

Pruebas Funcionales y No Funcionales

Objetivo

Comprender la diferencia entre las pruebas funcionales y no funcionales, su importancia en el ciclo de vida del software y cómo aplicarlas en un entorno real de desarrollo y calidad de software..

1. Introducción a los Tipos de Pruebas de Software

Las pruebas de software se dividen en varias categorías dependiendo de su enfoque y objetivo.

1.1. Clasificación General de las Pruebas

- **Pruebas Manuales:** Se ejecutan sin automatización, verificando el comportamiento del software manualmente.
 - **Pruebas Automatizadas:** Se utilizan scripts y herramientas para realizar pruebas repetitivas de manera eficiente.
 - **Pruebas Funcionales:** Evalúan si el software cumple con los requisitos especificados.
 - **Pruebas No Funcionales:** Analizan el rendimiento, seguridad y usabilidad.
 - **Pruebas de Caja Negra:** Evalúan la funcionalidad sin considerar el código interno.
 - **Pruebas de Caja Blanca:** Se enfocan en la estructura interna del código.
-

2. Detalle de los Tipos de Pruebas

Tipos de Pruebas de Software



3. Metodologías de Pruebas de Software

3.1. Pruebas en Diferentes Modelos de Desarrollo

Las pruebas pueden variar dependiendo del modelo de desarrollo utilizado:

▼ **3.1.1. Waterfall (Cascada):** Las pruebas ocurren en una fase separada después del desarrollo.

- Modelo secuencial donde las pruebas ocurren después del desarrollo.

- Poco flexible ante cambios.
- Se detectan errores tarde, lo que aumenta costos de corrección.

▼ **3.1.2. Agile:** Las pruebas son iterativas e integradas desde el inicio del desarrollo.

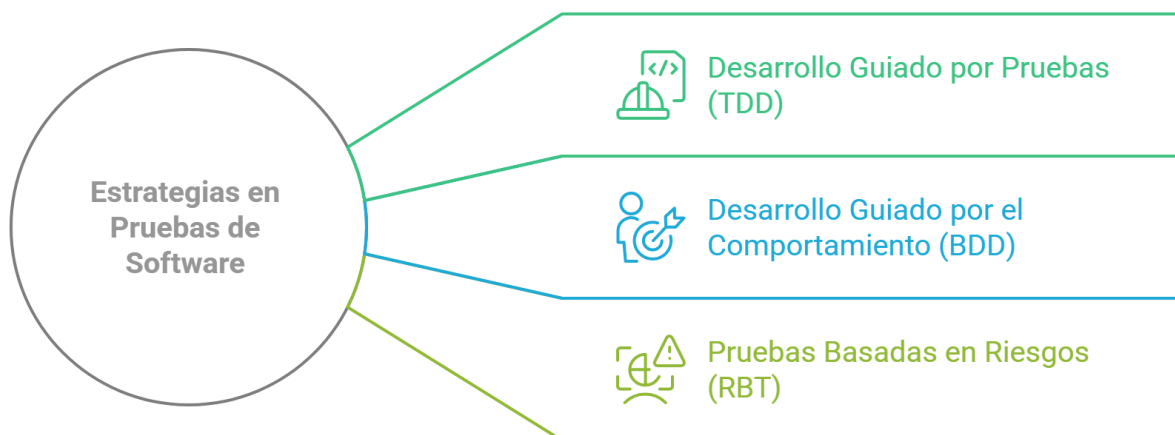
- Desarrollo iterativo con pruebas continuas.
- Se aplican pruebas en cada sprint.
- Mayor colaboración entre testers y desarrolladores.

▼ **3.1.3. DevOps:** Automatización y pruebas continuas para garantizar despliegues rápidos y sin errores.

- Integración y entrega continua (CI/CD).
- Automatización de pruebas en todas las etapas del desarrollo.
- Reducción del tiempo de entrega del software.

3.2. Estrategias de Pruebas

Explorando Estrategias en Pruebas de Software



▼ **3.2.1. Test-Driven Development (TDD) o Desarrollo Guiado por Pruebas** es una metodología de desarrollo de software donde **primero se escriben las pruebas y luego el código** para cumplirlas.

