|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **DEPARTAMENTO:** | **CIENCIAS DE LA COM- PUTACIÓN** | **CARRERA:** | **INGENIERÍA DE SOFTWARE** | | |
| **ASIGNATURA:** | Pruebas de Software | **NIVEL:** | 6to | **FECHA:** | 09/08/2025 |
| **DOCENTE:** | Ing. Luis Castillo, Mgtr. | **PRÁCTICA N°:** | 2 | **CALIF.:** |  |

CI/CD usando GitHub Actions

# Yeshua Amador Chiliquinga Amaya

**RESUMEN**

En el presente laboratorio, se llevó a cabo la implementación de un flujo de trabajo de

Integración Continua (CI) desde cero para un proyecto en Node.js. El proceso inició con la configuración del entorno de desarrollo, incluyendo la instalación de dependencias clave como Jest para pruebas unitarias y ESLint para el análisis estático de código. Posteriormente, se estableció la comunicación con un repositorio remoto en GitHub. El núcleo de la práctica fue la creación y configuración de un workflow de GitHub Actions, diseñado para automatizar la ejecución de pruebas y la verificación de la calidad del código con cada ‘push‘. Se realizaron pruebas adicionales añadiendo nuevas funcionalidades (factorial y Fibonacci) y se demostró la efectividad del pipeline de CI al provocar un fallo intencionado, observar la notificación de error y, finalmente, corregirlo, validando así el ciclo completo de detección y solución de problemas de forma automatizada.

**Palabras Claves:** GitHub Actions, Integración Continua, Jest.

# INTRODUCCIÓN

La integración continua (CI) es una práctica de desarrollo de software donde los desarrolladores fusionan sus cambios de código en un repositorio central de forma frecuente. El propósito de este laboratorio es familiarizarse con la automatización de tareas esenciales como la instalación de dependencias, la ejecución de pruebas unitarias y la verificación de calidad del código mediante ESLint, todo gestionado a través de GitHub Actions. A través de una aplicación sencilla en Node.js, se experimenta el poder de los flujos automatizados para detectar errores de forma temprana en el ciclo de vida del desarrollo.

# OBJETIVOS

* 1. **Objetivo General**

Configurar un flujo de integración continua (CI) en GitHub Actions que se active automática- mente con cada push a la rama principal del repositorio para automatizar la compilación y las pruebas del proyecto.

* 1. **Objetivos Específicos**

Implementar pruebas unitarias usando Jest para garantizar que la lógica del sistema funcione correctamente.



Aplicar análisis estático de código con ESLint para reforzar buenas prácticas de progra- mación.



Observar y documentar el comportamiento del flujo de CI ante la introducción de pruebas exitosas, fallidas y su posterior corrección.



# MARCO TEÓRICO

* 1. **Entorno de Desarrollo**

**Node.js:** Entorno de ejecución de JavaScript del lado del servidor, que permite construir aplicaciones escalables.



**Jest:** Un framework de pruebas de JavaScript, diseñado para garantizar la corrección de cualquier base de código JavaScript.



**ESLint:** Herramienta de análisis de código estático para identificar y reportar patrones problemáticos encontrados en el código JavaScript.



**GitHub Actions:** Plataforma de automatización que permite construir, probar y desplegar código directamente desde GitHub.



# DESCRIPCIÓN DEL PROCEDIMIENTO

El laboratorio comenzó con la inicialización de un proyecto de Node.js y la configuración del entorno de desarrollo. Se utilizó el comando npm install jest eslint –save-dev para instalar Jest y ESLint como dependencias de desarrollo, lo cual es una buena práctica para mantener separadas las librerías necesarias para la producción de las que solo se usan para pruebas y análisis (Figura 1). A continuación, se crearon los archivos base del proyecto: sum.js para la lógica de negocio inicial, y sum.test.js para su prueba unitaria correspondiente, estableciendo las bases para un desarrollo guiado por pruebas (Figura 2).

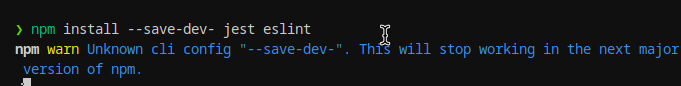


Figura 1: Instalación de Jest y ESLint como dependencias de desarrollo.

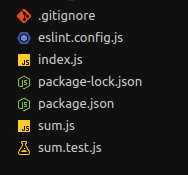


Figura 2: Estructura de archivos inicial del proyecto.

Para gestionar las tareas del proyecto, se modificó el archivo package.json. Se definieron scripts clave como test para ejecutar Jest y lint para ejecutar ESLint. Además, se añadió la directiva "type": "module" para habilitar el uso de módulos ES6 (import/export), modernizando la base del código (Figura 3). Se configuró también un archivo .gitignore para excluir la carpeta node\_modules del control de versiones, evitando así subir al repositorio las dependencias que pueden ser instaladas en cualquier entorno con npm install (Figura 4).

