UNIVERSIDAD DE LAS FUERZAS ARMADAS
ESPE

Sede Matriz Sangolquí

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

INFORME DE LABORATORIO

Práctica N° 6

CI/CD usando GitHub Actions

Integración y Entrega Continua con automatización de pruebas

Asignatura:
Pruebas de Software

Estudiante:
Denise Rea

Docente:
Ing. Enrique Calvopiña, Mgtr.

Nivel:
6to Semestre

NRC:
22431

Período:
202555

Laboratorio: H-205

Enero 2026

Índice

1. Introducción

La Integración Continua (CI) es una práctica fundamental del desarrollo de software moderno que permite a los equipos de desarrollo detectar y corregir errores de manera temprana en el ciclo de vida del software. Este laboratorio tiene como propósito 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 ello gestionado a través de **GitHub Actions**.

A través de una aplicación sencilla en **Node.js**, se experimentará el poder de los flujos automatizados y se comprenderá la importancia de detectar errores temprano en el ciclo de vida del desarrollo. GitHub Actions permite configurar pipelines de CI/CD directamente en el repositorio, ejecutando automáticamente pruebas y verificaciones con cada cambio en el código.

El flujo de trabajo implementado incluye:

- **Checkout del código:** Obtención del código fuente del repositorio.
- **Configuración del entorno:** Instalación de Node.js y dependencias.
- **Análisis estático:** Verificación de calidad con ESLint.
- **Pruebas unitarias:** Ejecución de tests con Jest.
- **Simulación de despliegue:** Preparación para entrega continua.

2. Objetivos

2.1. Objetivo General

Configurar un flujo de integración continua (CI) en GitHub Actions que se active automáticamente con cada push o pull request a la rama principal del repositorio, implementando pruebas unitarias y análisis estático de código.

2.2. Objetivos Específicos

1. Configurar un flujo de integración continua (CI) en GitHub Actions que se active automáticamente con cada push o pull request a la rama principal del repositorio.
2. Implementar pruebas unitarias usando Jest, garantizando que la lógica del sistema funcione correctamente en cada actualización del código.
3. Aplicar análisis estático de código con ESLint, reforzando buenas prácticas de programación y detección temprana de errores o inconsistencias.
4. Simular un proceso de despliegue automatizado, demostrando cómo se automatizan las etapas previas al paso final de entrega continua (CD).

3. Marco Teórico

3.1. Integración Continua (CI)

La Integración Continua es una práctica de desarrollo de software donde los desarrolladores fusionan sus cambios de código en un repositorio central de forma frecuente, preferiblemente varias veces al día. Cada integración es verificada mediante una compilación automatizada y pruebas, permitiendo detectar errores rápidamente.

3.2. Entrega Continua (CD)

La Entrega Continua es una extensión de la CI que garantiza que el código pueda ser desplegado a producción en cualquier momento. Mientras CI se enfoca en la automatización de pruebas, CD automatiza el proceso completo de lanzamiento.

3.3. GitHub Actions

GitHub Actions es una plataforma de automatización integrada en GitHub que permite crear flujos de trabajo (workflows) personalizados directamente en el repositorio. Características principales:

- **Workflows:** Procesos automatizados configurables mediante archivos YAML.
- **Events:** Triggers que inician los workflows (push, pull_request, etc.).
- **Jobs:** Conjuntos de pasos que se ejecutan en el mismo runner.
- **Actions:** Comandos reutilizables que se pueden compartir.

3.4. Jest

Jest es un framework de pruebas de JavaScript desarrollado por Facebook, diseñado para garantizar la corrección del código. Características:

- Fácil configuración con zero-config para proyectos JavaScript.
- Ejecución paralela de pruebas para mayor velocidad.
- Mocking integrado y snapshots testing.
- Cobertura de código incluida.

3.5. ESLint

ESLint es una herramienta de análisis estático para identificar patrones problemáticos en código JavaScript. Permite:

- Detectar errores de sintaxis y lógica.

- Asegurar consistencia en el estilo de código.
- Aplicar mejores prácticas automáticamente.
- Personalizar reglas según las necesidades del proyecto.

4. Materiales y Equipos

Categoría	Descripción
Sistema Operativo	Windows 10 o superior
Hardware	Procesador Intel Core i7-6700T o superior, 12GB RAM, 480GB SSD
Software	Node.js, Visual Studio Code, Git
Plataformas	GitHub, GitHub Actions
Dependencias	Express, Jest, ESLint
Conectividad	Acceso a Internet

Cuadro 1: Materiales y equipos utilizados en la práctica

5. Desarrollo de la Práctica

5.1. Parte 1: Establecimiento de la Estructura del Proyecto Base

5.1.1. Paso 1: Creación de la estructura básica

Se creó la estructura de carpetas del proyecto con los archivos necesarios para el desarrollo del laboratorio de CI/CD.