◆ **Ciclo de TDD:**

- 1 **Escribir una prueba** que falla (porque aún no hay código).
- 2 **Escribir el código mínimo** para que la prueba pase.
- 3 **Refactorizar** el código para mejorar su calidad sin cambiar su funcionalidad.
- 4 **Repetir el proceso** con nuevas pruebas.

📌 **Este proceso se conoce como el ciclo:**

Red (Falla) → Green (Pasa) → Refactor (Mejorar)

📌 **Beneficios de TDD**

- ✓ Código más robusto y libre de errores.
- ✓ Diseño más modular y mantenible.
- ✓ Fomenta el pensamiento en los requisitos antes del desarrollo.
- ✓ Facilita la detección de errores desde el principio.

▼ **3.2.2. Behavior-Driven Development (BDD) o Desarrollo Guiado por el Comportamiento** es una metodología de desarrollo de software que **extiende TDD**, enfocándose en el **comportamiento del sistema desde la perspectiva del usuario**

◆ **Principales características:**

- ✓ Usa un **lenguaje natural** para escribir pruebas (Gherkin).
- ✓ Facilita la comunicación entre **desarrolladores, testers y el negocio**.
- ✓ Se basa en **escenarios de uso real** escritos en formato **Given-When-Then**.


Diferencia con TDD:

- **TDD** se centra en pruebas unitarias (cómo funciona el código).
- **BDD** se centra en pruebas funcionales y de aceptación (cómo debería comportarse el sistema).

Beneficios de BDD

- ✓ Facilita la comunicación entre equipos técnicos y de negocio.
- ✓ Mejora la **claridad y documentación** de los requisitos.
- ✓ Se integra con herramientas como **Selenium, Jest, JUnit, SpecFlow**.

▼ **3.2.3. Risk-Based Testing (RBT) o Pruebas Basadas en Riesgos** es una estrategia de pruebas de software en la que **se priorizan las pruebas según el nivel de riesgo** que cada funcionalidad o módulo representa para el sistema.

 **El objetivo de RBT** es **optimizar los esfuerzos de prueba** centrándose en las áreas más críticas del software, aquellas que pueden causar **mayores fallos, pérdidas económicas o afectar la experiencia del usuario**.

¿Cómo se aplica RBT?

El proceso de **Pruebas Basadas en Riesgos** sigue estos pasos:

- 1 Identificar riesgos** → Analizar qué partes del sistema son más críticas.
- 2 Evaluar impacto y probabilidad** → Calificar cada riesgo en términos de gravedad y posibilidad de ocurrencia.
- 3 Priorizar pruebas** → Asignar más esfuerzo y tiempo a las pruebas de alto riesgo.
- 4 Diseñar y ejecutar pruebas** → Crear casos de prueba alineados con los riesgos identificados.
- 5 Monitorear y mitigar riesgos** → Ajustar las pruebas si cambian los riesgos en el desarrollo.

📌 Ejemplo de Aplicación de RBT

📌 Imaginemos una aplicación bancaria con estas funcionalidades:

- **Inicio de sesión** 🔴 (Alto riesgo, si falla, expone datos del usuario).
- **Transferencias** 🔴 (Alto riesgo, afecta transacciones económicas).
- **Consulta de saldo** 🟡 (Riesgo medio, afecta experiencia del usuario).
- **Personalización del perfil** 🟢 (Bajo riesgo, solo afecta estética).

📌 Evaluamos riesgo con una matriz (Impacto vs. Probabilidad):

Funcionalidad	Probabilidad	Impacto	Riesgo
Inicio de sesión	Alta	Alta	🔴 ALTO
Transferencias	Media	Alta	🔴 ALTO
Consulta de saldo	Media	Media	🟡 MEDIO
Perfil de usuario	Baja	Baja	🟢 BAJO

💡 Las pruebas funcionales se priorizarán para Inicio de Sesión y Transferencias.

💡 Se pueden reducir esfuerzos en pruebas de Personalización del Perfil.

📌 Beneficios de RBT

- ✅ **Optimiza el esfuerzo de pruebas** al centrarse en lo más crítico.
- ✅ **Reduce costos** al evitar pruebas innecesarias en áreas de bajo riesgo.
- ✅ **Aumenta la calidad del software** al minimizar fallos en partes sensibles.
- ✅ **Permite tomar decisiones estratégicas** en proyectos con tiempos ajustados.