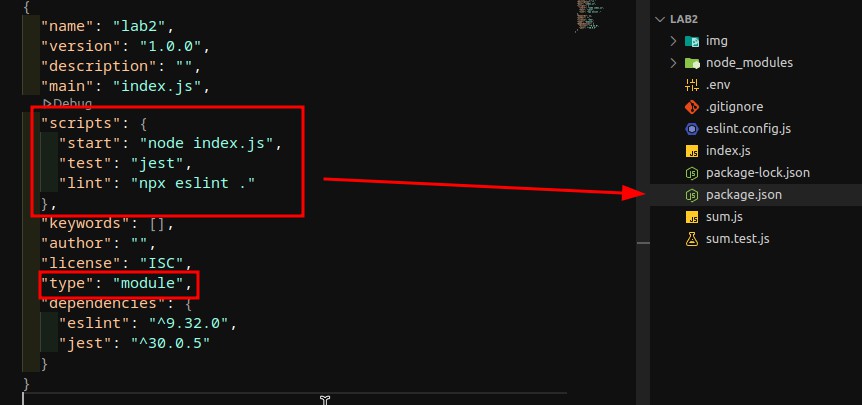


Figura 3: Configuración de scripts y tipo de módulo en package.json.

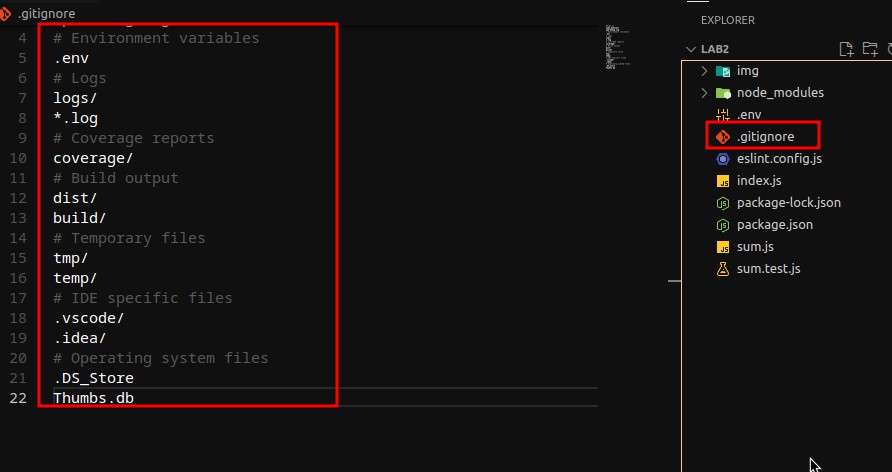


Figura 4: Configuración del archivo .gitignore.

Con el proyecto configurado localmente, el siguiente paso fue establecer el repositorio remoto en GitHub. Se creó un nuevo repositorio vacío (Figura 5) y se ejecutó la secuencia de comandos git init, git add ., git commit, git branch, git remote add y git push para enlazar el proyecto local y subir los archivos por primera vez (Figuras 6 y 7).