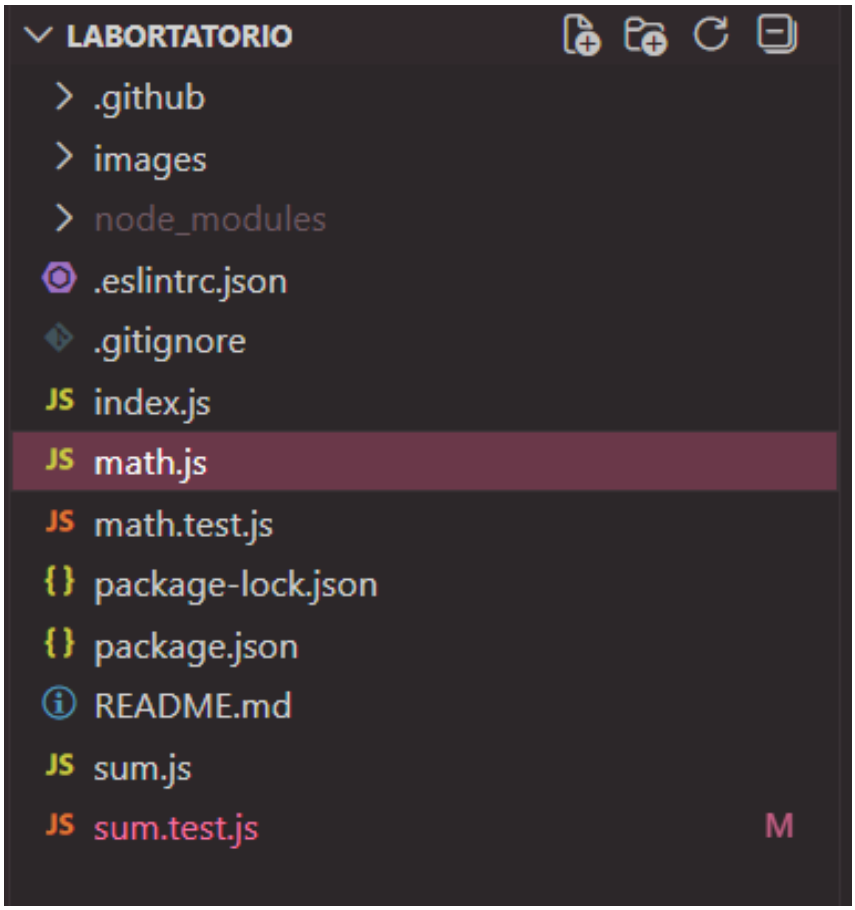

Figura 1: Estructura de carpetas del proyecto en Visual Studio Code

5.1.2. Paso 2: Instalación de dependencias necesarias

a) Creación del archivo package.json Se inicializó el proyecto con npm para generar el archivo de configuración:

Figura 2: Ejecución del comando npm init -y para crear package.json

b) Instalación de Express Se instaló el framework Express para crear el servidor:

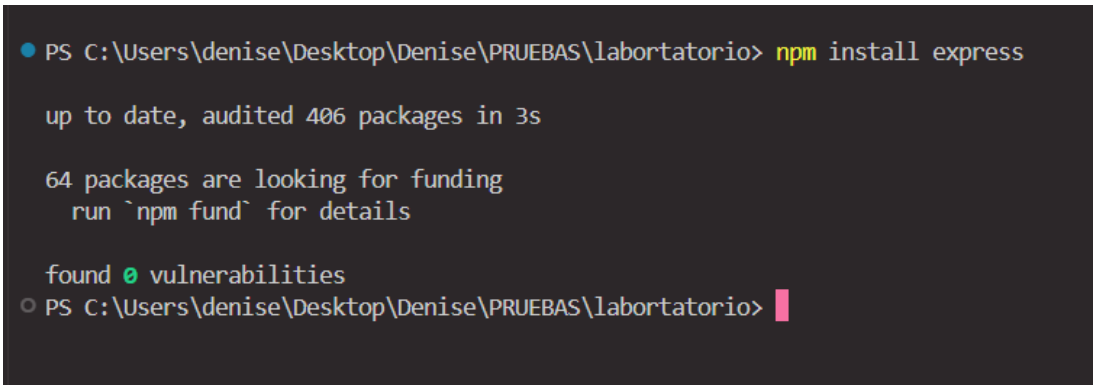

Figura 3: Instalación de Express con npm install express

c) Instalación de dependencias de desarrollo Se configuraron Jest y ESLint como dependencias de desarrollo utilizando el comando `npm install -save-dev jest eslint`.

5.2. Parte 2: Creación de Archivos Base

5.2.1. Paso 1: Crear archivo index.js

Se implementó un servidor Express con endpoints básicos:

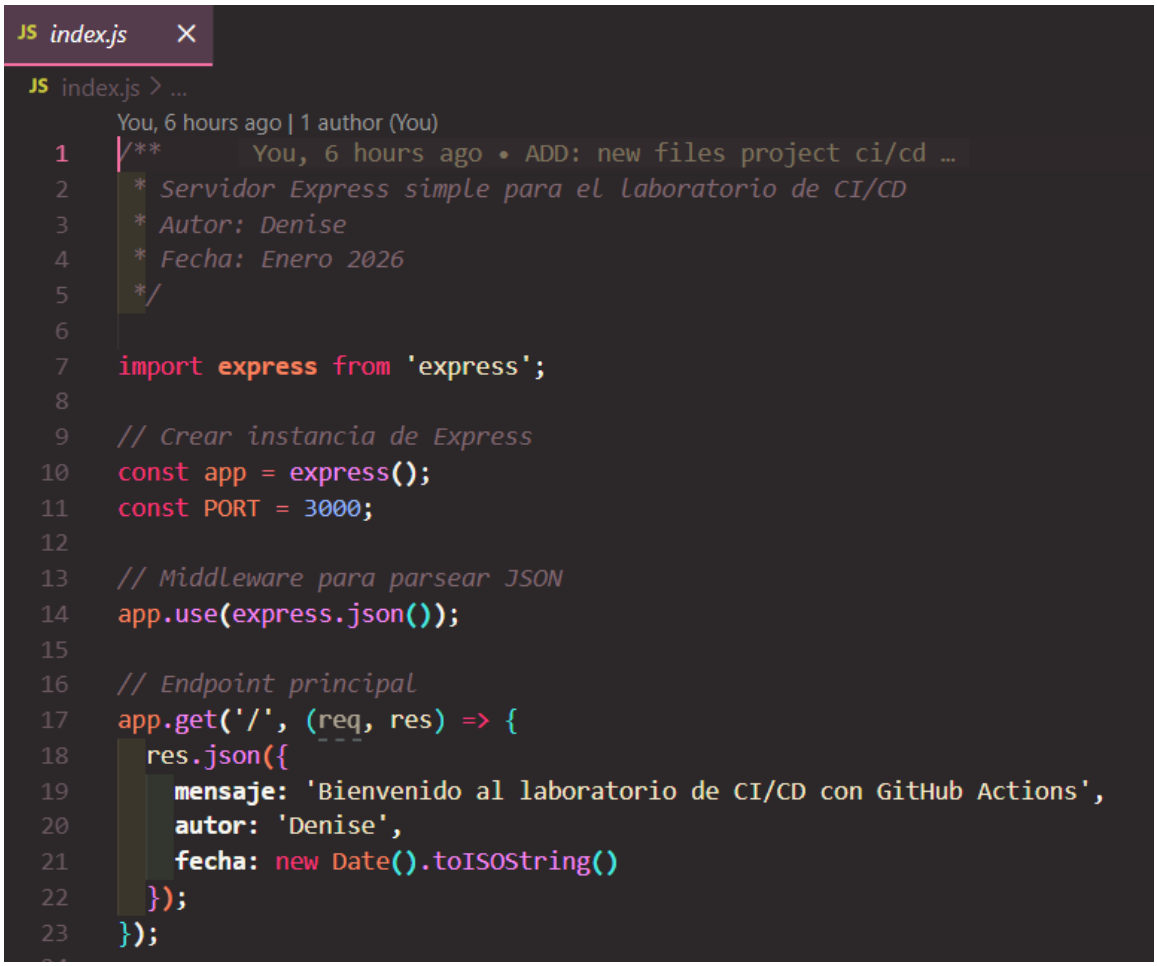

Figura 4: Código fuente del servidor Express (index.js)

```
/**
 * Servidor Express simple para el laboratorio de CI/CD
 * Autor: Denise
 * Fecha: Enero 2026
 */
import express from 'express';

const app = express();
const PORT = 3000;

// Middleware para parsear JSON
app.use(express.json());

// Endpoint principal
app.get('/', (req, res) => {
  res.json({
    mensaje: 'Bienvenido al laboratorio de CI/CD con GitHub
       Actions',
    autor: 'Denise',
    fecha: new Date().toISOString()
  });
});
```

```
22
23 // Endpoint de salud
24 app.get('/health', (req, res) => {
25   res.json({ status: 'OK', timestamp: Date.now() });
26 });
27
28 // Levantar el servidor
29 app.listen(PORT, () => {
30   console.log(`Servidor en http://localhost:${PORT}`);
31 });
32
33 export default app;
```

Listing 1: Contenido del archivo index.js

5.2.2. Paso 2: Crear archivo sum.js

Se implementó una función simple de suma para las pruebas unitarias:

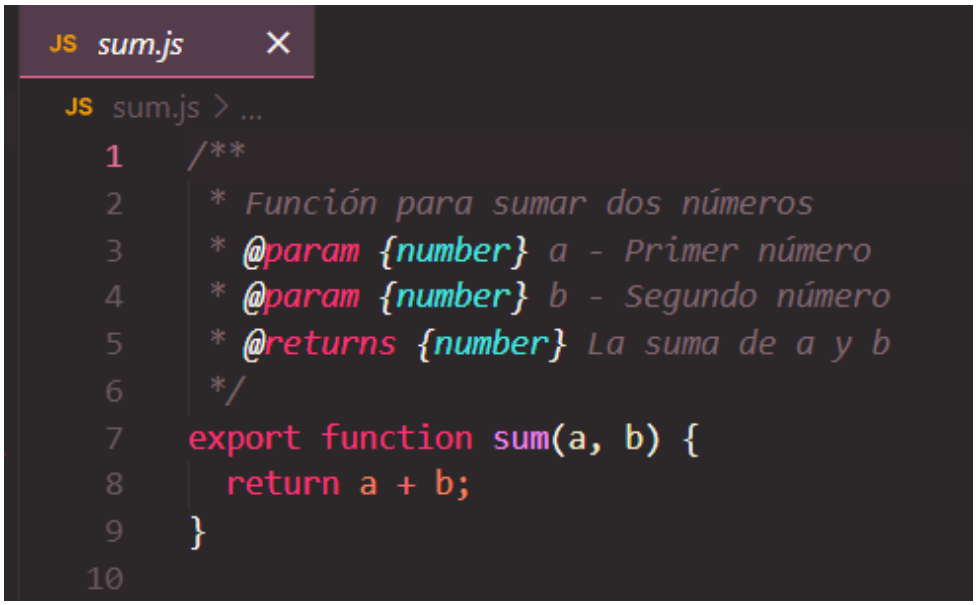

Figura 5: Código fuente de la función de suma (sum.js)

```
1 /**
2  * Funcion para sumar dos numeros
3  * @param {number} a - Primer numero
4  * @param {number} b - Segundo numero
5  * @returns {number} La suma de a y b
6  */
7 export function sum(a, b) {
8   return a + b;
9 }
```

Listing 2: Contenido del archivo sum.js

5.2.3. Paso 3: Crear archivo sum.test.js

Se crearon las pruebas unitarias para la función de suma:

```
JS sum.test.js M
JS sum.test.js > ...
1  /**
2   * Pruebas unitarias para la función sum
3   */
4
5  import { sum } from './sum.js';
6
7  describe('Función sum()', () => {
8    test('suma 1 + 2 es igual a 3', () => {
9      expect(sum(1, 9)).toBe(3);
10   });
11
12   test('suma números negativos: -1 + -1 es igual a -2', () => {
13     expect(sum(-1, 1)).toBe(-2);
14   });
15
16   test('suma con cero: 5 + 0 es igual a 5', () => {
17     expect(sum(5, 0)).toBe(5);
18   });
19
20   test('suma números decimales: 0.1 + 0.2', () => {
21     expect(sum(0.1, 0.2)).toBeCloseTo(0.3);
22   });
23 });
24
```

Figura 6: Código fuente de las pruebas unitarias (sum.test.js)

```
1  /**
2   * Pruebas unitarias para la funcion sum
3   */
4  import { sum } from './sum.js';
5
6  describe('Funcion sum()', () => {
7    test('suma 1 + 2 es igual a 3', () => {
8      expect(sum(1, 2)).toBe(3);
9    });
10
11   test('suma numeros negativos: -1 + -1 es igual a -2', () => {
12     expect(sum(-1, -1)).toBe(-2);
13   });
14
15   test('suma con cero: 5 + 0 es igual a 5', () => {
```

```
    expect(sum(5, 0)).toBe(5);
  });

  test('suma numeros decimales: 0.1 + 0.2', () => {
    expect(sum(0.1, 0.2)).toBeCloseTo(0.3);
  });
});
```

Listing 3: Contenido del archivo sum.test.js

5.2.4. Paso 4: Configurar package.json

Se agregaron los scripts necesarios para ejecutar el servidor, las pruebas y el linter:

