de forma reproducible.. 38

Travis CI Servicio externo de CI/CD basado en YAML, precedente de GitHub Actions en muchos proyectos open source.. 13

tutoría Sesión personalizada en la que el docente orienta al estudiante sobre dudas técnicas o de planificación de la práctica.. 36

variable Dato cuyo valor puede cambiar; en GitHub Actions las *variables de entorno* parametrizan rutas, versiones o credenciales.. 17

workflow Archivo YAML que define los jobs y pasos que GitHub Actions debe ejecutar ante un evento.. 4

Apéndice A

Contenidos expandidos

El presente anexo recopila material técnico complementario para apoyar y profundizar en el contenido expuesto en la memoria principal de este Trabajo de Fin de Grado.

A.1. Organización del anexo

Este anexo está estructurado para facilitar la consulta y reutilización de materiales clave relacionados con el sistema de corrección automatizada propuesto.

La organización del contenido es la siguiente:

- **Diagrama de secuencia detallado.** Ilustra las interacciones entre los componentes del sistema, resaltando las fases críticas del flujo operativo.
- **Workflows completos (YAML).** Se incluyen todos los workflows desarrollados para GitHub Actions, preparados para su adaptación directa en nuevas prácticas.
- **Evidencias de ejecución.** Documentación provisional de logs y resultados obtenidos durante pruebas simuladas, útiles para verificar el correcto funcionamiento de los workflows descritos.
- **Repositorio del proyecto.** Describe cómo se estructura el repositorio del proyecto, facilitando la localización de la memoria, ejemplos prácticos y workflows correspondientes.

A.2. Diagrama de interacción de componentes

A continuación se presenta un diagrama de secuencia más detallado, que tiene como objetivo ilustrar con mayor precisión la interacción entre todos los componentes que conforman el sistema desarrollado. En el diagrama se destacan, mediante recuadros, cuatro ciclos fundamentales que constituyen el flujo operativo completo del sistema, descritos a continuación en orden descendente:

1. **Creación del repositorio individual por parte del alumno.** Este ciclo se activa cuando el estudiante inicia la resolución de una práctica previamente publicada

por el docente a través de GitHub Classroom, generándose automáticamente un repositorio individual a partir de la plantilla preparada.

2. **Ejecución automatizada de pruebas o tests.** Este ciclo comienza con la subida del código al repositorio por parte del alumno, lo que dispara automáticamente la ejecución de los tests definidos mediante GitHub Actions. La ejecución es concurrente tanto a nivel individual (múltiples trabajos simultáneos dentro de la misma entrega) como a nivel colectivo (entregas simultáneas de diferentes estudiantes). El sistema gestiona adecuadamente esta concurrencia según la estrategia configurada.
3. **Obtención y notificación de resultados de la corrección.** En este punto, el sistema evalúa dos posibles escenarios tras ejecutar los tests: que la entrega supere todas las pruebas o que falle alguna o varias de ellas. Dependiendo del resultado obtenido, la notificación y el feedback facilitado al alumno serán distintos, indicando claramente los errores o la confirmación del éxito y la correspondiente puntuación.
4. **Revisión manual final por parte del profesor.** Una vez finalizado el periodo habilitado para la entrega de prácticas, el profesor dispone de acceso completo al código entregado, así como al historial detallado de commits realizados por cada alumno. Garantizada la corrección funcional automática por los pasos anteriores, este ciclo permite al docente ofrecer retroalimentación adicional sobre aspectos cualitativos como calidad del código, uso adecuado de buenas prácticas, documentación, o cualquier otro criterio relevante desde el punto de vista académico.

