

<b>DEPARTAMENTO:</b>	<b>CIENCIAS DE LA COMPUTACIÓN</b>	<b>CARRERA:</b>	<b>INGENIERÍA DE SOFTWARE</b>		
<b>ASIGNATURA:</b>	Pruebas de Software	<b>NIVEL:</b>	6to	<b>FECHA:</b>	09/08/2025
<b>DOCENTE:</b>	Ing. Luis Castillo, Mgtr.	<b>PRÁCTICA N°:</b>	2	<b>CALIF.:</b>	

## 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.

## 1. 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.

## 2. OBJETIVOS

### 2.1 Objetivo General

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

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

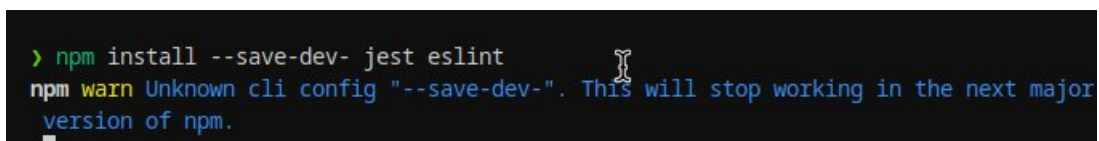
## 3. MARCO TEÓRICO

### 3.1 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.

## 4. 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).



```
> npm install --save-dev- jest eslint
npm warn Unknown cli config "--save-dev-". This will stop working in the next major
version of npm.
```

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

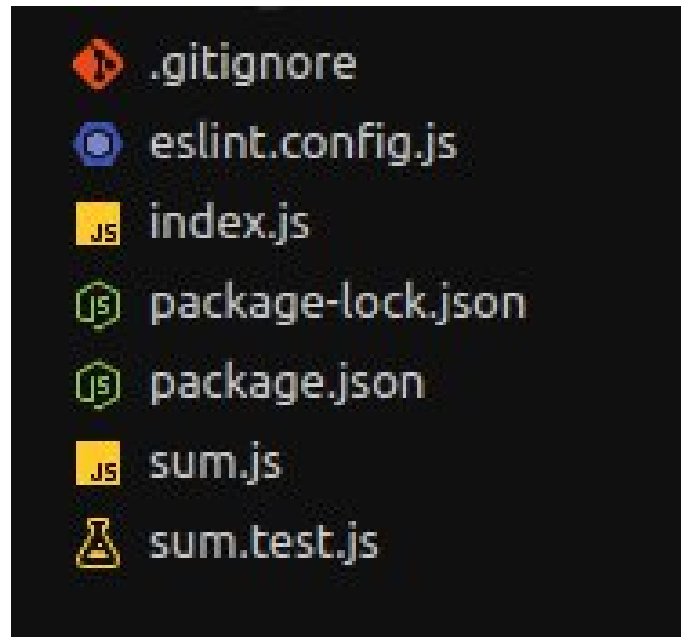


Figura 2: Estructura de archivos inicial del proyecto.

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).

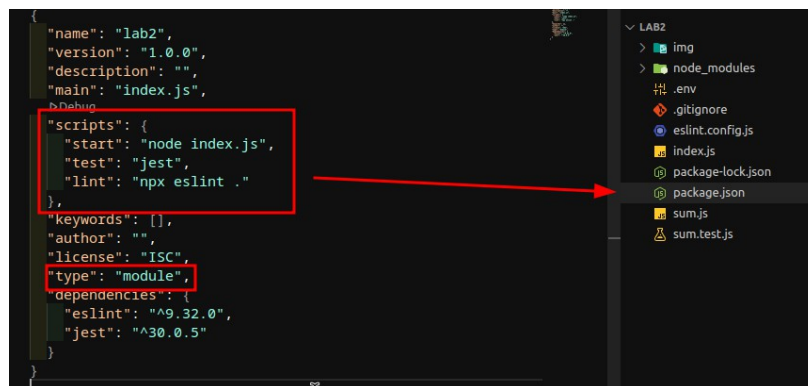


Figura 3: Configuración de scripts y tipo de módulo en package.json.