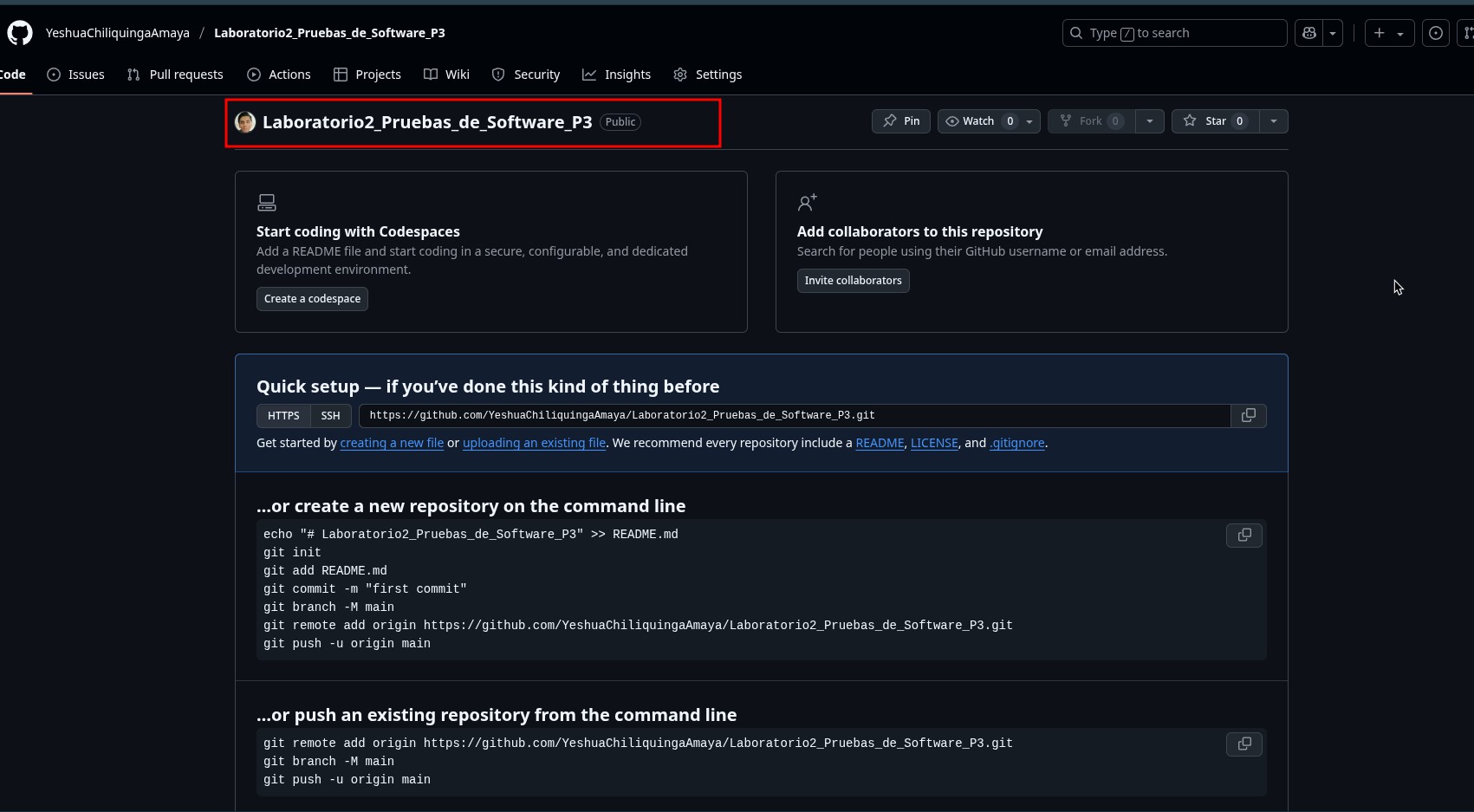


Figura 5: Creación del repositorio vacío en GitHub.

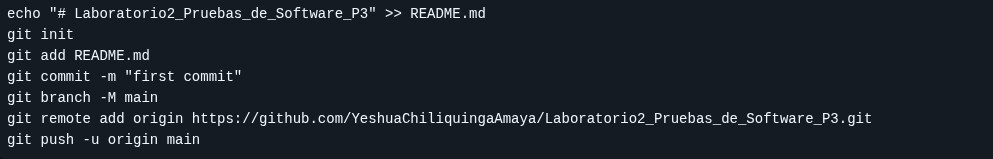


Figura 6: Comandos para subir el proyecto local a GitHub.

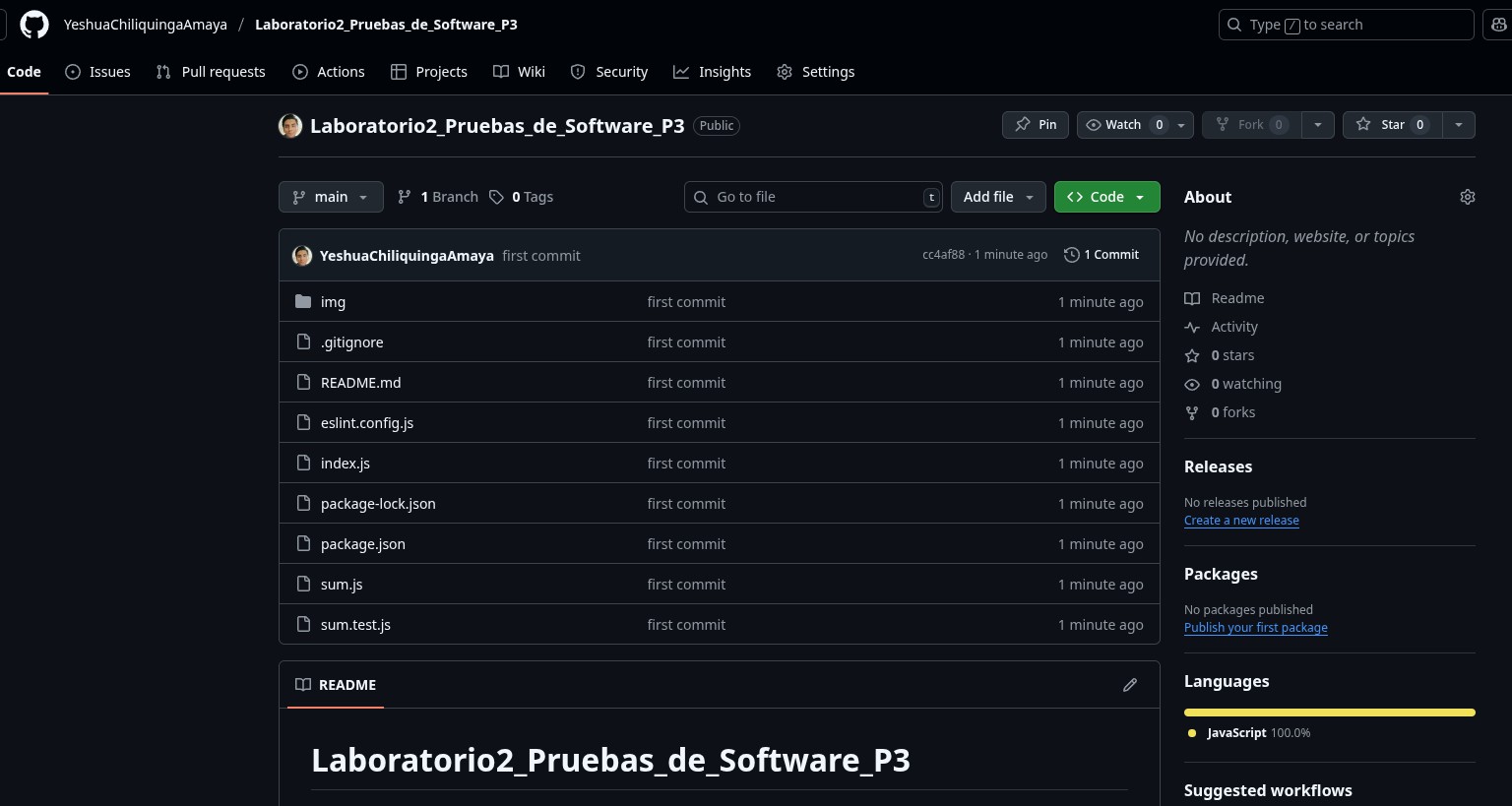


Figura 7: Verificación del repositorio actualizado en GitHub.