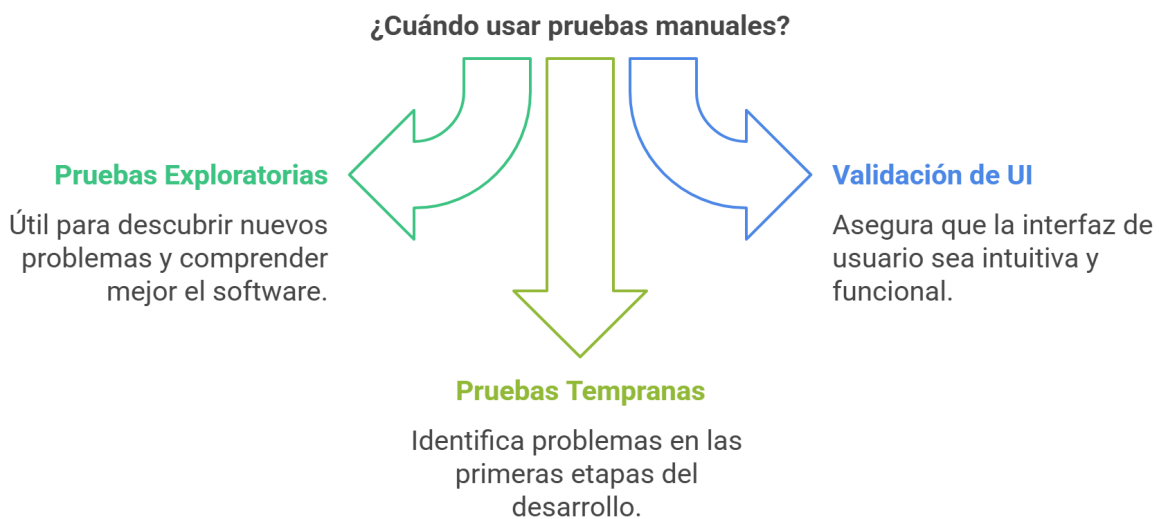
4. Pruebas Manuales vs. Automatizadas

4.1. Pruebas Manuales

- Se realizan de manera **manual** por un tester, sin el uso de herramientas automatizadas.
- El tester sigue un conjunto de **casos de prueba** y verifica que el software funcione como se espera.
- Son más **flexibles** y permiten detectar errores en aspectos subjetivos, como la **usabilidad** o la **experiencia de usuario**.
- Son útiles para pruebas exploratorias, de interfaz y de casos donde la automatización es poco viable.

Ejemplo de prueba manual:

Un tester ingresa credenciales en un formulario de inicio de sesión y verifica si accede correctamente.