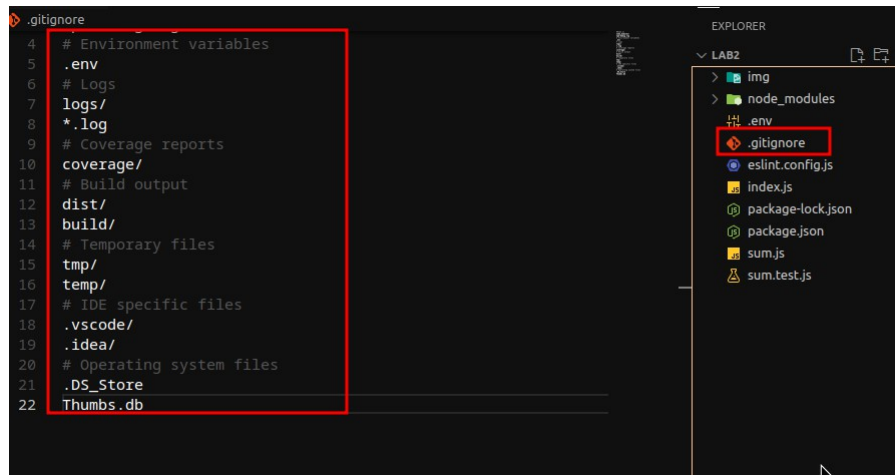


Figura 4: Configuración del archivo .gitignore.

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).

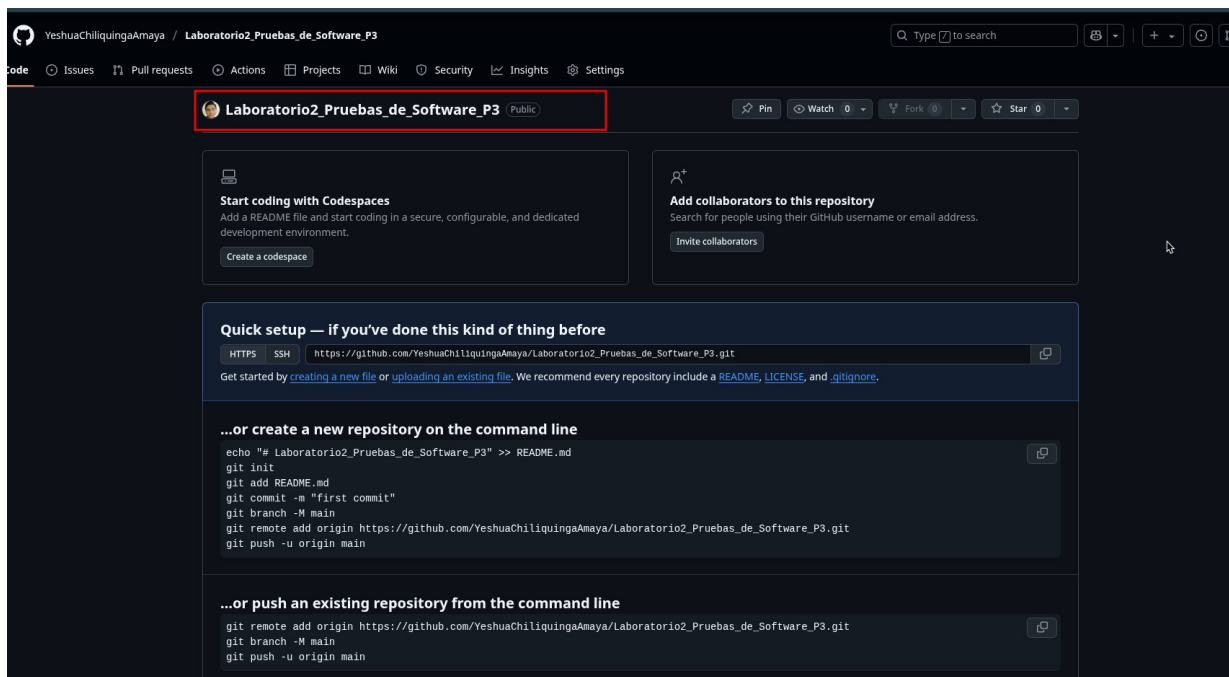


Figura 5: Creación del repositorio vacío en GitHub.

```
echo "# Laboratorio2_Pruebas_de_Software_P3" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/YeshuaChiliQuingaAmaya/Laboratorio2_Pruebas_de_Software_P3.git
git push -u origin main
```

