



SISTEMAS OPERATIVOS

SECCIÓN: 219

NOMBRE:

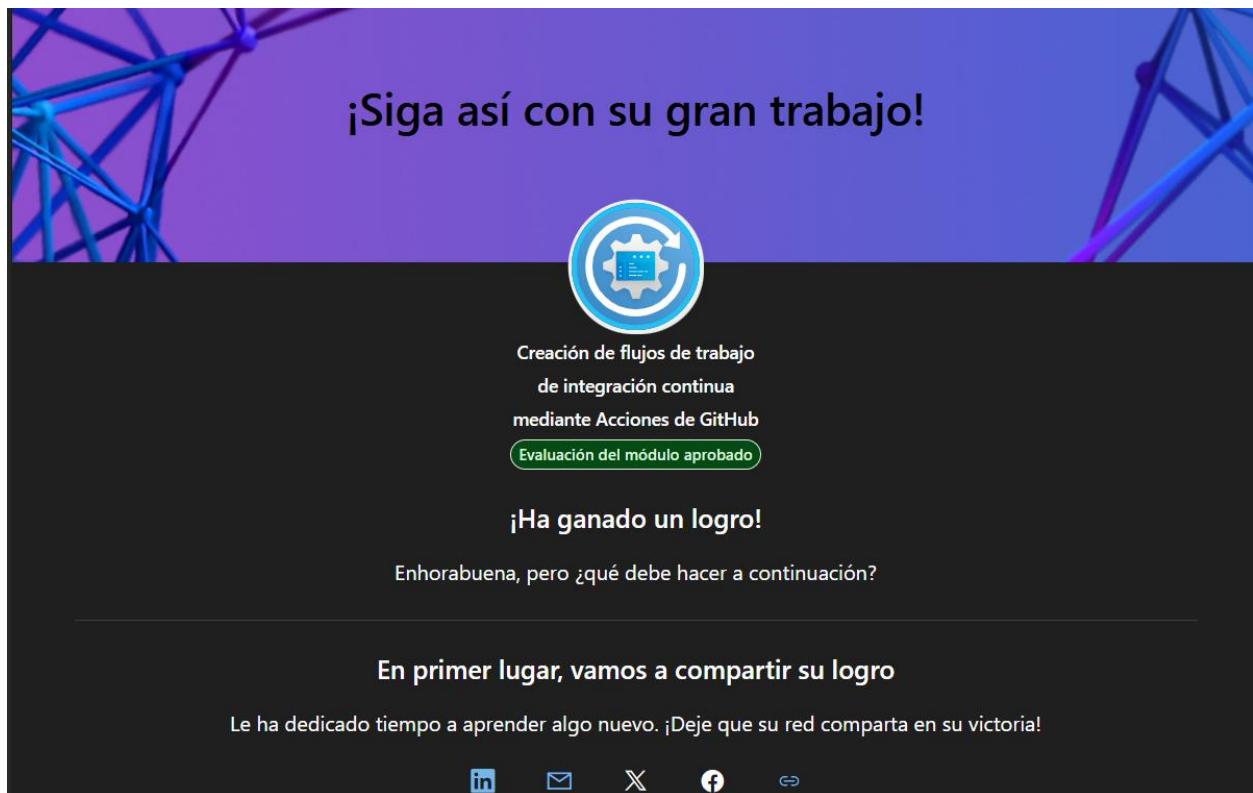
Abraham Josué Rivera Peralta

NO. CUENTA:

61911005

DOCENTE:

Ing. Kevin Iván Cruz



Workflows Básicos Multi-OS en GitHub Actions

Introducción

En este documento voy a explicar mi experiencia realizando el ejercicio de workflows multi-OS en GitHub Actions. Al principio me costó un poco entender cómo funcionaba todo el sistema, especialmente porque nunca había trabajado con GitHub Actions antes. Sin embargo, después de varios intentos y de comprender la estructura de los archivos YAML, logré completar el ejercicio exitosamente.

El objetivo principal del ejercicio era crear un workflow que se ejecutara simultáneamente en tres sistemas operativos diferentes: Ubuntu (Linux), Windows y macOS. Esto me permitió entender cómo GitHub Actions puede automatizar procesos en diferentes entornos, algo que resulta muy útil cuando se desarrolla software que debe funcionar en múltiples plataformas.