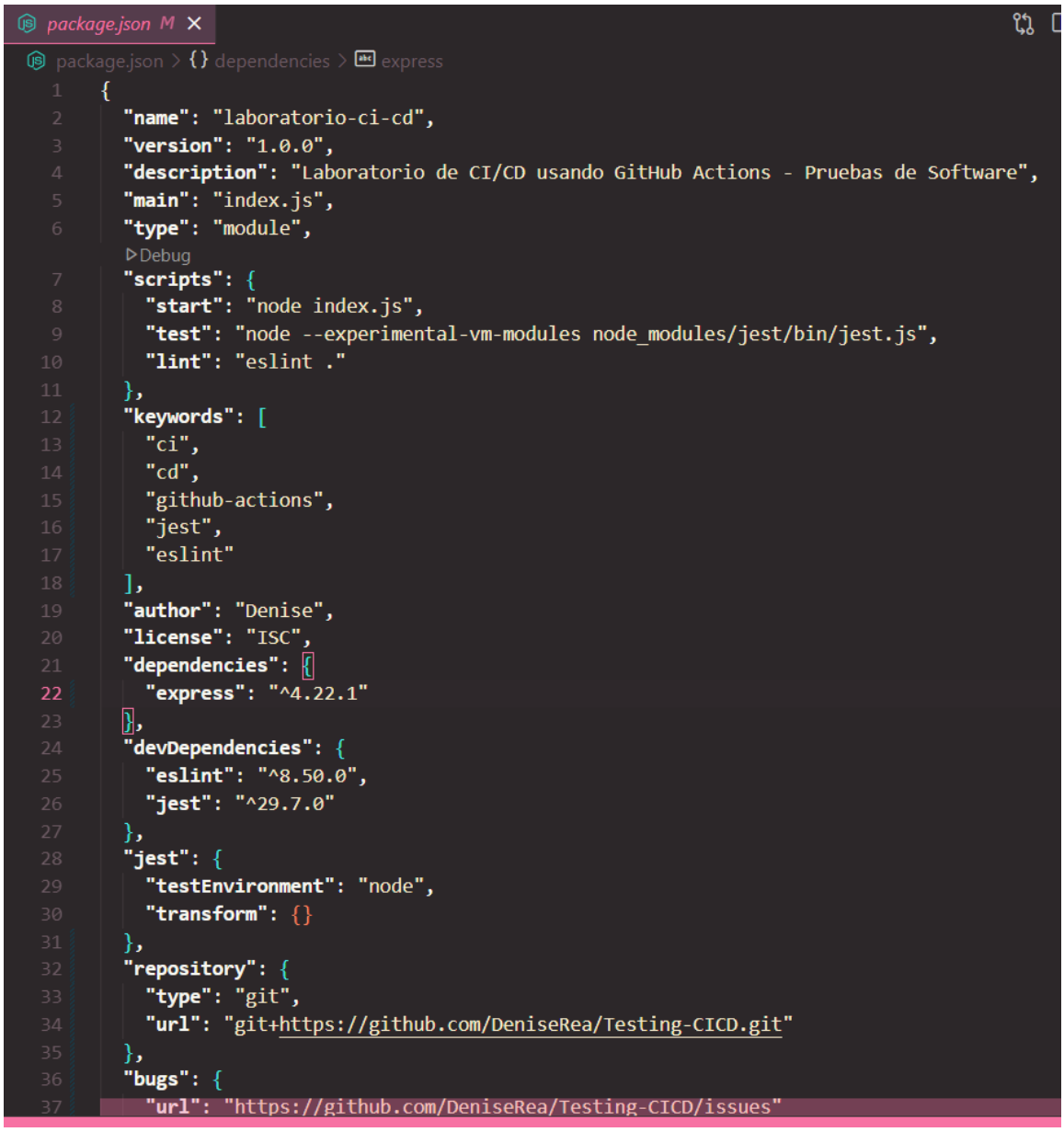

Figura 7: Configuración del archivo package.json con scripts

```
{
  "name": "laboratorio-ci-cd",
  "version": "1.0.0",
  "description": "Laboratorio de CI/CD usando GitHub Actions",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js",
    "test": "node --experimental-vm-modules node_modules/jest/
       bin/jest.js",
    "lint": "eslint ."
  },
  "author": "Denise",
  "license": "ISC",
  "dependencies": {
    "express": "^4.22.1"
  },
  "devDependencies": {
    "eslint": "^8.50.0",
    "jest": "^29.7.0"
  },
  "jest": {
    "testEnvironment": "node",
    "transform": {}
  }
}
```

Listing 4: Contenido del archivo package.json

5.2.5. Paso 5: Crear el archivo ESLint

Se configuró ESLint con reglas básicas para asegurar la calidad del código:

Figura 8: Configuración de ESLint (.eslintrc.json)

```
{
  "env": {
    "node": true,
    "es2021": true,
```

```
    "jest": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": "latest",
    "sourceType": "module"
  },
  "rules": {
    "indent": ["error", 2],
    "linebreak-style": "off",
    "quotes": ["error", "single"],
    "semi": ["error", "always"],
    "no-unused-vars": "warn",
    "no-console": "off",
    "eqeqeq": ["error", "always"],
    "curly": ["error", "all"],
    "brace-style": ["error", "1tbs"],
    "comma-dangle": ["error", "never"],
    "no-trailing-spaces": "error"
  }
}
```

Listing 5: Contenido del archivo .eslintrc.json

5.3. Parte 3: Configuración de Git y GitHub Actions

5.3.1. Paso 1: Crear repositorio en GitHub

Se creó un nuevo repositorio vacío en GitHub para alojar el proyecto.

5.3.2. Paso 2: Ejecución de comandos Git

Se ejecutaron los comandos necesarios para inicializar el repositorio local y conectarlo con GitHub:

```
git init
git add .
git commit -m "Proyecto base con CI"
git branch -M main
git remote add origin https://github.com/DeniseRea/Testing-CICD.
    git
git push -u origin main
```

Listing 6: Comandos Git para inicializar y subir el proyecto

5.3.3. Paso 3: Crear el workflow de GitHub Actions

Se creó el archivo `.github/workflows/ci.yml` con la configuración del pipeline de CI:

ci.yml M

.github > workflows > ci.yml

```yaml

name: CI - Integración Continua

# Triggers: Ejecutar el workflow en push y pull requests a main
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

# Definición de trabajos
jobs:
  # Job principal: test
  test:
    name: Pruebas y Análisis de Código
    runs-on: ubuntu-latest

    # Estrategia de matriz para probar múltiples versiones de Node (opcional)
    strategy:
      matrix:
        node-version: [18.x]

    steps:
      # Step 1: Checkout del código fuente
      - name: Checkout del repositorio
        uses: actions/checkout@v4

      # Step 2: Configurar Node.js
      - name: Configurar Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}

      # Step 3: Instalar dependencias
      - name: Instalar dependencias
        run: npm install

```

Figura 9: Código fuente del workflow de GitHub Actions (ci.yml)

```yaml
name: CI - Integracion Continua

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    name: Pruebas y Analisis de Codigo
    runs-on: ubuntu-latest
    strategy:
```

```yaml
    matrix:
      node-version: [18.x]

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

    - name: Configurar Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v4
      with:
        node-version: ${{ matrix.node-version }}

    - name: Instalar dependencias
      run: npm install

    - name: Analisis estatico con ESLint
      run: npm run lint

    - name: Ejecutar pruebas unitarias
      run: npm test

    - name: Simulacion de despliegue
      run: |
        echo "Todas las pruebas pasaron exitosamente"
        echo "Codigo validado por ESLint"
        echo "Listo para despliegue"
```