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

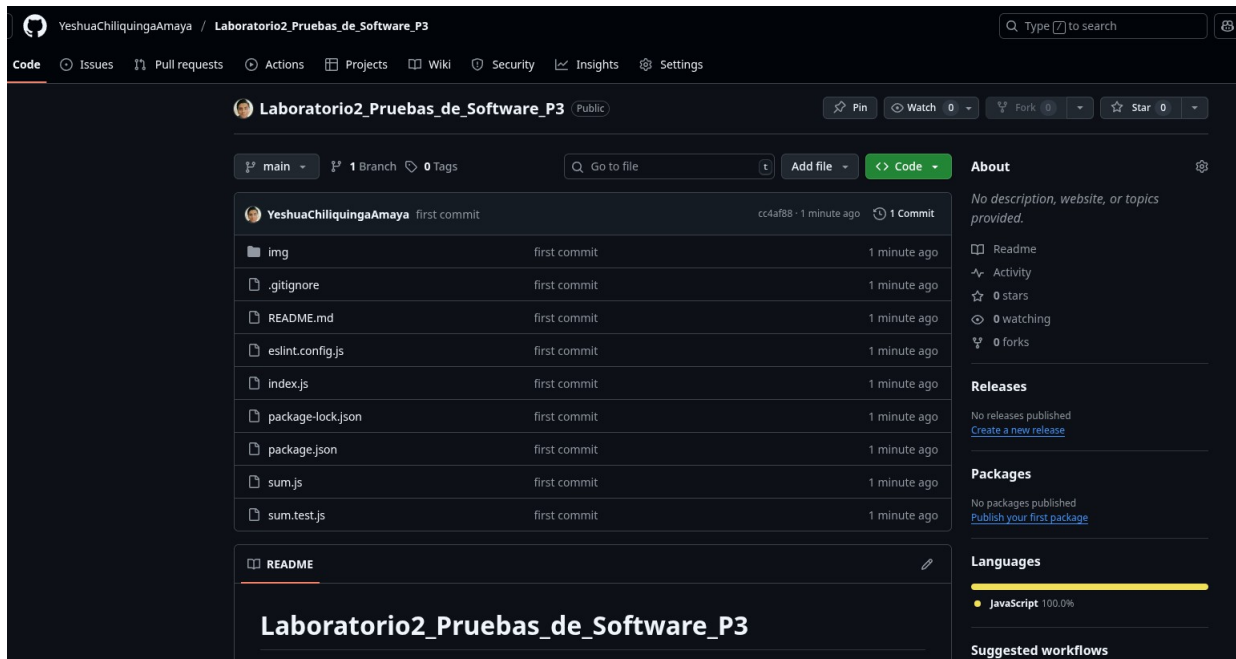


Figura 7: Verificación del repositorio actualizado en GitHub.

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 automáticamente. 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).