Proceso de Implementación

Creación del Repositorio y Estructura

Lo primero que hice fue crear un nuevo repositorio en GitHub. Esto fue relativamente sencillo, simplemente accedí a mi cuenta de GitHub y creé un repositorio público con un archivo README inicial.

Aprendí que GitHub Actions requiere una estructura específica de carpetas. El archivo del workflow debe estar en `.github/workflows/`. Al principio no entendía por qué debía empezar con un punto (.) en el nombre de la carpeta, pero después comprendí que en sistemas Unix/Linux esto indica que es una carpeta oculta. Crear esta estructura directamente en GitHub fue más fácil de lo que pensaba, ya que, al escribir el nombre del archivo con las barras diagonales, GitHub automáticamente creaba las carpetas.

Configuración del Workflow

La parte más complicada para mí fue entender la sintaxis YAML. Al principio no sabía que la indentación era tan importante en este tipo de archivos. Cometí algunos errores de formato que me causaron fallos en las primeras ejecuciones, pero eventualmente logré entender cómo estructurar correctamente el código.

El workflow que creé tiene tres secciones principales:

1. Triggers (Disparadores): Configuré el workflow para que se ejecute automáticamente cuando hago un push a la rama main o cuando se crea un pull request.
2. Estrategia de Matriz: Usando strategy: matrix, pude definir que el mismo trabajo se ejecutara en tres sistemas operativos diferentes. GitHub crea tres trabajos paralelos, uno para cada sistema operativo especificado.
3. Pasos del Workflow: Definí tres pasos principales: descargar el código del repositorio usando actions/checkout@v4, recolectar información del sistema operativo, y subir esa información como un artifact.

Diferencias Observadas Entre Sistemas Operativos

Una vez que el workflow se ejecutó exitosamente en los tres sistemas operativos, descargué los artifacts y analicé las diferencias. Esto fue realmente revelador porque pude ver de primera mano cómo cada sistema operativo maneja las cosas de manera diferente.

Ubuntu (Linux)

Ubuntu fue el sistema operativo donde todo funcionó más suavemente. El comando uname -a mostró información detallada del kernel de Linux, incluyendo la versión, la arquitectura del procesador (x86_64), y la fecha de compilación del kernel. Las variables de entorno en Ubuntu eran las más numerosas y me di cuenta de que muchas están relacionadas con las herramientas de desarrollo que GitHub proporciona, como variables para Node.js, Python, Java, y otros lenguajes de programación.

Lo que más me llamó la atención fue la cantidad de rutas (PATH) configuradas. Había muchísimos directorios listados, lo que me hizo entender que GitHub Actions pre-instala muchas herramientas en sus runners de Linux para que los desarrolladores puedan usarlas sin necesidad de instalarlas manualmente.

Windows

Windows fue el sistema operativo que más diferencias presentó. El primer problema que encontré es que Windows no tiene el comando uname, que es específico de sistemas Unix/Linux. Por eso tuve que usar el operador || en mi script para que, si el comando fallaba, simplemente escribiera "Windows System" en el archivo.

Las variables de entorno en Windows tenían una estructura diferente. Por ejemplo, usaban backslashes (\) en lugar de forward slashes (/) para las rutas de archivos. También noté que Windows usa diferentes nombres para sus variables: en lugar de HOME, usa USERPROFILE.

Otra diferencia importante fue que Windows mostró muchas más variables relacionadas con Microsoft, como las de Visual Studio, .NET Framework, y otras tecnologías específicas de Windows.

macOS

macOS fue un caso intermedio entre Ubuntu y Windows. Como macOS está basado en Unix, el comando uname -a sí funcionó correctamente. La información mostrada era similar a la de Ubuntu pero con diferencias claras: mostraba "Darwin" como el nombre del kernel (que es el kernel de macOS), y la arquitectura también era x86_64.

Las variables de entorno en macOS eran más limpias y organizadas en comparación con Ubuntu. Había menos variables preconfiguradas, pero las esenciales estaban presentes. También noté que macOS usa forward slashes (/) como Linux. Lo interesante fue ver variables específicas de Apple, como las relacionadas con Xcode y herramientas de desarrollo para iOS.

Cómo GitHub Actions Gestiona los Recursos

Durante este ejercicio aprendí bastante sobre cómo GitHub Actions gestiona los recursos para ejecutar workflows en diferentes sistemas operativos.