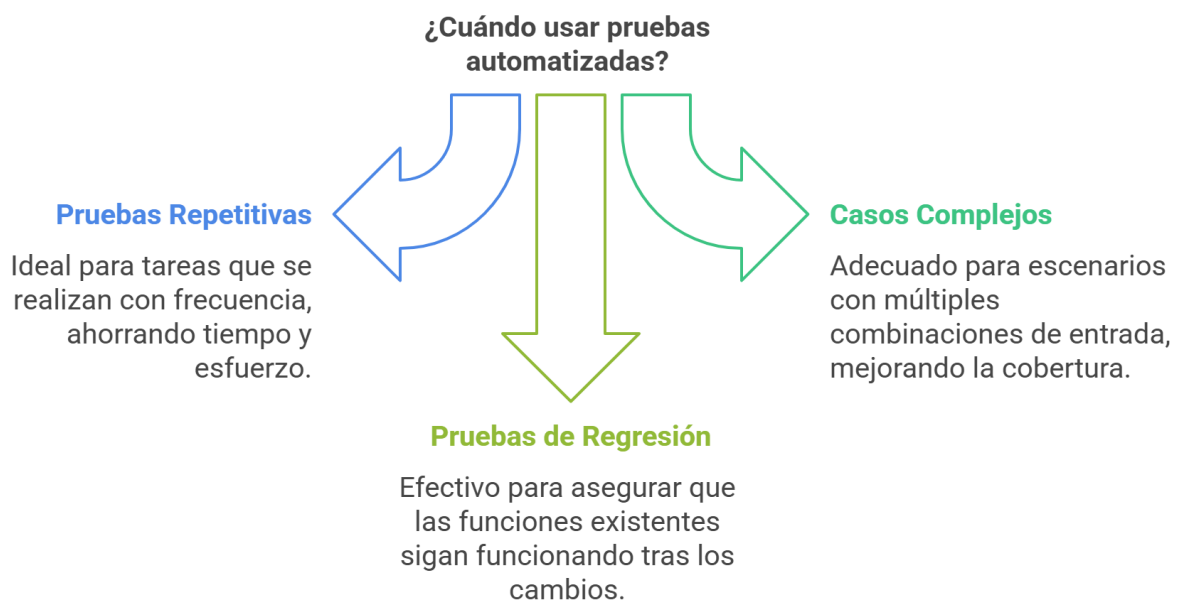
4.2. Pruebas Automatizadas

- Se realizan mediante **scripts y herramientas** que ejecutan los casos de prueba de forma automática.
- Son ideales para **pruebas repetitivas y regresión**, reduciendo el tiempo y el esfuerzo manual.

- Proporcionan **mayor cobertura** y son más eficientes en escenarios donde se deben probar muchas combinaciones.
- Requieren inversión inicial en **desarrollo y mantenimiento de scripts**.

Ejemplo de prueba automatizada:

Automatizar con Selenium la verificación del proceso de inicio de sesión en diferentes navegadores.



5. Pruebas Funcionales

Evalúan si el software cumple con los requisitos funcionales especificados. Se centran en la validación de la lógica del negocio.

5.1. Técnicas comunes:

5.1.1. Pruebas de Caja Negra

Concepto:

- Se enfocan en la **funcionalidad** del software sin analizar el código interno.

- Se basa en las **entradas y salidas esperadas** sin conocer cómo se procesan internamente.

Ejemplo:

Si tienes una calculadora y pruebas la suma de $2 + 3$, solo verificas que el resultado sea 5, sin importar cómo el sistema realizó la operación.

5.1.2. Pruebas Basadas en Casos de Uso

Concepto:

- Se basan en los **casos de uso** definidos en la documentación del sistema.
- Se evalúa el comportamiento del software desde la perspectiva del **usuario final**.
- Útil para validar **flujos de navegación y procesos completos**.

Ejemplo:

Para un **sistema de compras en línea**, un caso de uso podría ser:

1. El usuario agrega productos al carrito.
2. Procede al pago.
3. Ingresa datos de tarjeta.
4. Finaliza la compra y recibe confirmación. La prueba verificará que cada paso funcione correctamente.

5.1.3. Particiones de Equivalencia

Concepto:

- Se divide el conjunto de datos de entrada en **grupos (particiones)** que se comportan de la misma manera.
- Se prueba **solo un caso representativo por cada grupo**, reduciendo el número de pruebas necesarias.

Ejemplo:

Si un sistema solicita la edad de un usuario (entre 18 y 60 años), las particiones podrían ser:

- **Válida:** 25 (dentro del rango).
- **Inválida - Menor a 18:** 15.
- **Inválida - Mayor a 60:** 65. Solo se prueba un número por cada categoría en lugar de probar todos los posibles valores.

5.1.4. Análisis de Valores Límite

Concepto:

- Se prueban los valores **mínimos y máximos** en los límites de un rango aceptado, ya que es donde ocurren la mayoría de los errores.

Ejemplo:

Si un formulario acepta edades entre 18 y 60 años, las pruebas serían con valores:

- **Límite Inferior:** 17 (fuera del rango) y 18 (dentro del rango).
- **Límite Superior:** 60 (dentro del rango) y 61 (fuera del rango).

Esto asegura que el sistema maneje correctamente los bordes del rango permitido.

5.2. Tipos de Pruebas Funcionales:

1 Pruebas Unitarias

✓ **Objetivo:** Verificar que cada **módulo o función individual** del software funcione correctamente.

✓ Características:

- Se enfocan en componentes pequeños (métodos, clases, funciones).
- Son escritas y ejecutadas por los desarrolladores.
- Se utilizan frameworks como **Jest (JavaScript), JUnit (Java), PyTest (Python)**.

◆ **Ejemplo:** Prueba para una función que suma dos números.

```
function suma(a, b) {  
  return a + b;  
}
```

```
test('sumar 2 + 3 debe dar 5', () => {  
  expect(suma(2, 3)).toBe(5);  
});
```

2 Pruebas de Integración

✓ **Objetivo:** Validar que los **módulos individuales interactúan correctamente** cuando se combinan.