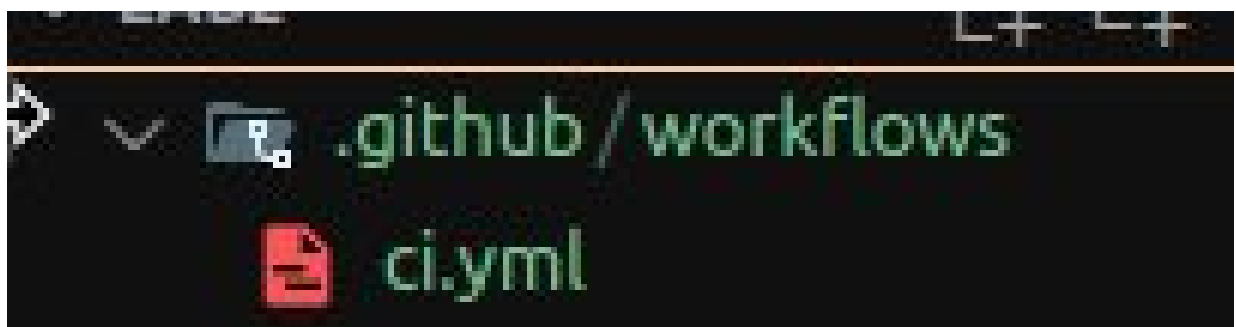
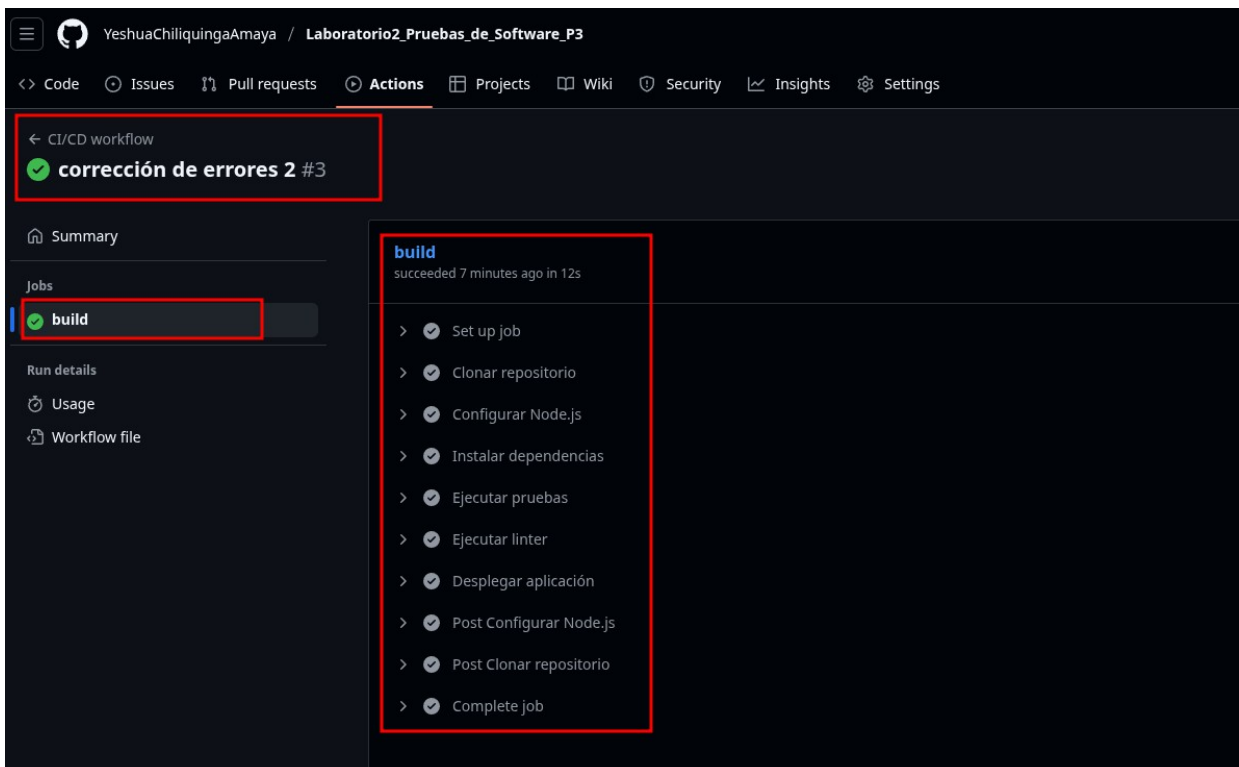


Figura 8: Creación del archivo de configuración del workflow.

```
github > workflows > ci.yml
1  name : CI/CD workflow
2
3  on:
4    push:
5      branches:
6        - main
7    pull_request:
8      branches:
9        - main
10
11 jobs:
12   build:
13     runs-on: ubuntu-latest
14     steps:
15       - name : Clonar repositorio
16         uses: actions/checkout@v4
17
18       - name : Configurar Node.js
19         uses: actions/setup-node@v4
20         with:
21           node-version: '20'
22
23       - name : Instalar dependencias
24         run: npm install
25
26       - name : Ejecutar pruebas
27         run: npm test
28
29       - name : Ejecutar linter
30         run: npm run lint
31
32       - name : Desplegar aplicación
33         run: |
34           echo "Despliegue simulado"
35     env:
```

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



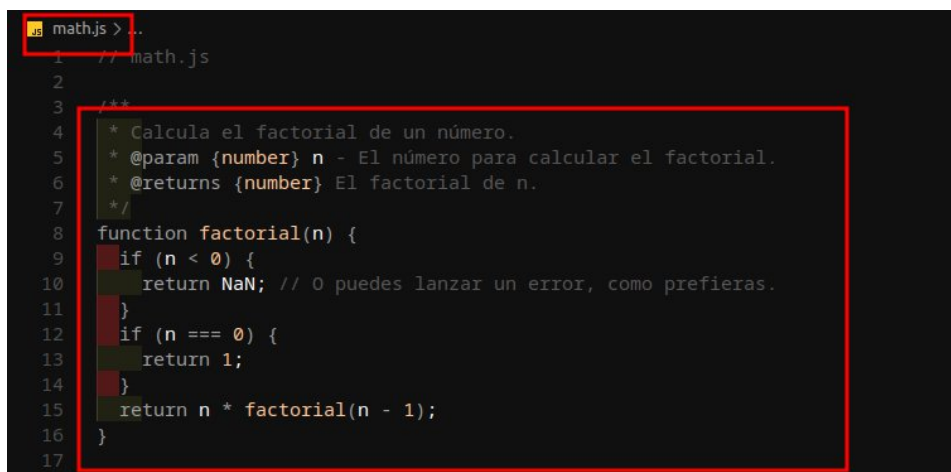
The screenshot displays the GitHub Actions interface for a repository named 'Laboratorio2\_Pruebas\_de\_Software\_P3'. The 'Actions' tab is selected, showing a workflow named 'CI/CD workflow' with a status of 'corrección de errores 2 #3'. The 'build' job is highlighted, indicating it 'succeeded 7 minutes ago in 12s'. A list of steps for the 'build' job is shown, all marked as successful with checkmarks:

- > Set up job
- > Clonar repositorio
- > Configurar Node.js
- > Instalar dependencias
- > Ejecutar pruebas
- > Ejecutar linter
- > Desplegar aplicación
- > Post Configurar Node.js
- > Post Clonar repositorio
- > Complete job

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

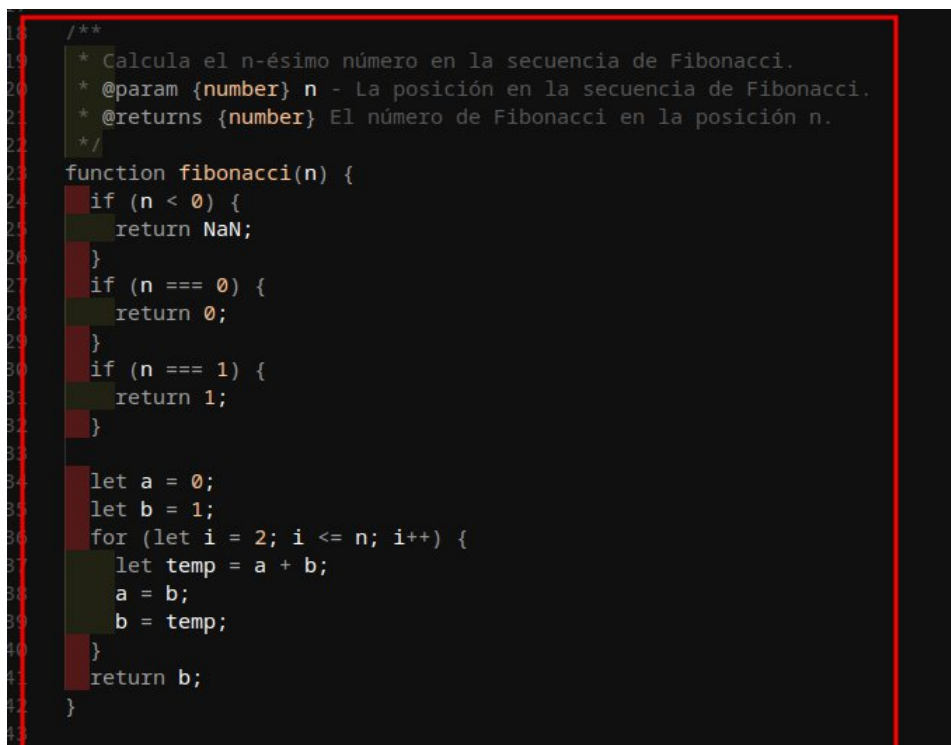
## 5. PREGUNTAS/ACTIVIDADES

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).



```
1 // math.js
2
3 /**
4  * Calcula el factorial de un número.
5  * @param {number} n - El número para calcular el factorial.
6  * @returns {number} El factorial de n.
7  */
8 function factorial(n) {
9   if (n < 0) {
10    return NaN; // 0 puedes lanzar un error, como prefieras.
11   }
12   if (n === 0) {
13    return 1;
14   }
15   return n * factorial(n - 1);
16 }
17
```