Máquinas Virtuales y Ejecución Paralela

GitHub Actions no ejecuta los trabajos directamente en servidores físicos, sino que usa máquinas virtuales. Cada vez que mi workflow se ejecutaba, GitHub creaba tres máquinas virtuales diferentes, una para cada sistema operativo. Esto significa que cada ejecución comienza con un ambiente completamente limpio y aislado.

Una de las cosas que más me gustó fue ver cómo los tres trabajos se ejecutaban en paralelo. Al usar la estrategia de matriz, GitHub Actions no esperaba a que terminara Ubuntu para empezar con Windows, sino que los tres comenzaban casi simultáneamente. Esto hace que el proceso sea mucho más rápido. A veces uno terminaba antes que otro, generalmente Ubuntu era el más rápido.

Artifacts y Limitaciones

El sistema de artifacts fue algo nuevo para mí. Comprendí que son una forma de guardar archivos generados durante la ejecución del workflow. En mi caso, cada sistema operativo generó su propio archivo system-info.txt, y GitHub los guardó como artifacts separados que se almacenan por 90 días por defecto.

También aprendí que GitHub Actions tiene limitaciones. Por ejemplo, hay límites de tiempo de ejecución (6 horas por trabajo) y límites de almacenamiento para artifacts. Para cuentas gratuitas, hay un límite mensual de minutos de ejecución, aunque los repositorios públicos tienen minutos ilimitados. Los runners de macOS consumen más minutos que los de Linux (un minuto de macOS cuenta como 10 minutos de tu cuota).

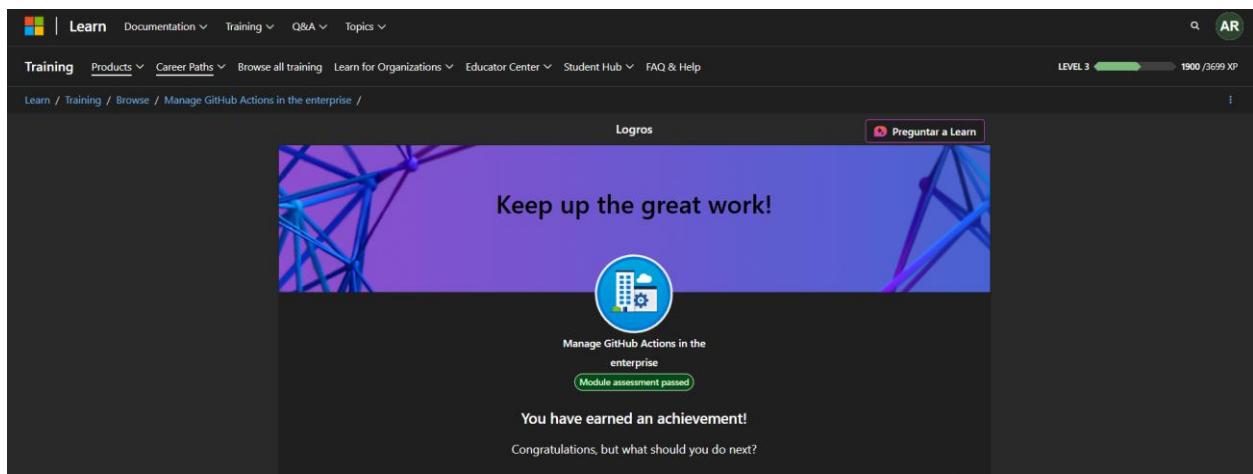
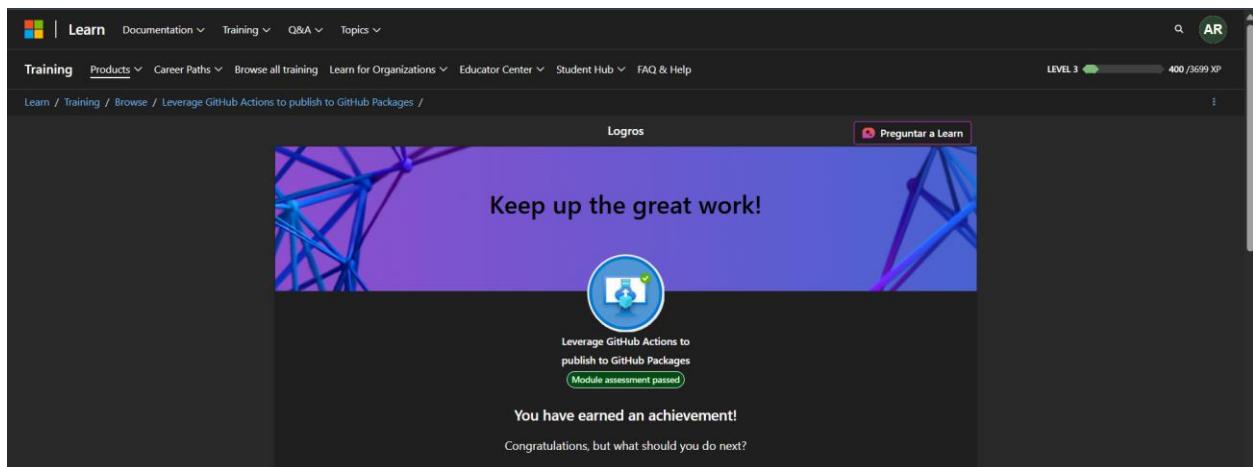
Conclusión