✓ **Características:**

- Verifican la comunicación entre diferentes partes del sistema.
- Pueden incluir interacciones con **bases de datos, APIs, servicios externos**.
- Se usan herramientas como **Postman, Jest con Supertest, JUnit con Spring Boot**.

◆ **Ejemplo:** Prueba de integración entre una API y una base de datos.

```
const request = require('supertest');  
const app = require('./app');
```

```
test('Debería obtener una lista de usuarios', async () => {  
  const res = await request(app).get('/api/users');  
  expect(res.status).toBe(200);  
  expect(res.body).toBeInstanceOf(Array);  
});
```

3 Pruebas de Sistema

✓ **Objetivo:** Evaluar el sistema completo **como un todo**, asegurando que cumpla con los requisitos funcionales.

✓ **Características:**

- Se ejecutan en un entorno realista.
- Incluyen pruebas de **UI, backend, base de datos y APIs**.
- Se usan herramientas como **Selenium, Cypress, TestCafe**.

◆ **Ejemplo:** Prueba automatizada con Selenium para verificar que un usuario pueda iniciar sesión.

```
const { Builder, By } = require('selenium-webdriver');

async function pruebaLogin() {
  let driver = await new Builder().forBrowser('chrome').build();
  await driver.get('https://miapp.com/login');

  await driver.findElement(By.name('email')).sendKeys('usuario@example.com');
  await driver.findElement(By.name('password')).sendKeys('password123');
  await driver.findElement(By.id('login-button')).click();

  let panel = await driver.findElement(By.id('dashboard'));
  console.log('Prueba exitosa');

  await driver.quit();
}

pruebaLogin();
```

4 Pruebas de Aceptación

✓ **Objetivo:** Validar que el software cumple con los **requisitos del negocio y las necesidades del usuario**.

✓ Características:

- Son realizadas por clientes, usuarios finales o testers.
- Se enfocan en **escenarios reales de uso**.
- Se pueden definir en formato **BDD (Gherkin/Cucumber)**.

◆ **Ejemplo:** Caso de prueba en **Gherkin** para validar el inicio de sesión.

Feature: Inicio de sesión

Scenario: Usuario ingresa con credenciales válidas

Given el usuario está en la página de inicio de sesión

When ingresa su correo "usuario@example.com"

And su contraseña "password123"

And hace clic en "Iniciar sesión"