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



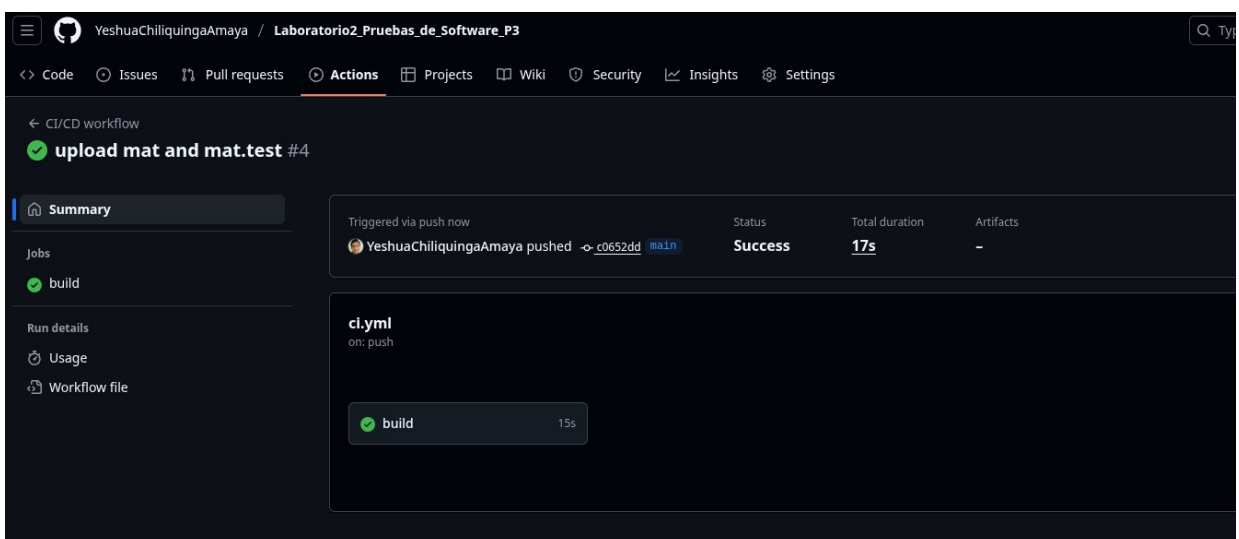
```
18 /**
19  * Calcula el n-ésimo número en la secuencia de Fibonacci.
20  * @param {number} n - La posición en la secuencia de Fibonacci.
21  * @returns {number} El número de Fibonacci en la posición n.
22  */
23 function fibonacci(n) {
24   if (n < 0) {
25    return NaN;
26   }
27   if (n === 0) {
28    return 0;
29   }
30   if (n === 1) {
31    return 1;
32   }
33
34   let a = 0;
35   let b = 1;
36   for (let i = 2; i <= n; i++) {
37    let temp = a + b;
38    a = b;
39    b = temp;
40   }
41   return b;
42 }
43
```

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



```
math.test.js > ...
1 // math.test.js
2
3 const { factorial, fibonacci } = require('./math');
4
5 // Pruebas para la función factorial
6 describe('factorial', () => {
7   test('El factorial de 5 debe ser 120', () => {
8     expect(factorial(5)).toBe(120);
9   });
10
11   test('El factorial de 0 debe ser 1', () => {
12     expect(factorial(0)).toBe(1);
13   });
14
15   test('El factorial de un número negativo debe ser NaN', () => {
16     expect(isNaN(factorial(-1))).toBe(true);
17   });
18 });
19
20 // Pruebas para la función fibonacci
21 describe('fibonacci', () => {
22   test('El 6to número de Fibonacci debe ser 8', () => {
23     expect(fibonacci(6)).toBe(8);
24   });
25
26   test('El 1er número de Fibonacci debe ser 1', () => {
27     expect(fibonacci(1)).toBe(1);
28   });
29
30   test('El 0-ésimo número de Fibonacci debe ser 0', () => {
31     expect(fibonacci(0)).toBe(0);
32   });
33 });
```

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



The screenshot shows the GitHub Actions interface for a workflow named 'upload mat and mat.test #4'. The workflow is triggered by a push to the 'main' branch. The status is 'Success' with a total duration of 17s. The 'build' job is shown as completed with a duration of 15s. The workflow file is 'cl.yml' and it runs on 'push'.