Este ejercicio me ayudó a entender cómo funcionan los sistemas de integración continua y cómo se puede automatizar la ejecución de código en diferentes entornos. Al principio me sentí abrumado por toda la terminología nueva (workflows, jobs, steps, runners, artifacts), pero al ir avanzando todo empezó a tener más sentido.

Lo que más valoro es haber podido ver de manera práctica las diferencias entre sistemas operativos. Ver las variables de entorno, los comandos disponibles, y la estructura de archivos me dio una perspectiva mucho más profunda. También aprendí la importancia de escribir código que sea compatible con múltiples plataformas.

Aunque el ejercicio fue desafiante al principio, especialmente por mi falta de experiencia con YAML y GitHub Actions, logré completarlo exitosamente y aprendí conceptos valiosos sobre automatización, sistemas operativos, y desarrollo de software multiplataforma.

CONTENEDORES Y GESTIÓN DE PROCESOS



Docker en GitHub Actions - Fase 2

Introducción

En esta fase trabajé con Docker para entender cómo funcionan los contenedores. La verdad es que al principio me costó bastante diferenciar entre contenedores y máquinas virtuales, pero después de hacer el ejercicio todo empezó a tener sentido.

El objetivo era crear un workflow que construyera una imagen Docker, corriera un contenedor, y monitoreara sus recursos. Suena complicado pero al final resultó ser más simple de lo que pensaba.

Lo que hice

El Dockerfile

Creé un Dockerfile básico para mi app de Node.js. Usé node:18-alpine como base porque es más ligera que las otras versiones. Básicamente el Dockerfile copia los archivos, instala las dependencias con npm, y ejecuta la aplicación.

Lo que me pareció interesante es que Docker construye la imagen por capas. Si no cambio las dependencias, las capas anteriores se reutilizan y el build es más rápido.

El Workflow

El workflow hace varias cosas:

1. **Construye la imagen:** Con docker build crea la imagen a partir del Dockerfile
2. **Corre el contenedor:** Lo ejecuta en segundo plano con -d y mapea el puerto 3000
3. **Prueba la comunicación:** Hace requests HTTP al contenedor desde el host
4. **Monitorea recursos:** Usa docker stats para ver cuánta CPU y memoria usa
5. **Genera un reporte:** Guarda toda la información en un archivo

Lo más complicado fue entender cómo comunicar el host con el contenedor, pero con -p 3000:3000 se mapea el puerto y funciona.

Monitoreo de Recursos

Creé un script que monitorea el contenedor durante 10 segundos. Me sorprendió ver que el contenedor solo usaba como 40 MB de RAM y casi nada de CPU. Eso me hizo entender por qué Docker es tan eficiente.

Contenedores vs Máquinas Virtuales

Esta fue la parte más importante que aprendí. Antes pensaba que eran lo mismo pero son bastante diferentes.

Máquinas Virtuales

Las VMs son como tener una computadora completa dentro de otra. Cada VM tiene:

- Su propio sistema operativo completo
- Su propio kernel
- Sus propios drivers y todo

Si tienes 3 VMs corriendo, tienes 3 sistemas operativos completos ejecutándose. Esto las hace:

- **Pesadas:** Cada VM ocupa varios GB
- **Lentas:** Tardan minutos en iniciar
- **Caras en recursos:** Cada una consume mucha RAM y CPU

Pero son muy seguras porque están completamente aisladas.