Listing 7: Contenido del archivo ci.yml

5.3.4. Paso 4: Probar la CI

Se realizó un push al repositorio para verificar que el workflow se ejecutara correctamente:

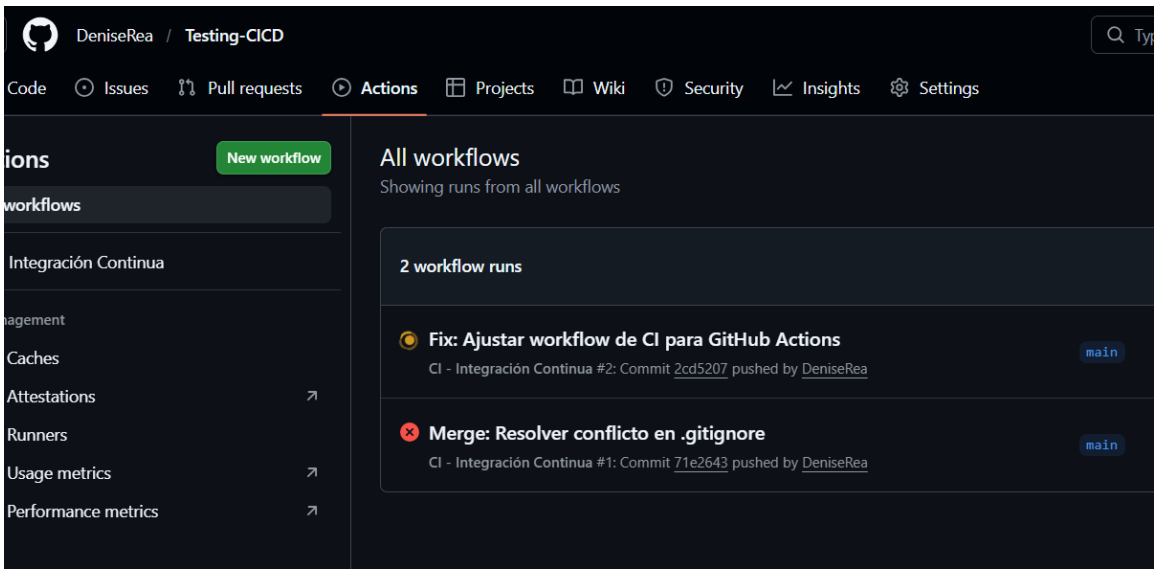

Figura 10: Vista de los workflows en ejecución en GitHub Actions

6. Sección de Preguntas/Actividades

6.1. Actividad 1: Agregar más pruebas unitarias

Se implementaron dos funciones matemáticas adicionales (factorial y fibonacci) con sus correspondientes pruebas unitarias.

6.1.1. Archivo math.js

math.test.js

```
/**
 * Pruebas unitarias para funciones matemáticas
 * Pruebas para factorial y fibonacci
 */

import { factorial, fibonacci } from './math.js';

describe('Función factorial()', () => {
  test('factorial de 0 es 1', () => {
    expect(factorial(0)).toBe(1);
  });

  test('factorial de 1 es 1', () => {
    expect(factorial(1)).toBe(1);
  });

  test('factorial de 5 es 120', () => {
    expect(factorial(5)).toBe(120);
  });

  test('factorial de 10 es 3628800', () => {
    expect(factorial(10)).toBe(3628800);
  });

  test('factorial de número negativo lanza error', () => {
    expect(() => factorial(-1)).toThrow('El factorial no está definido para números n
  });
});

describe('Función fibonacci()', () => {
  test('fibonacci(0) es 0', () => {
    expect(fibonacci(0)).toBe(0);
  });

  test('fibonacci(1) es 1', () => {
    expect(fibonacci(1)).toBe(1);
  });
```

Figura 11: Funciones matemáticas adicionales (math.js y math.test.js)

```
/**
 * Funciones matematicas adicionales
 * Autor: Denise
```

```
 */

/**
 * Calcula el factorial de un numero
 */
export function factorial(n) {
  if (n < 0) {
    throw new Error('El factorial no esta definido para numeros negativos');
  }
  if (n === 0 || n === 1) {
    return 1;
  }
  let resultado = 1;
  for (let i = 2; i <= n; i++) {
    resultado *= i;
  }
  return resultado;
}

/**
 * Calcula el numero de Fibonacci en la posicion n
 */
export function fibonacci(n) {
  if (n < 0) {
    throw new Error('La posicion debe ser un numero no negativo');
  }
  if (n === 0) return 0;
  if (n === 1) return 1;

  let prev = 0;
  let curr = 1;
  for (let i = 2; i <= n; i++) {
    const temp = curr;
    curr = prev + curr;
    prev = temp;
  }
  return curr;
}
```

Listing 8: Contenido del archivo math.js

6.1.2. Archivo math.test.js

```
import { factorial, fibonacci } from './math.js';

describe('Funcion factorial()', () => {
  test('factorial de 0 es 1', () => {
    expect(factorial(0)).toBe(1);
```

```
  });
  test('factorial de 5 es 120', () => {
    expect(factorial(5)).toBe(120);
  });
  test('factorial de 10 es 3628800', () => {
    expect(factorial(10)).toBe(3628800);
  });
  test('factorial de numero negativo lanza error', () => {
    expect(() => factorial(-1)).toThrow();
  });
});

describe('Funcion fibonacci()', () => {
  test('fibonacci(0) es 0', () => {
    expect(fibonacci(0)).toBe(0);
  });
  test('fibonacci(10) es 55', () => {
    expect(fibonacci(10)).toBe(55);
  });
  test('fibonacci de numero negativo lanza error', () => {
    expect(() => fibonacci(-1)).toThrow();
  });
});
```

Listing 9: Pruebas unitarias para math.js

6.2. Actividad 2: Provocar un error intencional y corregirlo

6.2.1. Error intencional

Se modificó una prueba para que fallara intencionalmente:

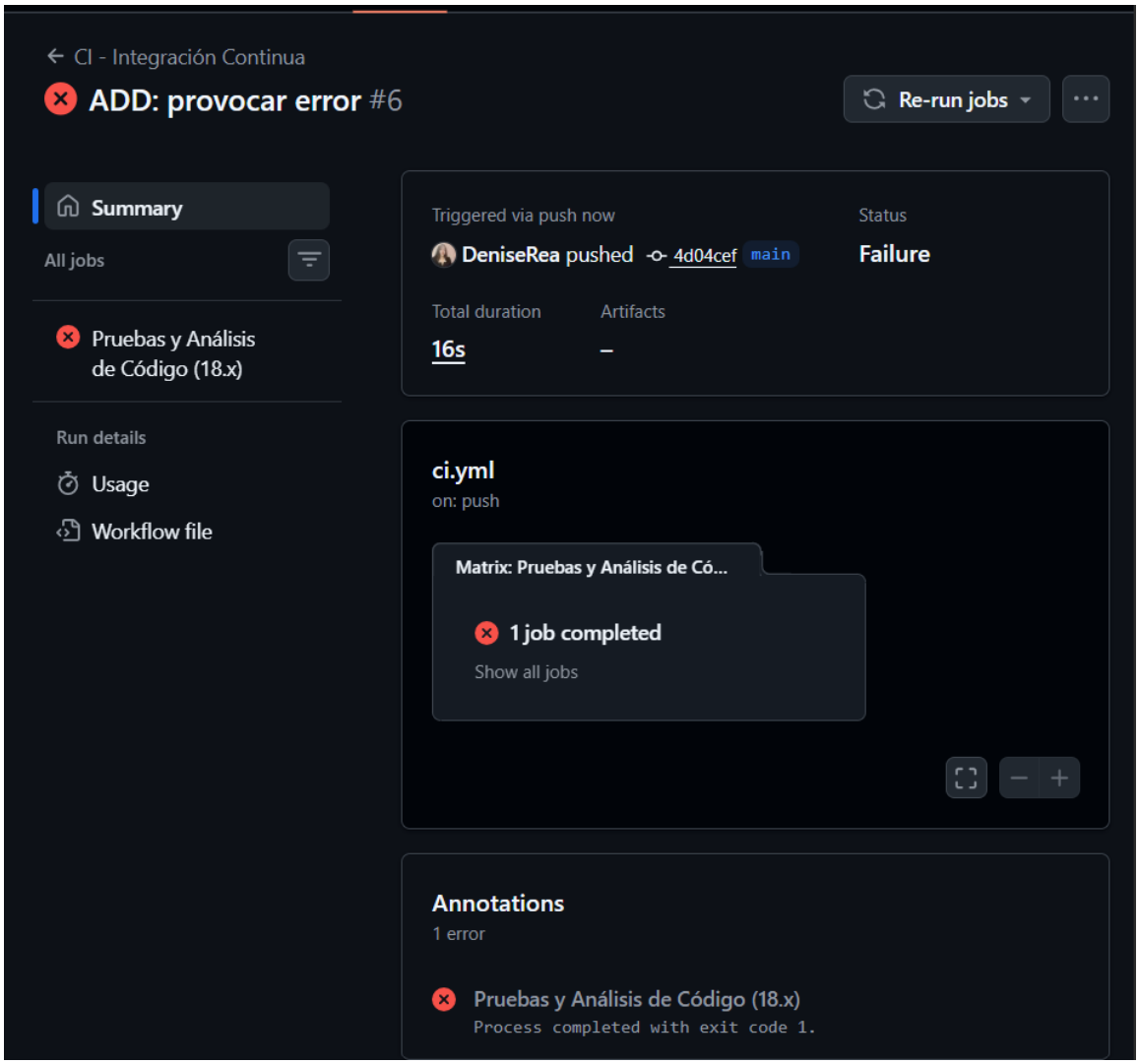

Figura 12: Resumen del workflow fallido mostrando el commit .ADD: provocar errorçon estado Failure

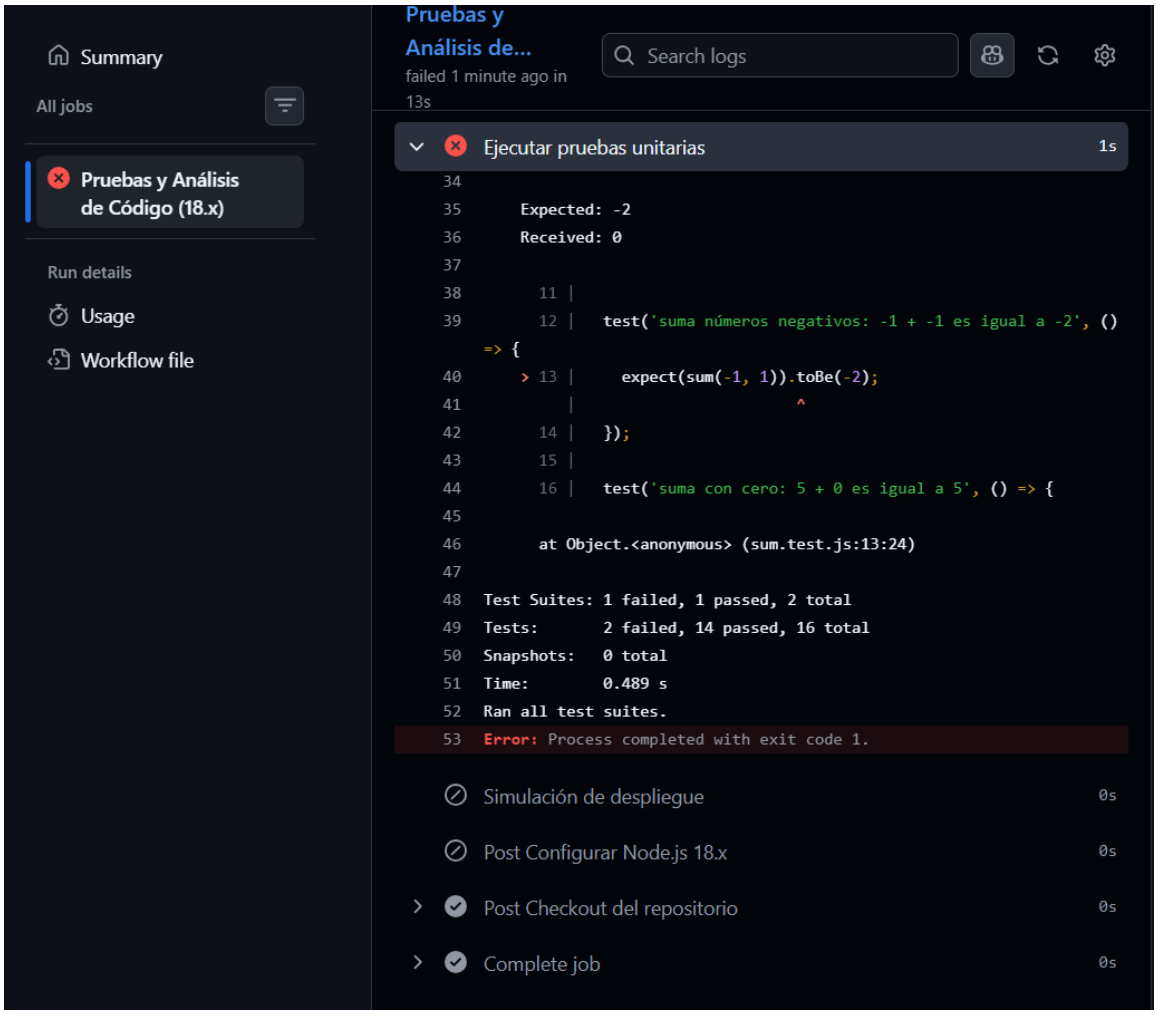

Figura 13: Log detallado mostrando el fallo en las pruebas unitarias: Expected -2, Received 0