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

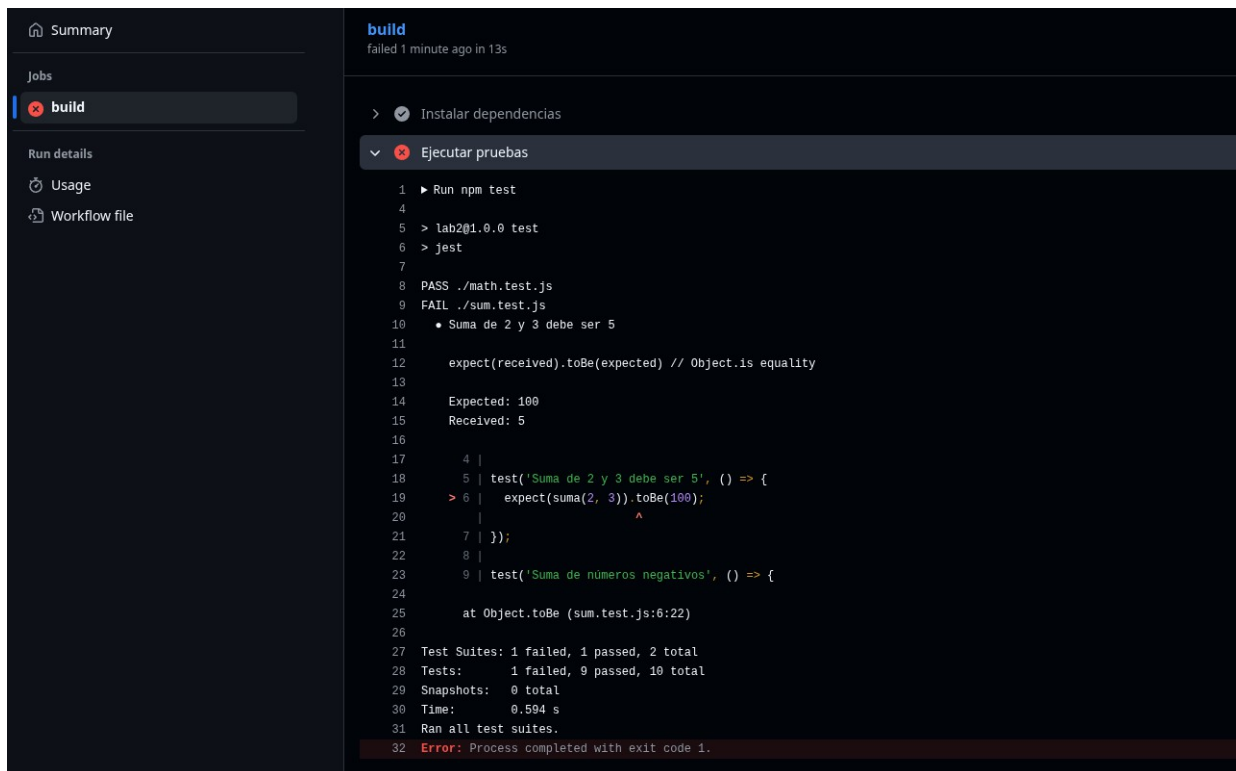
El siguiente paso fue simular un escenario real de desarrollo: la introducción de un error. Se modificó intencionadamente el archivo sum.test.js para que una prueba fallara; se cambió la aserción de que '2 + 3' es '5' a que es '100' (Figura 15). Inmediatamente después de hacer



‘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).

```
1 // sum.test.js
2
3 const suma = require('./sum');
4
5 test('Suma de 2 y 3 debe ser 5', () => {
6   expect(suma(2, 3)).toBe(100);
7 });
8
9 test('Suma de números negativos', () => {
10  expect(suma(-1, -1)).toBe(-2);
11 });
12
13 test('Suma de un número positivo y uno negativo', () => {
14  expect(suma(-5, 5)).toBe(0);
15 });
16
17 test('Suma con cero', () => {
18  expect(suma(10, 0)).toBe(10);
19 });
```

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



```
Summary
Jobs
  build
Run details
Usage
Workflow file

build
failed 1 minute ago in 13s

> [x] Instalar dependencias
  [x] Ejecutar pruebas
    1 Run npm test
    4
    5 > lab2@1.0.0 test
    6 > jest
    7
    8 PASS ./math.test.js
    9 FAIL ./sum.test.js
    10   • Suma de 2 y 3 debe ser 5
    11
    12     expect(received).toBe(expected) // Object.is equality
    13
    14     Expected: 100
    15     Received: 5
    16
    17       4 |
    18       5 | test('Suma de 2 y 3 debe ser 5', () => {
    19     > 6 |   expect(suma(2, 3)).toBe(100);
    20         |                                     ^
    21       7 | });
    22       8 |
    23     9 | test('Suma de números negativos', () => {
    24
    25     at Object.toBe (sum.test.js:6:22)
    26
    27 Test Suites: 1 failed, 1 passed, 2 total
    28 Tests:      1 failed, 9 passed, 10 total
    29 Snapshots:  0 total
    30 Time:       0.594 s
    31 Ran all test suites.
    32 Error: Process completed with exit code 1.
```