Contenedores Docker

Los contenedores son diferentes porque **comparten el kernel del sistema operativo**. Todos los contenedores en una máquina usan el mismo kernel pero tienen sus propios archivos y procesos aislados.

Esto los hace:

- **Ligeros:** Mi imagen fue de solo 150 MB
- **Rápidos:** Inician en 2-3 segundos
- **Eficientes:** Puedes correr muchos contenedores en una sola máquina

La desventaja es que tienen menos aislamiento que las VMs y solo puedes correr Linux en Linux (o Windows en Windows).

¿Cuándo usar cada uno?

Después del ejercicio entendí que:

- **Contenedores:** Para apps web, microservicios, CI/CD
- **VMs:** Cuando necesitas diferentes sistemas operativos o seguridad máxima

Para GitHub Actions, los contenedores son perfectos porque son rápidos y livianos.

Cómo GitHub Actions Maneja los Contenedores

GitHub Actions hace casi todo automáticamente, lo cual me facilitó mucho las cosas:

Preparación: Docker ya está instalado y listo en los runners de Ubuntu.

Construcción: Cuando hago docker build, GitHub Actions descarga las imágenes base y cachea las capas para builds más rápidos.

Ejecución: El contenedor corre aislado pero puede comunicarse con el host por los puertos.

Limpieza: Al final del workflow elimino el contenedor y la imagen manualmente, pero GitHub Actions de todas formas destruye todo el runner después. Cada ejecución empieza limpia.

Artifacts: Puedo guardar reportes como artifacts para revisarlos después.

Ventajas de Usar Docker en Actions

Encontré varias ventajas importantes:

- 1. Consistencia:** La app funciona igual en mi computadora, en GitHub Actions, y en producción. Se acabó el "en mi máquina sí funciona".
- 2. Rapidez:** Los contenedores inician en segundos. Esto hace que el CI/CD sea mucho más rápido.
- 3. Eficiencia:** Mi contenedor usaba solo 40 MB de RAM. Una VM usaría mínimo 512 MB solo para el OS.
- 4. Reproducibilidad:** El Dockerfile es código versionado. Puedo ver exactamente cómo se construyó cada versión.
- 5. Portabilidad:** La misma imagen funciona en diferentes plataformas.
- 6. Ecosistema:** Docker Hub tiene millones de imágenes listas. Usé node:18-alpine oficial en lugar de configurar todo manualmente.

Problemas que Tuve

No todo fue fácil:

Al principio no entendía el mapeo de puertos con -p

Tuve que hacer el script ejecutable con chmod +x

Los logs de Docker son muy verbosos, tuve que limitarlos

Sin limpieza las imágenes se acumulan

Pero nada que no se pudiera resolver investigando un poco.

Conclusión

Este ejercicio me ayudó a entender realmente cómo funcionan los contenedores. No son solo "VMs ligeras", son una tecnología completamente diferente que comparte el kernel para ser más eficiente.

Ver cómo GitHub Actions maneja todo automáticamente me mostró por qué Docker es el estándar para CI/CD. La combinación de rapidez, eficiencia y consistencia es perfecta para automatización.

Aunque tuve algunos problemas menores, el resultado final funciona bien y demuestra todos los conceptos importantes. Definitivamente voy a usar contenedores en proyectos futuros.

Scripts de Sistemas y Automatización

The screenshot shows a GitHub Actions run for a workflow named 'system-automation.yml' triggered via push 12 minutes ago. The run was successful with a total duration of 1m 53s and 4 artifacts produced. The artifacts table lists four items: 'comparative-analysis' (1.1 KB), 'linux-system-reports' (2.16 KB), 'macos-system-reports' (1.75 KB), and 'windows-system-reports' (1.64 KB). Each artifact has a download icon and a delete icon.

Name	Size	Digest	Actions
comparative-analysis	1.1 KB	sha256:c8010991791e6985bf8f449a9f18a349779713ba6d1...	Download Delete
linux-system-reports	2.16 KB	sha256:2961902f946625d083abe5ec84b069f6e5f15db2ac3...	Download Delete
macos-system-reports	1.75 KB	sha256:bf8638baa38d64846a0ecd56f065c07ea30818bba50...	Download Delete
windows-system-reports	1.64 KB	sha256:36f3b6c52716c30168d063e20a6ab2984e76a445950...	Download Delete