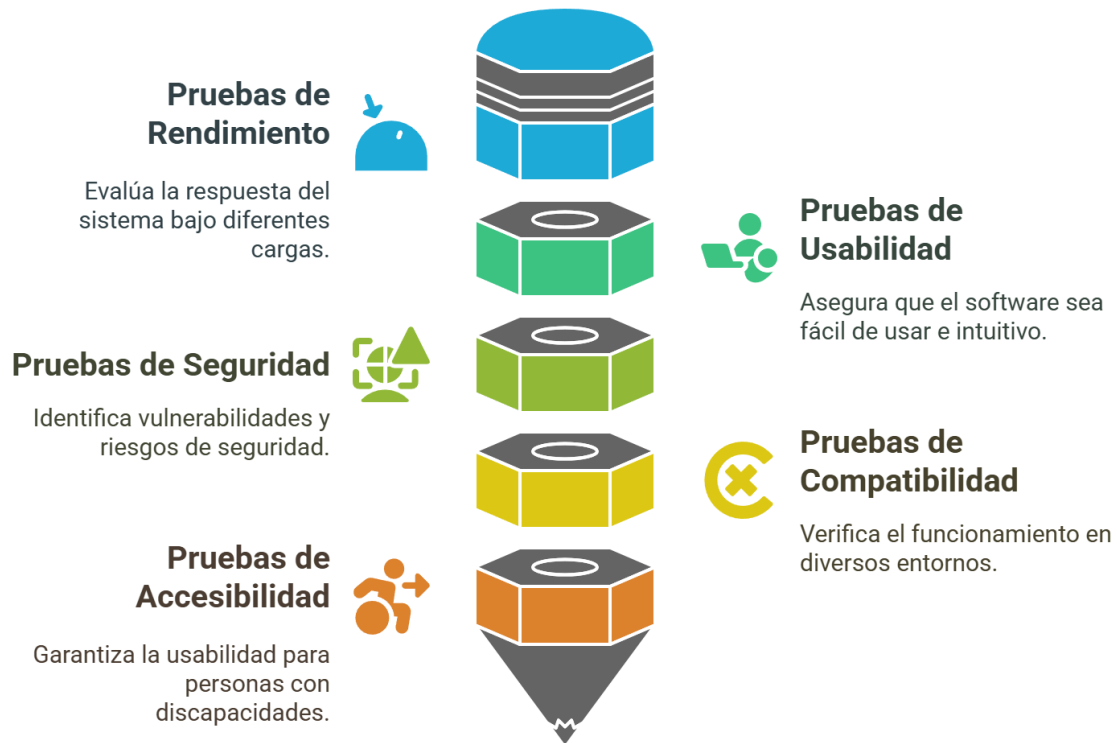
Then debería ver su panel de usuario

6. Pruebas No Funcionales

Evalúan atributos del software como rendimiento, usabilidad y seguridad. Se centran en la calidad y eficiencia del sistema.

6.1. Tipos de pruebas no funcionales:

Explorando Pruebas de Software No Funcionales



6.1.1. Pruebas de Rendimiento

Las pruebas de rendimiento se utilizan para evaluar cómo responde un sistema bajo diferentes condiciones de carga y uso. Se dividen en:

▼ ◆ Pruebas de Carga

- **Objetivo:** Medir cómo se comporta el sistema bajo una carga específica de usuarios concurrentes.
- **Ejemplo:** Simular 500 usuarios accediendo simultáneamente a una página web.
- **Herramientas:** JMeter, Gatling, LoadRunner.

▼ ◆ Pruebas de Estrés

- **Objetivo:** Evaluar la estabilidad del sistema al sobrecargarlo más allá de sus límites esperados.
- **Ejemplo:** Aumentar gradualmente el tráfico de usuarios hasta que el sistema falle.
- **Herramientas:** JMeter, K6, Locust.

▼ ◆ Pruebas de Volumen

- **Objetivo:** Analizar el rendimiento del sistema cuando maneja grandes volúmenes de datos.
- **Ejemplo:** Insertar 1 millón de registros en una base de datos y medir el tiempo de respuesta.
- **Herramientas:** JMeter, DBMonster.

6.1.2. Pruebas de Usabilidad

Evalúan la facilidad de uso y experiencia del usuario en la aplicación.

◆ **Objetivo:** Garantizar que el sistema sea intuitivo y fácil de usar.



Ejemplo: Realizar pruebas con usuarios para identificar dificultades en la navegación.



Herramientas: UsabilityHub, Hotjar, Google Lighthouse.

6.1.3. Pruebas de Seguridad

Identifican vulnerabilidades y riesgos de seguridad en el software.

◆ **Objetivo:** Proteger la aplicación contra ataques y accesos no autorizados.



Ejemplo: Simular un ataque de inyección SQL para verificar la seguridad de la base de datos.



Herramientas: OWASP ZAP, Burp Suite, Nessus.

6.1.4. Pruebas de Compatibilidad

Verifican que la aplicación funcione correctamente en diferentes entornos.

◆ **Objetivo:** Garantizar que el software sea compatible con distintos navegadores, dispositivos y sistemas operativos.



Ejemplo: Probar una web en Chrome, Firefox y Safari para detectar inconsistencias.



Herramientas: BrowserStack, CrossBrowserTesting, Sauce Labs.

6.1.5. Pruebas de Accesibilidad

Aseguran que la aplicación sea usable por personas con discapacidades.

◆ **Objetivo:** Cumplir con estándares como WCAG (Web Content Accessibility Guidelines) Las Web Content Accessibility Guidelines son una parte de las directrices de accesibilidad web publicadas por la Web Accessibility Initiative parte del World Wide Web Consortium, la principal organización de estándares de internet..



Ejemplo: Validar que un lector de pantalla pueda navegar correctamente por una página web.



Herramientas: Axe, WAVE, Google Lighthouse.

7. Material de Apoyo

- Libro: "Software Testing: A Craftsman's Approach" - Paul C. Jorgensen.
- Curso: "Introduction to Software Testing" (Coursera, Udacity).
- Herramientas sugeridas: Selenium, Cypress, JIRA, TestRail para gestión de pruebas.