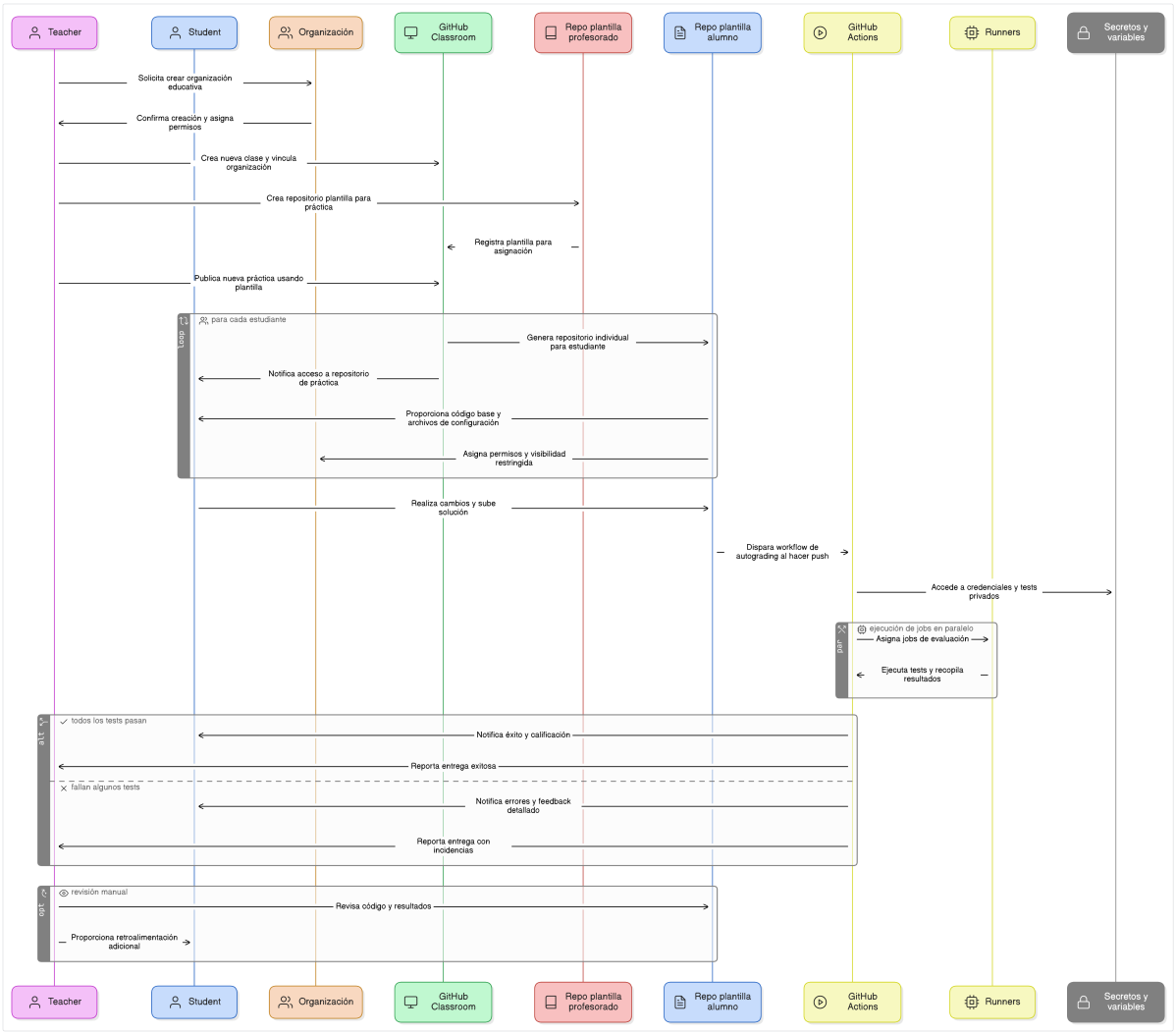


Figura A.1: Diagrama de secuencia en detalle

A.3. Workflows de GitHub Actions (YAML completos)

En esta sección se incluyen los workflows completos, comentados, para cada uno de los casos contemplados en la memoria.