Finalmente, se implementó el flujo de CI. Se creó la estructura de carpetas .github/workflows y dentro de ella el archivo ci.yml (Figura 8). En este archivo se definió el pipeline: se especificó que se activara con cada ‘push‘ a la rama ‘main‘, que utilizara un entorno de Node.js v20, y que ejecutara secuencialmente los pasos de ‘checkout‘ del código, instalación de dependencias con npm ci, análisis de código con npm run lint y ejecución de pruebas con npm run test (Figura 9). Al hacer ‘push‘ de este nuevo archivo, GitHub Actions detectó el workflow y lo ejecutó automáti- camente. El resultado fue un éxito, indicado por una marca de verificación verde, confirmando que la configuración inicial era correcta y el pipeline funcional (Figura 10).



Figura 8: Creación del archivo de configuración del workflow.

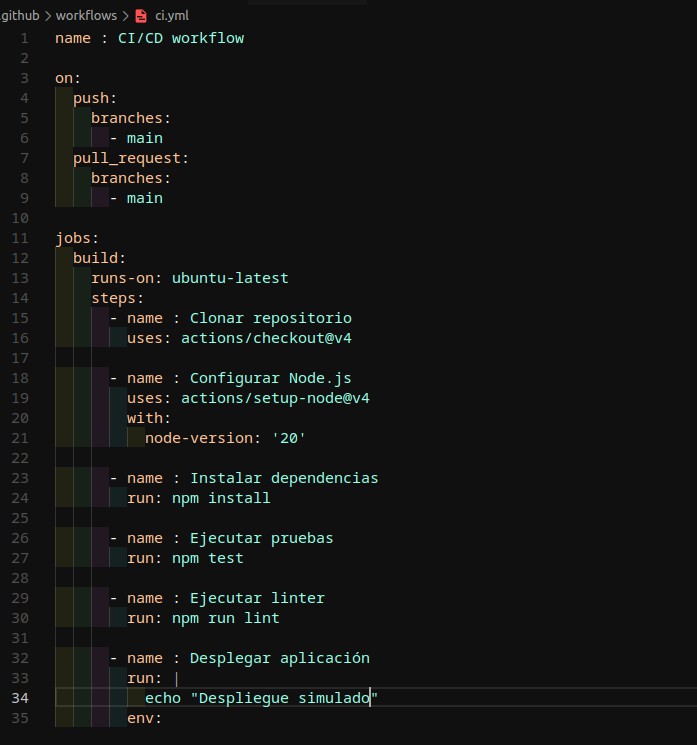


Figura 9: Contenido del archivo ci.yml con los pasos del workflow.