6.2.2. Corrección del error

Se corrigió el error y se volvió a subir el código. El workflow se ejecutó exitosamente:

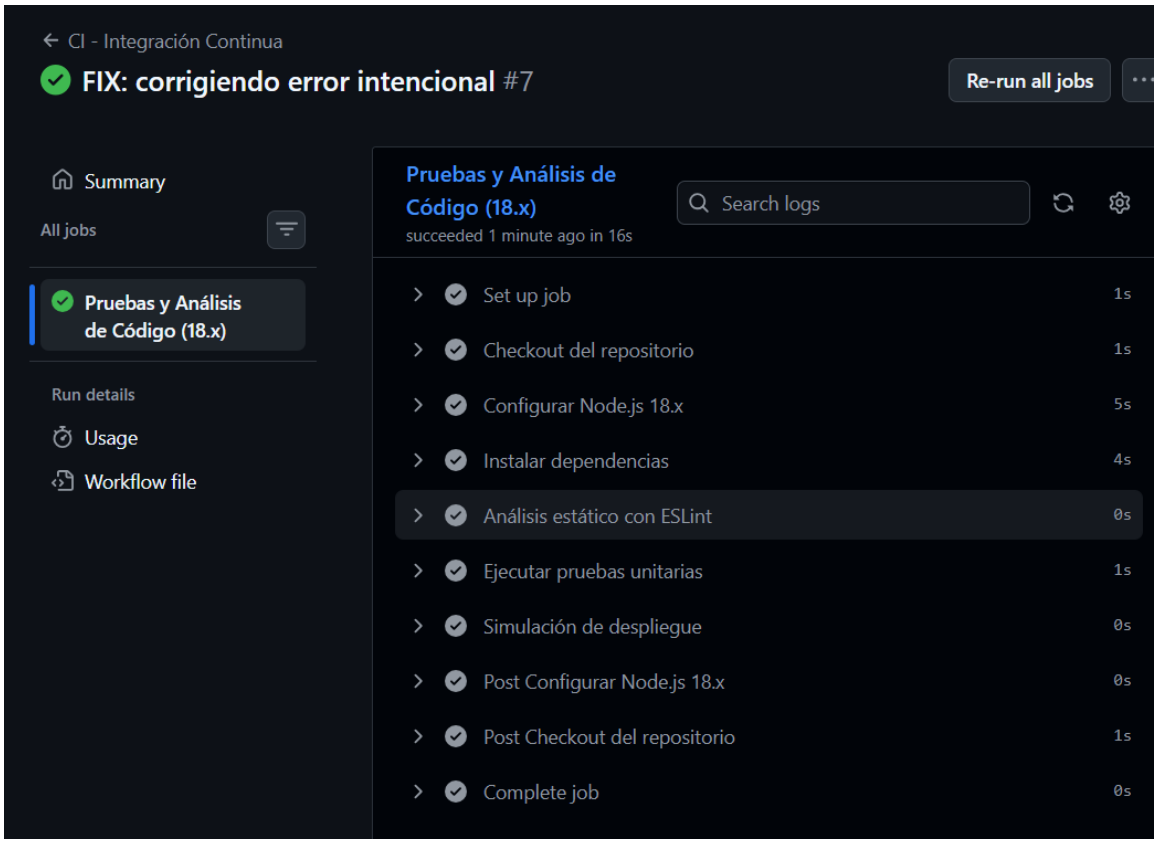

Figura 14: Workflow exitoso después de la corrección: commit "FIX: corrigiendo error intencionalçon todos los pasos completados

6.3. Errores Encontrados Durante el Laboratorio

Durante el desarrollo del laboratorio se presentó un error de configuración que impidió la ejecución exitosa del workflow en GitHub Actions.

6.3.1. Problema: Configuración no multiplataforma

El error principal fue una mala configuración de los archivos del workflow, específicamente en la regla `linebreak-style` de ESLint. El archivo `.eslintrc.json` originalmente tenía la siguiente configuración:

```
"linebreak-style": ["error", "unix"]
```

Listing 10: Configuración original con error de linebreak-style

Esta configuración causaba que el workflow fallara porque:

- En Windows, los saltos de línea usan CRLF (`\r\n`).
- En Linux (runner de GitHub Actions), los saltos de línea usan LF (`\n`).
- La regla `ünix"` exigía solo LF, generando errores en desarrollo local.