A.3.1. Workflow para la correción mediante Entrada/Salida

Workflow que compila una clase Java y ejecuta una batería de pruebas definidas mediante pares de entrada/salida, verificando automáticamente la salida esperada. Sigue lo explicado en la Sección 3.3.1, pero ofreciendo todo el flujo completo.

```
name: IO-basic
on: [push, pull_request]

jobs:
  io-basic:
    runs-on: ubuntu-latest
    timeout-minutes: 3
```

```
    concurrency:
      group: ${{ github.ref }}
      cancel-in-progress: true
    permissions:
      contents: read          # minimo necesario

    steps:
    - name: Checkout codigo del alumno
      uses: actions/checkout@v4

    - name: Instalar Java 21
      uses: actions/setup-java@v4
      with:
        distribution: temurin
        java-version: 21

    - name: Compilar con javac
      run: |
        mkdir -p out
        javac -encoding UTF-8 -d out $(git ls-files '*.java')

    - name: Auto-grading I/O (GitHub Classroom)
      uses: education/autograding@v1

    - name: Resumen Markdown (bash)
      run: |
        echo '### Caso `input01`' >> $GITHUB_STEP_SUMMARY
        if [ $? -eq 0 ]; then
          echo 'Todo OK' >> $GITHUB_STEP_SUMMARY
        else
          echo 'La salida no coincide' >> $GITHUB_STEP_SUMMARY
        fi
```

Listing A.1: Flujo completo para la corrección mediante Entrada/Salida

Adicionalmente, se presenta este mismo flujo, pero incluyendo las alternativas de corrección y de generación de resúmenes vistas en la Sección 3.3.1.

```
name: IO-basic
on: [push, pull_request]

jobs:
  io-basic:
    runs-on: ubuntu-latest
    timeout-minutes: 3
    concurrency:
      group: ${{ github.ref }}
      cancel-in-progress: true
    permissions:
      contents: read          # minimo necesario

    steps:
    - name: Checkout codigo del alumno
      uses: actions/checkout@v4

    - name: Instalar Java 21
      uses: actions/setup-java@v4
      with:
        distribution: temurin
        java-version: 21

    - name: Compilar con javac
      run: |
        mkdir -p out
        javac -encoding UTF-8 -d out $(git ls-files '*.java')

    - name: Auto-grading I/O (GitHub Classroom)
      uses: education/autograding@v1
```

```
31
32      # --- opcion clasica con diff ---
33      # - name: Ejecutar y comparar con diff
34      #   run: |
35      #     java -cp out Main < tests/public/input01.txt > alumno.out
36      #     diff -q alumno.out tests/public/output01.txt
37
38      # --- opcion avanzada con Action propia ---
39      # - name: Corrector I/O (accion propia)
40      #   uses: org/corrector-io@v1
41      #   with:
42      #     input: tests/public/input01.txt
43      #     expected: tests/public/output01.txt
44      #     main-class: Main
45
46      - name: Resumen Markdown (bash)
47        run: |
48          echo '### Caso `input01`' >> $GITHUB_STEP_SUMMARY
49          if [ $? -eq 0 ]; then
50            echo 'Todo OK' >> $GITHUB_STEP_SUMMARY
51          else
52            echo 'La salida no coincide' >> $GITHUB_STEP_SUMMARY
53          fi
54
55      # --- opcion alternativa con JavaScript mediante Action oficial ---
56      # - name: Resumen (github-script)
57      #   uses: actions/github-script@v7
58      #   with:
59      #     script: |
60      #       const ok = process.exitCode === 0
61      #       await core.summary
62      #         .addHeading('Caso 'input01')
63      #         .addRaw(ok ? 'Todo OK' : 'La salida no coincide')
64      #         .write()
65
66      # --- opcion avanzada con Action propia ---
67      # - name: Resumen desde script
68      #   run: scripts/post-summary.sh ${{ steps.evaluacion.outcome }}
```