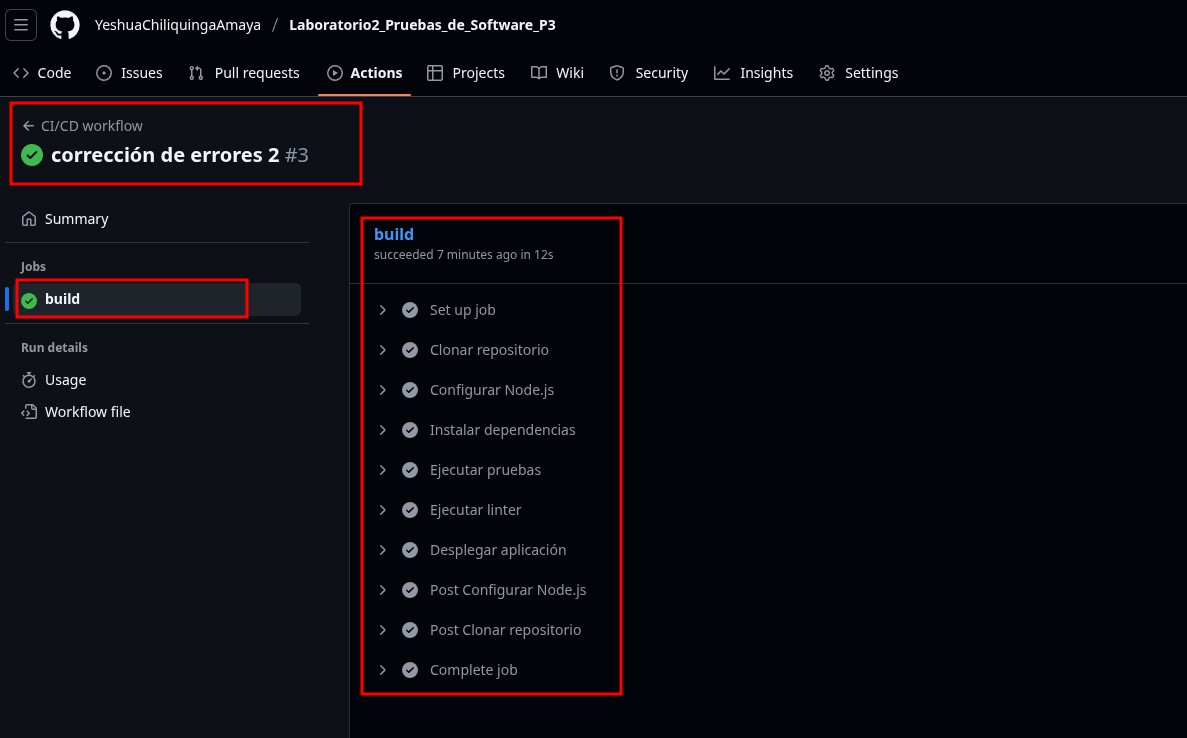


Figura 10: Flujo de trabajo ejecutado exitosamente en GitHub Actions.

# PREGUNTAS/ACTIVIDADES

Para validar la extensibilidad del proyecto y la robustez del pipeline, se procedió a añadir nueva funcionalidad. Se implementaron dos funciones matemáticas clásicas, factorial y Fibonacci, en el archivo math.js (Figuras 11 y 12). Siguiendo las buenas prácticas, para cada nueva función se crearon sus correspondientes pruebas unitarias en un nuevo archivo math.test.js. Estas pruebas cubrían casos base y casos típicos para asegurar que la lógica implementada era correcta (Figura 13). Al realizar el ‘push‘ con estas nuevas adiciones, el flujo de trabajo de CI en GitHub Actions se disparó automáticamente. El pipeline ejecutó todos los pasos definidos, incluyendo las nuevas pruebas, y concluyó exitosamente, lo que se visualizó con un build en verde. Esto demostró que las nuevas funcionalidades estaban correctamente implementadas y no rompían ninguna parte existente del código (Figura 14).

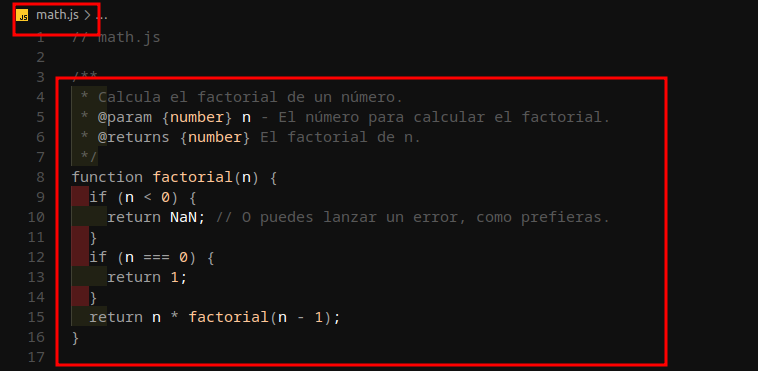


Figura 11: Función de factorial añadida en math.js.

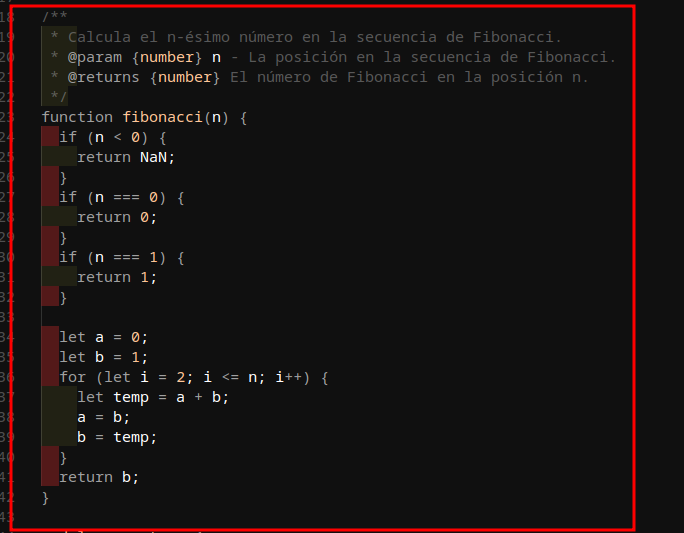


Figura 12: Función de Fibonacci añadida en math.js.