Crear Aplicación Base

```
DELL@DESKTOP-VG0EU42 MINGW64 ~/Documentos/so-github-actions-labs/app (main)
$ npm start

> app@1.0.0 start
> node server.js

[dotenv@17.2.3] injecting env (3) from .env -- tip: ⚙ override existing env vars with { override: true }
Servidor corriendo en http://localhost:3000

DELL@DESKTOP-VG0EU42 MINGW64 ~/Documentos/so-github-actions-labs/app (main)
$ npm test

> app@1.0.0 test
> jest

  console.log
    [dotenv@17.2.3] injecting env (2) from .env -- tip: ☑ audit secrets and track compliance: https://dotenvx.com/ops

      at _log (node_modules/dotenv/lib/main.js:142:11)

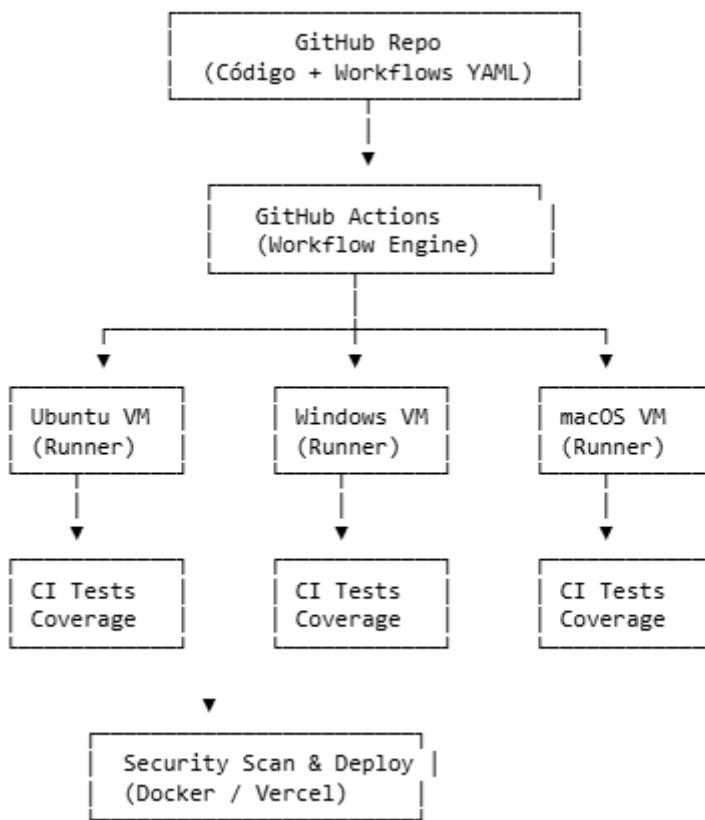
PASS  tests/server.test.js
Pruebas del servidor Express
  ✓ Debe responder en la ruta principal (64 ms)
  ✓ Debe devolver información del servidor (14 ms)
  ✓ Debe devolver la hora actual (69 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.919 s
Ran all test suites.
```

CI PIPELINE con Tests Automatizados

The screenshot shows the GitHub Actions CI pipeline for the repository 'so-github-actions-labs'. The workflow 'Add CI workflow for testing and building application #1' has been triggered via push 4 minutes ago by user 'AbrahamRP97' to branch 'main'. The status is 'Success' with a total duration of '1m 13s' and 1 artifact. The pipeline consists of several jobs: 'Test on ubuntu-latest', 'Test on windows-latest', 'Test on macos-latest', and 'Build Application'. All jobs have completed successfully. The 'ci.yml' configuration file shows a matrix test job.

The screenshot shows the GitHub README page for the repository 'so-github-actions-labs'. It features a 'README' section and a prominent 'CI - Continuous Integration' badge with the status 'passing'. This badge corresponds to the successful CI pipeline shown in the previous screenshot.



Cada runner de GitHub Actions es una máquina virtual temporal que se crea al iniciar un job y se destruye al finalizarlo.

Uso de recursos en el proyecto:

- * CPU: usada por Jest durante las pruebas.
- * RAM: utilizada por Node.js y dependencias.
- * Disco: temporalmente para node_modules y coverage/.
- * Red: para descargar dependencias desde npm y subir artifacts.

Sistema Operativo	Tiempo promedio de ejecución	Observaciones
Ubuntu	~45 segundos	Más rápido, mejor caching y compatibilidad con Node.js
Windows	~1 minuto 10 segundos	Más lento por overhead del sistema
macOS	~1 minuto 30 segundos	Menor disponibilidad, pero estable