Listing A.2: Flujo completo para la corrección. Contiene varias alternativas

A.3.2. Workflow para la correción mediante tests privados

Workflow avanzado que utiliza tests alojados en repositorios privados accesibles únicamente para el profesorado mediante tokens y secretos, ejecutando estas pruebas contra el código entregado por el alumno y proporcionando feedback claro y útil. Los secretos deben ser configurados debidamente antes de difundir la práctica al alumnado. Sigue lo explicado en la Sección 3.3.2, pero ofreciendo todo el flujo completo.

```
1  name: JUnit-private
2  on: [push]
3
4  jobs:
5    junit-private:
6      runs-on: ubuntu-latest
7      timeout-minutes: 5
8      permissions:
9        contents: read          # minimo necesario
10
11     steps:
12     - name: Checkout codigo del alumno
13       uses: actions/checkout@v4
14
15     - name: Checkout tests privados
16       uses: actions/checkout@v4
```

```
17        with:
18          repository: org/tests-privados-p2
19          ssh-key: ${{ secrets.TEST_REPO_KEY }}
20          path: tests_privados
21          persist-credentials: false        # borra la clave tras el clone
22
23      - name: Instalar Java 21 + cache Maven
24        uses: actions/setup-java@v4
25        with:
26          distribution: temurin
27          java-version: 21
28          cache: maven
29
30      - name: Compilar con Maven (sin tests)
31        run: mvn -B package -DskipTests
32
33      - name: Copiar pruebas privadas a src/test/java
34        run: cp -R tests_privados/*.java src/test/java/ # se copian los tests al directorio
              esperado por Maven
35
36      - name: Ejecutar pruebas JUnit
37        run: mvn -B test
38
39      - name: Publicar informe Surefire
40        uses: scacap/action-surefire-report@v1  # usa accion externa
41        with:
42          github_token: ${{ secrets.GITHUB_TOKEN }}
```

Listing A.3: Flujo completo para la corrección usando tests privados

A.3.3. Workflow para la correción de un proyecto Maven

Workflow orientado a la compilación y ejecución de proyectos basados en Maven, incluyendo la ejecución automatizada de tests unitarios (JUnit) y generación de documentación (Javadoc). Con el fin de ser ilustrativo, se simplifica a la compilación y los pasos vistos en la Sección 3.3.3, pero ofreciendo todo el flujo completo.

```
1  name: Maven-full
2  on:
3    push:
4      paths-ignore:
5        - "**/*.md"
6    pull_request:
7
8  jobs:
9    build-test-doc:
10     runs-on: ubuntu-latest
11     timeout-minutes: 6
12     permissions:
13       contents: read          # suficiente para checkout y artifacts
14
15     steps:
16     - name: Checkout codigo del alumno
17       uses: actions/checkout@v4
18
19     - name: Instalar Java 21 + cache Maven
20       uses: actions/setup-java@v4
21       with:
22         distribution: temurin
23         java-version: 21
24         cache: maven
25
26     - name: Compilar, ejecutar tests y generar cobertura
27       run: mvn -B verify jacoco:report
28
29     - name: Generar Javadoc
```

```
30        run: mvn -B javadoc:javadoc
31
32      - name: Subir artefactos (JAR, Jacoco, Javadoc)
33        uses: actions/upload-artifact@v4
34        with:
35          name: build-output
36          path: |
37            target/*.jar
38            target/site/jacoco
39            target/site/apidocs
40          retention-days: 30
```

Listing A.4: Flujo completo para la corrección de un proyecto Maven

A.4. Evidencias de ejecución

Se incluyen aquí evidencias documentales sobre la ejecución de los workflows desarrollados. Las siguientes subsecciones presentan, para cada uno de los casos de uso implementados (entrada/salida estándar, tests privados y proyecto Maven), capturas y resultados detallados que ilustran claramente el comportamiento y la eficacia del sistema de corrección automática propuesto.

A.4.1. Ejecución del flujo Entrada/Salida

A continuación se presentan los resultados obtenidos tras ejecutar el workflow diseñado para verificar prácticas básicas mediante comparaciones directas de entrada y salida estándar.