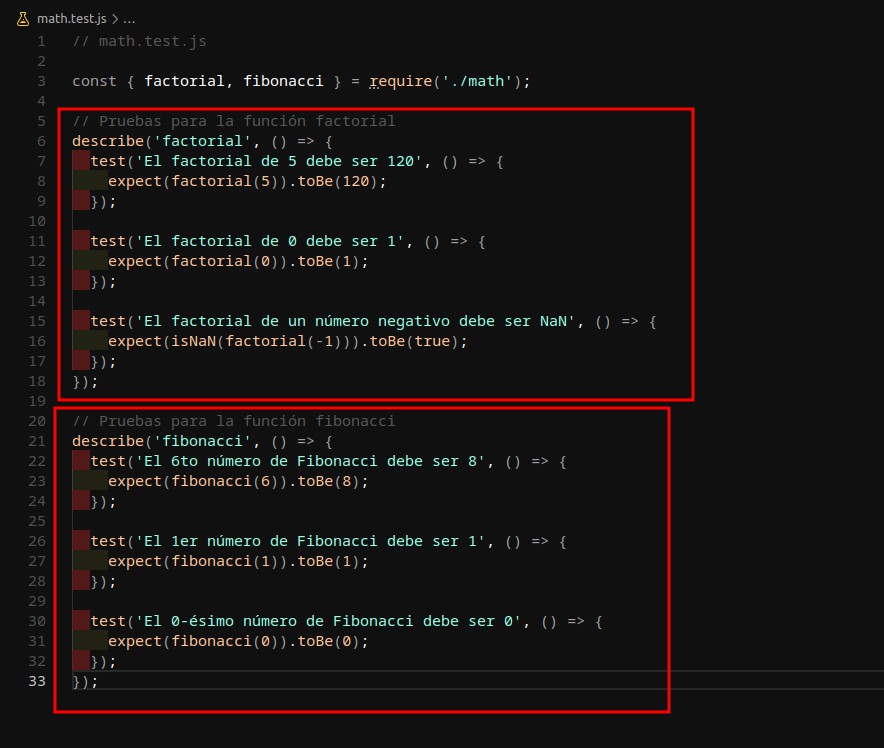


Figura 13: Pruebas unitarias para factorial y Fibonacci en math.test.js.

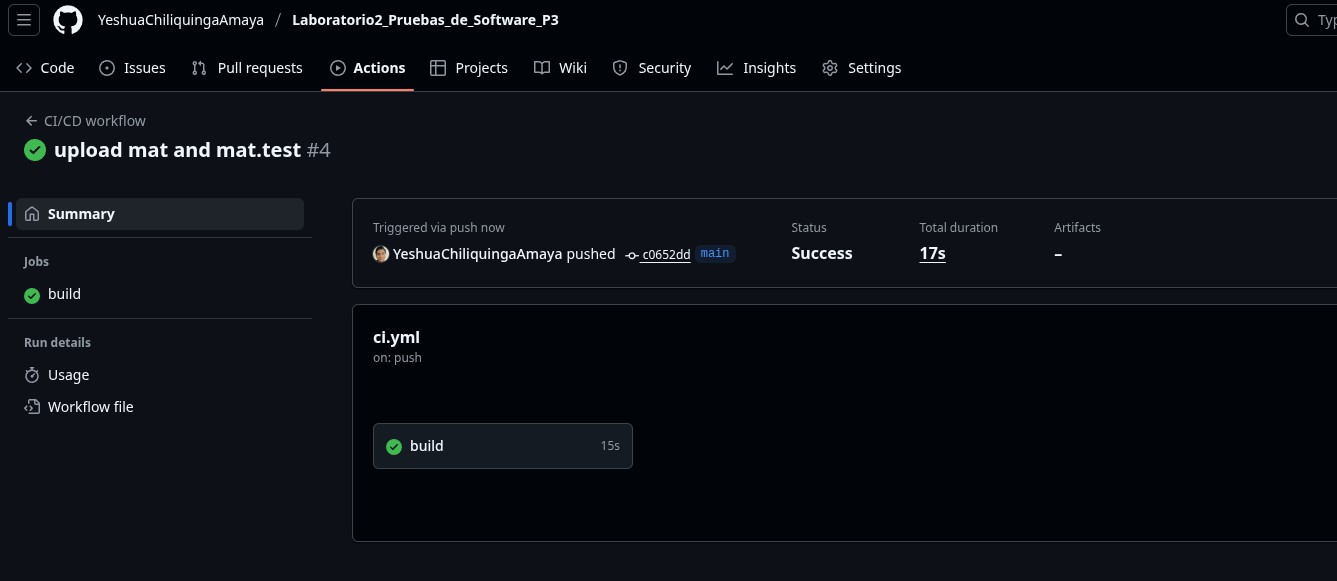


Figura 14: El build en GitHub Actions se completa con éxito tras añadir las nuevas pruebas.

El siguiente paso fue simular un escenario real de desarrollo: la introducción de un error. Se modificó intencionadamente el archivo sum.test.js para que una prueba fallara; se cambió la aserción de que ‘2 + 3‘ es ‘5‘ a que es ‘100‘ (Figura 15). Inmediatamente después de hacer

‘push‘ de este cambio, GitHub Actions inició el pipeline y, como era de esperar, falló en el paso de ejecución de pruebas. La interfaz de GitHub mostró una "Xroja junto al commit, indicando visualmente el fallo. Al hacer clic en los detalles, se podía ver exactamente qué prueba había fallado y el motivo, proporcionando una retroalimentación rápida y precisa sobre el origen del problema (Figura 16).

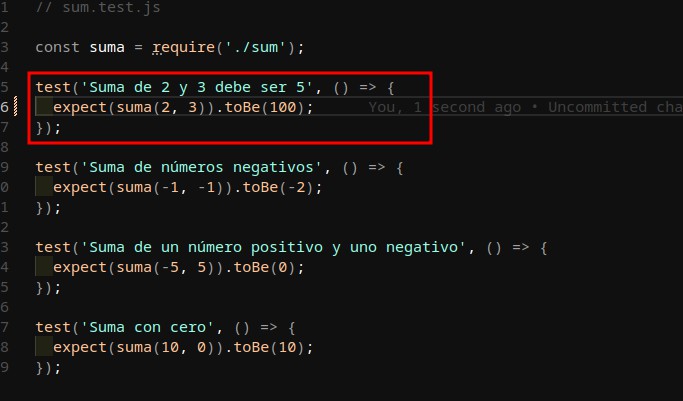


Figura 15: Modificación de sum.test.js para provocar un fallo intencional.