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.

```
> git add .
> git commit -m "corregimos el codigo"
[main 6ddd078] corregimos el codigo
4 files changed, 1 insertion(+), 1 deletion(-)
create mode 100644 img/el codigo fallo y en github podemos ver en donde y que línea nos da el error para poder corregir y por ende nos da una x en color rojo de error.png
create mode 100644 img/pusheamos el codigo del math para que nos reconozca github actions.png
create mode 100644 img/pusheamos el codigo que falla para que haga la prueba github actions.png
> git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 204.03 KiB | 29.15 MiB/s, done.
Total 7 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/YeshuaChiliQuingaAmaya/Laboratorio2_Pruebas_de_Software_P3.git
dce8704..6ddd078 main -> main
```

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

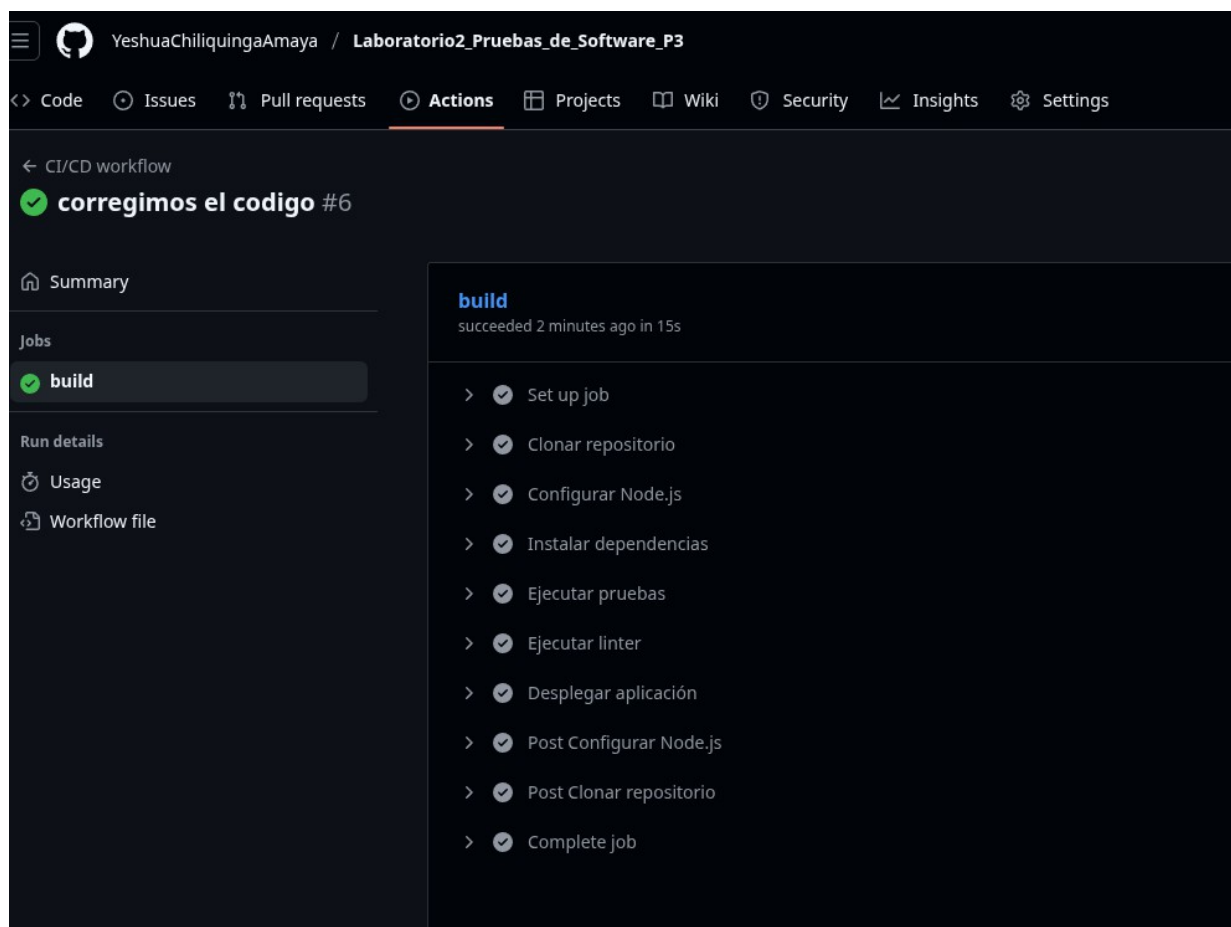


Figura 18: El flujo de trabajo vuelve a ser exitoso tras la corrección.

## 6. 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.

## **7. 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.

## **8. 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/>