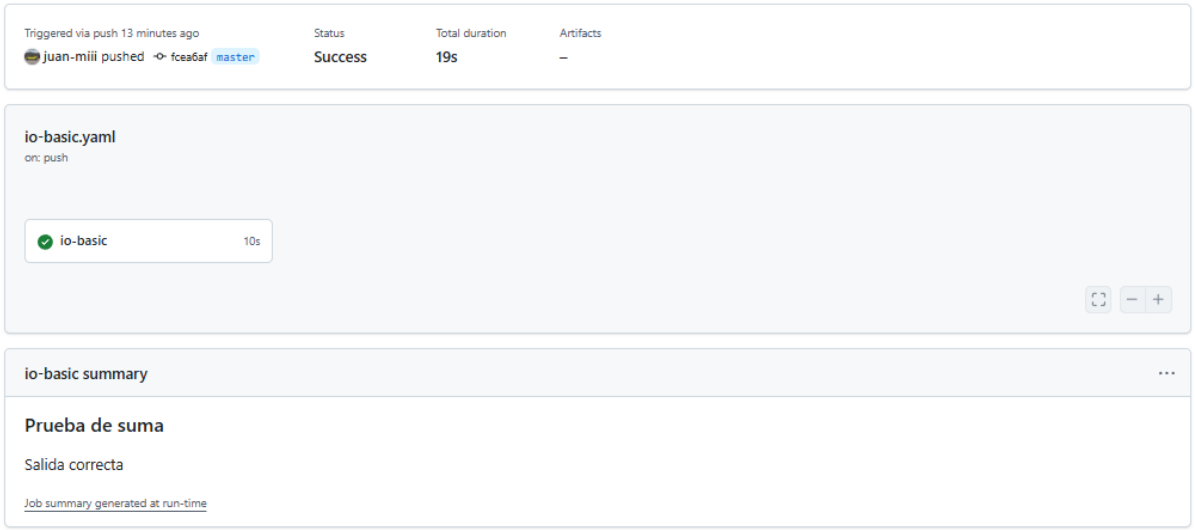


Figura A.2: Resultado de la ejecución del flujo Entrada/Salida

En este flujo, el objetivo es proporcionar al estudiante un feedback claro y directo sobre si ha superado las pruebas planteadas. La ejecución muestra el resultado inmediato de la comparación entre la salida esperada y la obtenida, facilitando así la detección rápida y sencilla de errores.