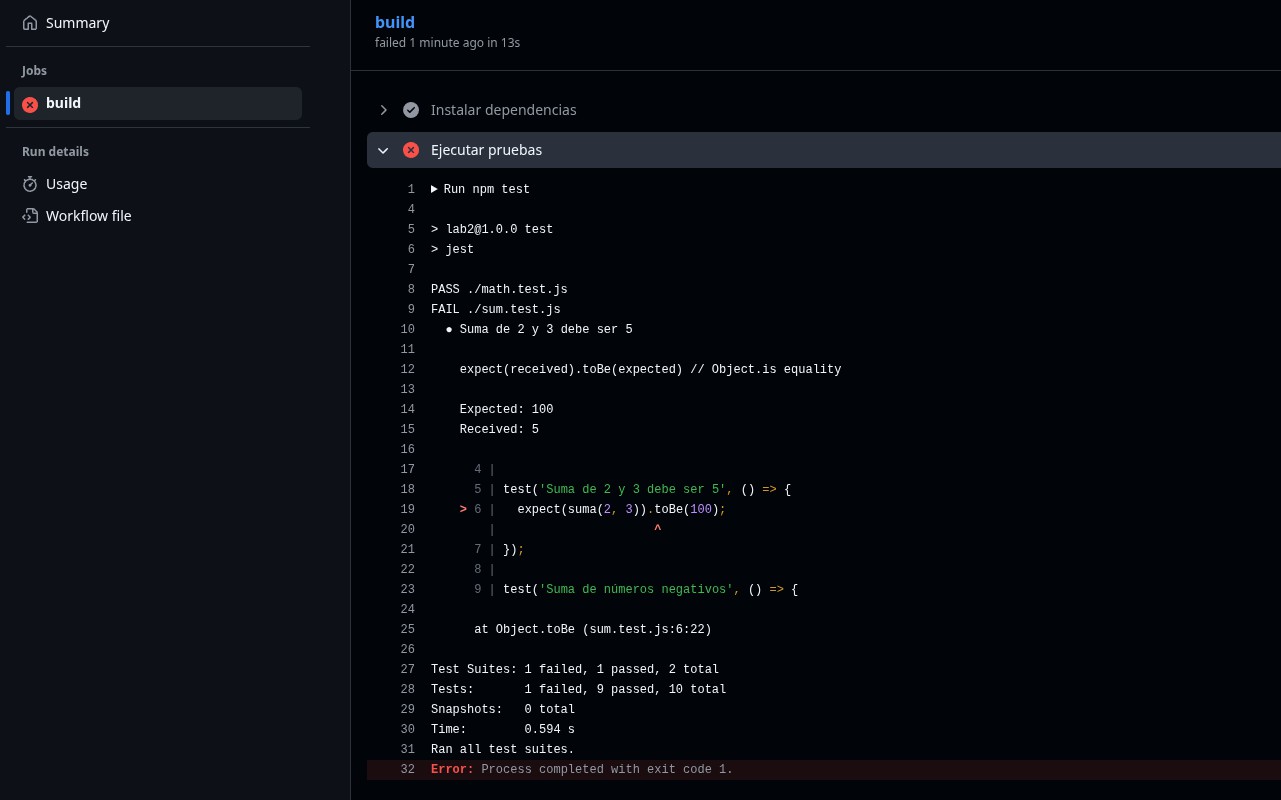


Figura 16: El flujo de CI falla y GitHub Actions reporta el error.

Finalmente, se completó el ciclo de detección y corrección. Se revirtió el cambio en sum.test.js a su estado original y correcto, donde la suma de ‘2 + 3‘ esperaba el resultado ‘5‘ (Figura 17). Tras confirmar y subir la corrección con un nuevo ‘push‘, el pipeline de CI se ejecutó una vez más. Esta vez, todas las pruebas pasaron, el linter no reportó problemas y el flujo de trabajo se completó

con éxito, mostrando nuevamente la marca de verificación verde (Figura 18). Este proceso validó la efectividad del pipeline de CI como una red de seguridad que previene la integración de código defectuoso en la rama principal.

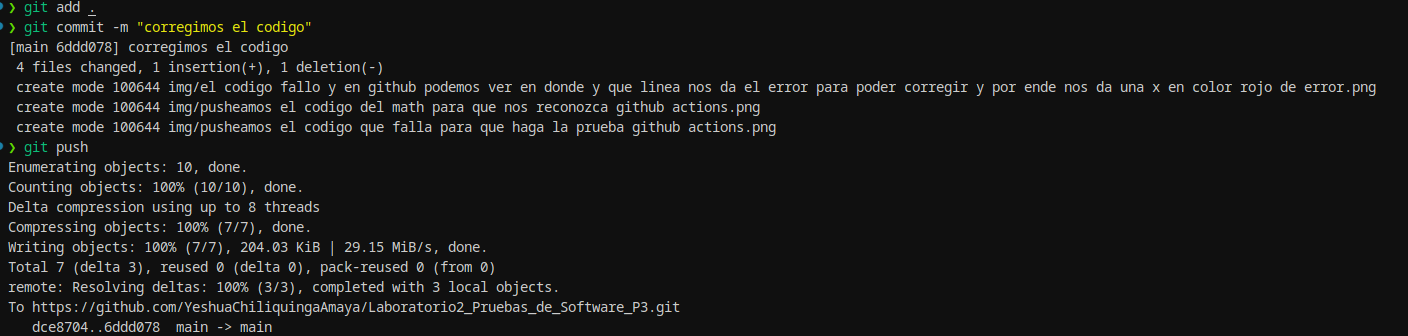


Figura 17: Corrección del código en el archivo de prueba.