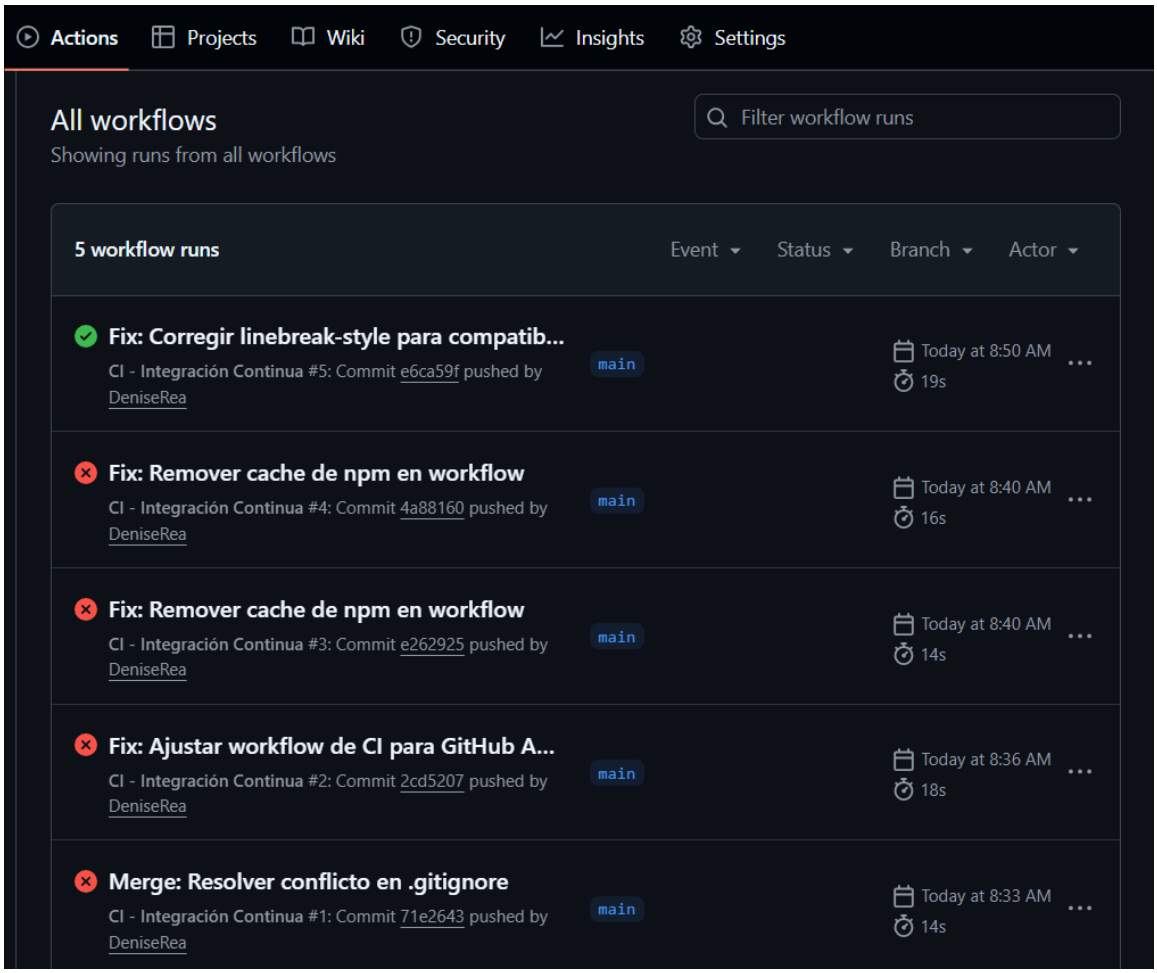

Figura 15: Historial de workflows mostrando los intentos de corrección: Fix linebreak-style, Fix remover cache de npm, y Fix ajustar workflow

6.3.2. Solución aplicada

Se modificó la configuración de ESLint para desactivar la verificación de estilo de salto de línea:

```
"linebreak-style": "off"
```

Listing 11: Configuración corregida para compatibilidad multiplataforma

Esta solución permite que el código funcione tanto en Windows como en Linux sin errores de linting, garantizando la compatibilidad multiplataforma del proyecto.

7. Resultados Obtenidos

7.1. Resumen de Pruebas Ejecutadas

Archivo de Pruebas	Tests	Pasaron	Estado
sum.test.js	4	4	PASS
math.test.js	12	12	PASS
Total	**16**	**16**	100 %

Cuadro 2: Resumen de pruebas unitarias ejecutadas

7.2. Funcionalidades del Pipeline CI

Pipeline de CI Implementado

- **Trigger automático:** Se activa con cada push y pull request a main.
- **Análisis estático:** ESLint verifica el código antes de las pruebas.
- **Pruebas unitarias:** Jest ejecuta todas las pruebas del proyecto.
- **Simulación de despliegue:** Confirmación visual del éxito del pipeline.
- **Notificaciones:** GitHub muestra el estado del workflow en cada commit.

7.3. Verificación del Error Intencional

Se comprobó que el sistema de CI detecta correctamente los errores:

1. Se modificó el test `sum(1, 2)` para esperar un valor incorrecto.
2. El workflow falló y marcó el commit con una X roja.
3. Al corregir el error, el workflow pasó exitosamente.
4. El commit se marcó con un check verde.

8. Análisis de Resultados

8.1. Beneficios de la Integración Continua

La implementación de CI/CD con GitHub Actions demostró varios beneficios:

1. **Detección temprana de errores:** Los errores se identifican inmediatamente después del push, antes de que afecten a otros desarrolladores o lleguen a producción.
2. **Automatización del proceso:** No es necesario ejecutar manualmente las pruebas ni el linter; todo se ejecuta automáticamente.
3. **Documentación del proceso:** Los logs del workflow sirven como registro de lo que ocurrió en cada ejecución.

4. **Confianza en el código:** Cada commit que pasa el pipeline tiene garantía de calidad básica.

8.2. Comparación de Escenarios

Escenario	Sin CI	Con CI
Detección de errores	Manual	Automática
Tiempo de feedback	Horas/Días	Minutos
Consistencia	Variable	Garantizada
Documentación	Manual	Automática
Esfuerzo repetitivo	Alto	Nulo

Cuadro 3: Comparación de desarrollo con y sin CI

9. Conclusiones

La práctica permitió implementar un pipeline de CI funcional con GitHub Actions que ejecuta automáticamente ESLint y Jest en cada push. Se comprobó que el sistema detecta errores correctamente, ya que al introducir un fallo intencional, el workflow lo identificó y marcó como fallido.

Un aprendizaje importante fue la necesidad de configurar el proyecto para compatibilidad multiplataforma, desactivando la regla `linebreak-style` en ESLint para evitar conflictos entre Windows (desarrollo local) y Linux (runner de GitHub Actions).

10. Recomendaciones

Se recomienda agregar reportes de cobertura con `npm test -coverage` e integrar servicios como Codecov. También es útil configurar protección de ramas en GitHub y usar caché para dependencias npm en el workflow para reducir tiempos de ejecución.

11. Referencias Bibliográficas

1. GitHub. (2024). *GitHub Actions Documentation*. Recuperado de: https://docs.github.com/en/actions
2. Jest. (2024). *Jest - Delightful JavaScript Testing*. Recuperado de: https://jestjs.io/docs/getting-started
3. ESLint. (2024). *ESLint - Pluggable JavaScript Linter*. Recuperado de: https://eslint.org/docs/latest/

4. Express.js. (2024). *Express - Node.js web application framework*. Recuperado de: https://expressjs.com/
5. Node.js. (2024). *Node.js Documentation*. Recuperado de: https://nodejs.org/docs/
6. Fowler, M. (2006). *Continuous Integration*. Recuperado de: https://martinfowler.com/articles/continuousIntegration.html

12. Anexos: Código Fuente Completo

12.1. Repositorio GitHub

El código fuente completo del proyecto está disponible en:
https://github.com/DeniseRea/Testing-CICD