A.4.2. Ejecución del flujo mediante tests privados

Se muestran aquí los resultados derivados de la ejecución del workflow que realiza pruebas más complejas utilizando tests privados proporcionados por el profesorado.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.example.UtilityPrivateTest
[INFO] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.135 s -- in com.
    example.UtilityPrivateTest
[INFO] Running com.example.UtilityStudentTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.020 s -- in com.
    example.UtilityStudentTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 18, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.899 s
[INFO] Finished at: 2025-06-03T22:16:51Z
[INFO] ------------------------------------------------------------------------
```

Listing A.5: Fragmento del log donde se muestra el resumen de la ejecución de los tests

En esta imagen puede apreciarse cómo los tests privados se ejecutan de manera segura utilizando repositorios con acceso restringido. Se muestra claramente el número total de tests ejecutados, diferenciándose entre las pruebas definidas por el alumno (5 tests) y aquellas adicionales definidas por el profesorado (13 tests), sumando un total de 18 pruebas ejecutadas. El log detalla el resultado general de las pruebas sin revelar información específica sobre el contenido exacto de los tests privados, garantizando así la integridad académica.

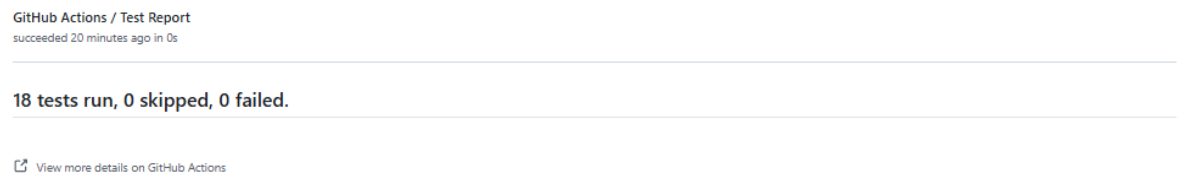


Figura A.3: Reporte Surefire producido en el flujo

El reporte generado por Surefire complementa el log principal, ofreciendo una visión consolidada de las pruebas superadas y aquellas fallidas, permitiendo al alumno identificar rápidamente puntos específicos que requieren revisión adicional.

A.4.3. Ejecución del flujo para proyectos Maven

Finalmente, se incluyen evidencias relacionadas con el workflow específico para la evaluación automática de proyectos Java gestionados mediante Maven.

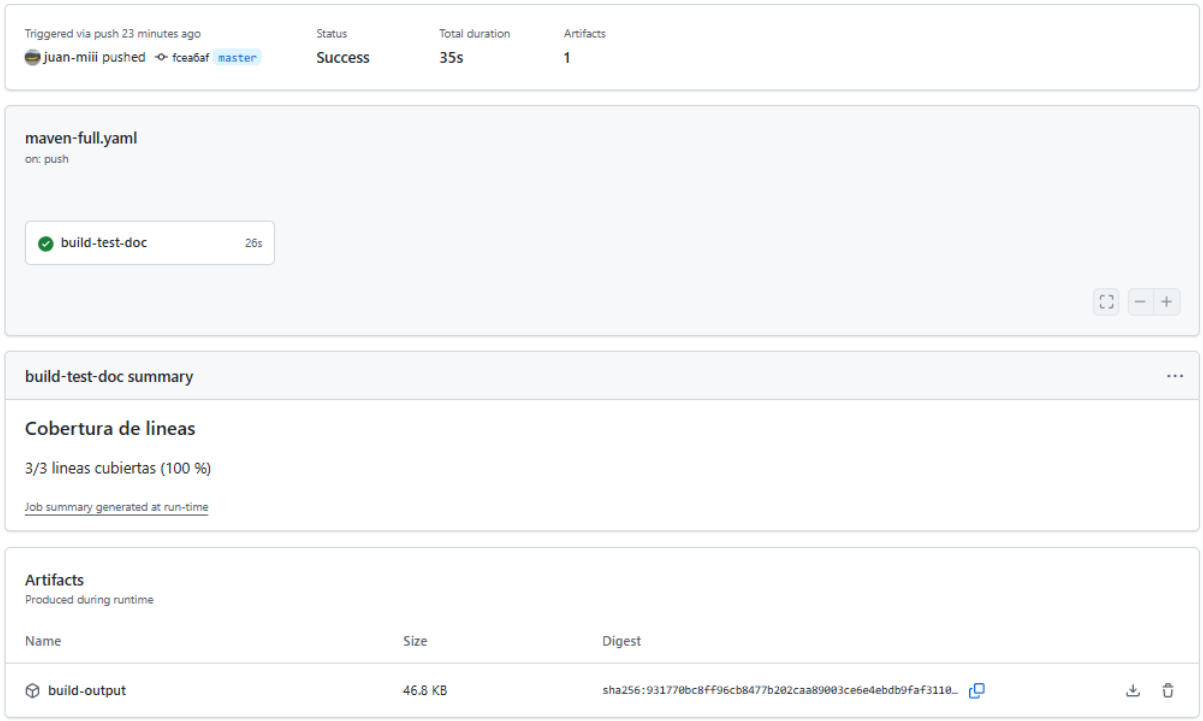


Figura A.4: Resultado de la ejecución del flujo Maven

La captura evidencia que el workflow compila correctamente el proyecto Maven, ejecuta satisfactoriamente los tests unitarios definidos y genera automáticamente la documentación mediante Javadoc. Además, los reportes generados por Jacoco y Javadoc son almacenados como artefactos descargables, permitiendo así una validación exhaustiva inicial que complementa y facilita la posterior evaluación manual realizada por el docente.

A.5. Repositorio del proyecto