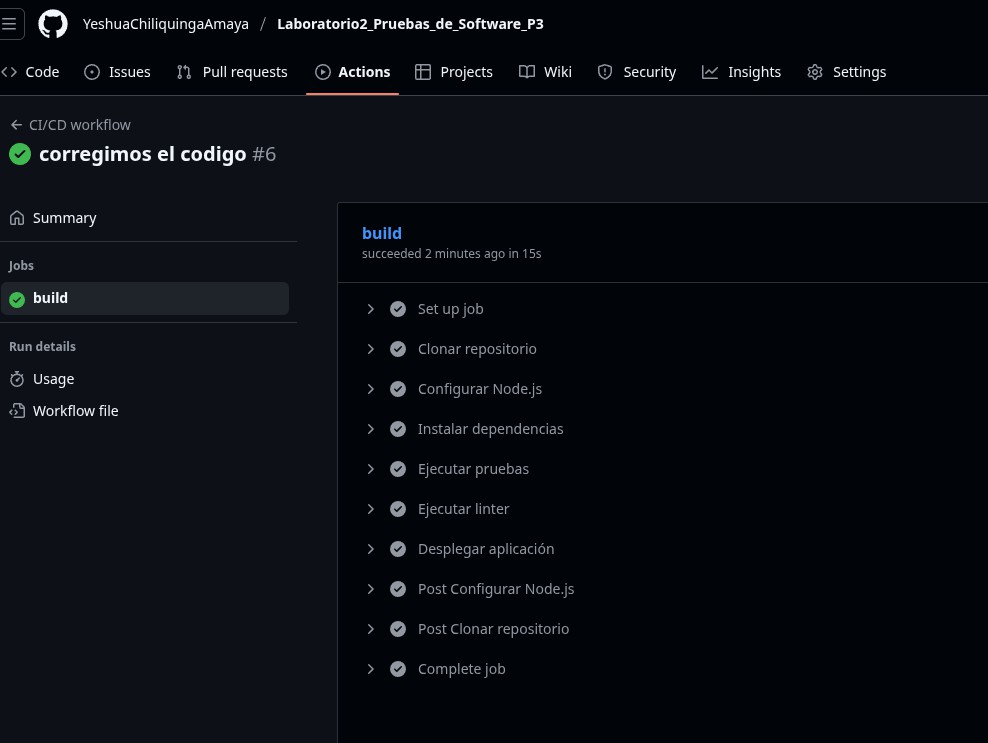


Figura 18: El flujo de trabajo vuelve a ser exitoso tras la corrección.

# CONCLUSIONES

La implementación de un flujo de Integración Continua con GitHub Actions es una herramienta eficaz para automatizar la validación del código, asegurando que cada cambio cumpla con los estándares de calidad y funcionalidad definidos antes de ser fusionado.



La retroalimentación inmediata que proporciona el pipeline de CI es fundamental. La capacidad de detectar errores al instante mediante notificaciones visuales (rojo/verde) permite a los desarrolladores corregirlos rápidamente, evitando que los problemas se integren en la rama principal y afecten a otros miembros del equipo.



El uso de un archivo de configuración como ci.yml estandariza el entorno de pruebas y validación. Esto garantiza que el código se evalúe siempre de la misma manera (misma versión de Node.js, mismas dependencias, mismos comandos), eliminando el clásico problema de “en mi máquina funciona” y asegurando la consistencia en todo el ciclo de desarrollo.



# RECOMENDACIONES

Es recomendable separar los trabajos (jobs) en el archivo de workflow si las tareas son independientes (p. ej., un job para linting y otro para pruebas) para optimizar el tiempo de ejecución al permitir que se ejecuten en paralelo, lo que puede reducir significativamente el tiempo de espera para obtener retroalimentación.



Utilizar secretos de GitHub (‘GitHub Secrets‘) para manejar credenciales, tokens o claves de API en lugar de escribirlas directamente en el archivo .yml. Esta práctica es crucial para mejorar la seguridad del proyecto y evitar la exposición accidental de información sensible en el repositorio de código.



Implementar el almacenamiento en caché de dependencias (caching) dentro del workflow de GitHub Actions. Para proyectos con muchas dependencias, como los de Node.js, guardar en caché la carpeta node\_modules puede acelerar drásticamente las ejecuciones posteriores del pipeline, ya que se evita tener que descargar e instalar todas las librerías desde cero en cada ejecución.



# BIBLIOGRAFÍA

1. GitHub, “GitHub Actions Documentation”, 2025. [En línea]. Disponible: https://docs.github. com/en/actions
2. Jest, “Jest - Delightful JavaScript Testing”, 2025. [En línea]. Disponible: https://jestjs.io/