El repositorio del proyecto está diseñado para facilitar el acceso rápido y claro al código y ejemplos desarrollados, permitiendo su reutilización y modificación posterior por parte del profesorado y alumnado.

Debido a que la configuración real de GitHub Classroom no es necesaria para demostrar las capacidades técnicas de corrección automatizada, se han adaptado los workflows para ejecutarse automáticamente tras cada commit realizado sobre los ejemplos proporcionados. De esta forma, se garantiza la ejecución repetible y documentable de las pruebas descritas en la memoria principal.

El enlace para acceder a estos contenidos es: `https://github.com/juan-miii/tfg`

Todo el material de referencia está documentado in situ dentro del propio repositorio, de modo que cualquier lector puede navegar por el árbol y entender cada pieza en contexto; no obstante, a continuación se ofrece un panorama global.

En la Sección 3.3 de la memoria se describen tres escenarios de evaluación (Entra-

da/Salida básica, Tests JUnit privados y proyecto Maven completo). El repositorio reproduce esos mismos tres casos con una implementación mínima de alumno:

ejemplos/io/ Contiene un "Hola mundo de la suma" que lee dos enteros y devuelve su resultado; sirve para demostrar la corrección por diff.

ejemplos/externo/ Aloja la clase Utility con cinco funciones y un único test visible por función; el resto de los casos límite (nulos, vacíos, división por cero...) se cubren mediante tests externos inyectados en tiempo de CI.

ejemplos/maven/ Proporciona una aplicación Maven sencilla (Calculator, Greeter) con Jacoco y Javadoc configurados para ilustrar cobertura y documentación automáticas.

Cada ejemplo tiene su propio workflow bajo **.github/workflows/**, configurado con filtros paths: para que solo se dispare si se modifica su carpeta; así se evita gastar minutos de runner en los demás ejemplos. Los pasos del workflow están muy comentados y emplean timeouts y caché de Maven (cuando procede) para reflejar buenas prácticas de robustez.

El caso más sofisticado es el de JUnit con tests externos. Aquí se aprecia que:

1. Hay dos variables que cubren la privacidad de las pruebas: (`TEST_REPO_NAME`) se ha obtenido del repositorio como `usuario/repositorio`, permitiendo obtener las pruebas de un repositorio desconocido para el alumno. (`TEST_REPO_KEY`) es la clave privada (`ssh-ed25519`) generada en el repositorio para poder hacer esto. En el repositorio que contiene las pruebas se ha incluido una *Deploy Key* con la clave pública.

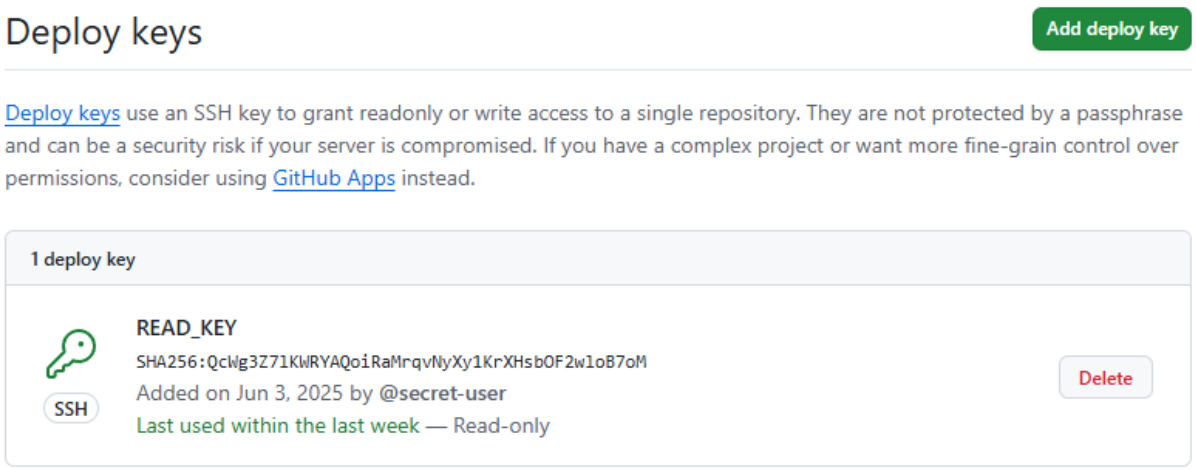


Figura A.5: Deploy Key en el repositorio de pruebas

En el repositorio destino, `juan-miii/tfg`, se almacenan ambas como secrets para Actions, por lo que no aparecen en los logs ni en el YAML.

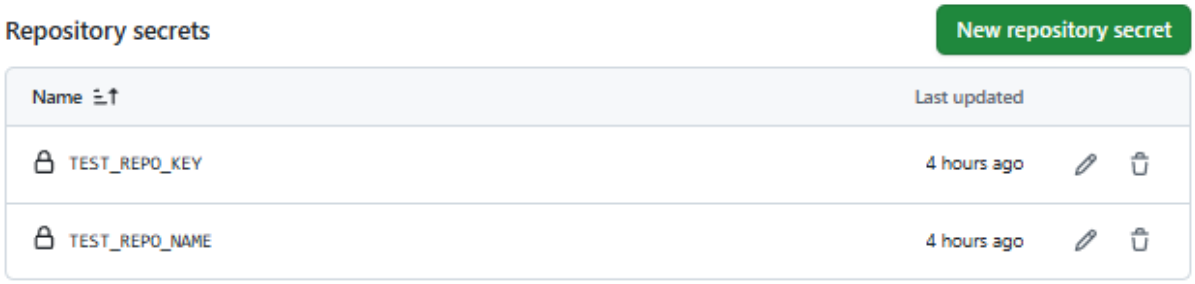


Figura A.6: Secretos del repositorio

2. Los tests se importan en tiempo de ejecución con una deploy key de sólo lectura y sin persistencia de credenciales, lo que garantiza que desaparecen del runner una vez clonados.
3. El alumno no controla el workflow (forma parte del template o del repositorio Classroom), de modo que no puede interceptar ni exfiltrar los ficheros de prueba.
4. Aun así, se proporciona feedback: la acción `scacap/action-surefire-report` publica un *check run* que lista cuántas pruebas pasaron o fallaron y, si se desea, los nombres de los métodos fallidos. De esta forma, el estudiante obtiene pistas para mejorar su solución sin que el código de las pruebas quede expuesto, cumpliendo el objetivo pedagógico de evaluar la robustez real del programa manteniendo la integridad del banco de tests.

A.6. Informe de originalidad

A continuación, se adjunta el recibo, que a su vez contiene el identificador (ID) proporcionado por Turnitin, que certifica la autoría del presente trabajo y garantiza su originalidad, en cumplimiento con las normativas académicas vigentes.

Identificador de la entrega en Turnitin: 2691887327

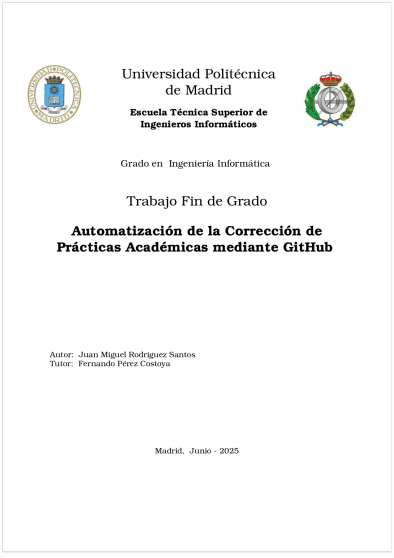
turnitin

Recibo digital

Este recibo confirma quesu trabajo ha sido recibido por Turnitin. A continuación podrá ver la información del recibo con respecto a su entrega.

La primera página de tus entregas se muestra abajo.

Autor de la entrega:	JUAN MIGUEL RODRIGUEZ SANTOS
Título del ejercicio:	Turnitin Memoria Final
Título de la entrega:	TFG_JUAN_MIGUEL_vt3.pdf
Nombre del archivo:	17882_JUAN_MIGUEL_RODRIGUEZ_SANTOS_TFG_JUAN_MIGUE...
Tamaño del archivo:	2M
Total páginas:	80
Total de palabras:	26,600
Total de caracteres:	150,063
Fecha de entrega:	04-jun.-2025 09:45a. m. (UTC+0200)
Identificador de la entrega:	2691887327



Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Automatización de la Corrección de Prácticas Académicas mediante GitHub

Autor: Juan Miguel Rodriguez Santos
Tutor: Fernando Pérez Costoya

Madrid, Junio - 2025

Figura A.7: Comprobante digital generado por Turnitin

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jun 04 10:11:03 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)