UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO
DE INFORMÁTICA

Estrategia de pruebas automatizadas y análisis de calidad para frontend React: Caso Auditforge

Sebastián Alvarado

sebastian.alvarador@usm.cl

Profesor Guía: Mauricio Figueroa

Resumen

AuditForge es un software open-source para automatizar reportes de ciberseguridad. En la práctica, el alto esfuerzo de elaboración de informes y la ausencia de un enfoque sólido de pruebas automatizadas limitan la confiabilidad y mantenibilidad del sistema. Este trabajo diseña e implementa una suite de pruebas automatizadas para el frontend y configura un pipeline de Integración Continua en GitHub Actions, complementado con análisis de calidad mediante SonarQube y umbrales de aceptación.

La propuesta prioriza funcionalidades críticas usando *Risk-Based Testing* y emplea Jest y React Testing Library con datos deterministas y *mocks* centralizados. La validación incluye pruebas unitarias, de componentes e integración de rutas; los reportes de cobertura se generan en formato LCOV para su integración con Sonar.

Como resultado, se obtuvieron 997 pruebas con coberturas de 84.43 % en *statements* y 84.17 % en líneas, 68.02 % en ramas y 76.66 % en funciones, además de ejecuciones estables (100 % éxito) y tiempos sub-minuto en CI. Estos resultados habilitan refactorizaciones seguras, reducen el riesgo de regresiones y elevan la adecuación funcional, fiabilidad y mantenibilidad del producto, dejando una base reproducible para la evolución del proyecto y futuros colaboradores.

Palabras clave: cybersecurity, software quality, automated testing, CI/CD, code coverage

1 Introducción

1.1 Contexto y motivación

El contexto de esta investigación se da en el marco del proyecto AuditForge, realizado para la 32.ª Feria de Software realizada el año 2024, organizado por el Departamento de Informática de la Universidad Técnica Federico Santa María. [1]

AuditForge es un software *open-source* diseñado para automatizar los reportes de ciberseguridad de las organizaciones y personas interesadas en el análisis de seguridad de sistemas informáticos, o bien, *hacking*. [2] Esta automatización se realiza mediante el uso de *templates* definidos para la generación customizada de los reportes, asimismo con la asistencia de algoritmos de Inteligencia Artificial para la clasificación de las vulnerabilidades halladas según estándares internacionales.

1.2 Definición del problema

El hacking ético es la práctica de identificar y explotar vulnerabilidades en sistemas informáticos de manera autorizada. A diferencia de los hackers maliciosos, los hackers éticos actúan de manera legal y ética. Bajo este contexto, hay empresas especializadas que realizan pruebas de seguridad mediante auditorías, con

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): X Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Estrategia de pruebas automatizadas y análisis de calidad para frontend React: Caso Auditforge

Nombre del candidato(a): Sebastián Alvarado

Carrera / Grado: Ingeniería Civil Informática

Campus: Casa Central **Departamento:** Departamento de Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Mauricio Figueroa, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

X El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por **(marcar una opción)**:

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 21/10/2025 **Firma:** ____________________

Estudiante o Candidato(a):

Fecha: 21/10/2025 **Firma:** ____________________

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

el fin de informar sobre debilidades que comprometen los sistemas del cliente. Así, los auditores de seguridad son los encargados de realizar las pruebas de vulnerabilidad y culminan el proceso de identificación de vulnerabilidades con un reporte que resume todos sus hallazgos.

El tiempo utilizado en la generación de informes de los auditores de seguridad resulta en un problema, ya que puede ocupar hasta el 40 % del tiempo total dedicado al proyecto. [3] Esto hace que el proceso de hacking ético no sea óptimo, ya que resta tiempo útil a la realización de pruebas de seguridad adicionales en función del plazo establecido por el cliente. Este problema impacta negativamente al auditor y a la empresa de ciberseguridad, dado que reduce el tiempo que el auditor dedica a la búsqueda de vulnerabilidades y aumenta el tiempo destinado a la generación de reportes.

El núcleo del problema radica en la ausencia de un enfoque robusto de *QA* y *pruebas* automatizadas en AuditForge, lo cual limita su capacidad para generar informes realmente confiables y precisos. Aunque AuditForge permite a los auditores automatizar parte del proceso de generación de informes, su fiabilidad sigue dependiendo en gran medida de la exactitud y consistencia del software al procesar los datos introducidos por el auditor. En un ámbito tan crítico como el de la ciberseguridad, donde cualquier error en los informes puede incrementar los riesgos de seguridad y derivar en reportes incompletos o incorrectos, la implementación de un sistema de *QA* y *pruebas* automatizadas se vuelve fundamental. Este sistema permitiría garantizar que el software funcione correctamente en una variedad de entornos y condiciones operativas, lo cual es esencial para reducir al mínimo los riesgos de fallos y errores.

La implementación de *QA* y *pruebas* automatizadas en AuditForge no solo mejoraría la precisión y confiabilidad de los informes, sino que además establecería una base sólida para asegurar la escalabilidad del software y su capacidad de adaptarse a nuevos escenarios, mayores volúmenes de datos, y el desarrollo de nuevas funcionalidades a largo plazo. Este desafío implica un **problema de ingeniería** de alta complejidad, ya que requiere abordar diversos aspectos como la validación de la funcionalidad del software, su capacidad de recuperación ante fallos, y la integridad de la información procesada en diversas condiciones de uso, entre otros. Solucionar este problema tendría un impacto significativo en la calidad de las auditorías y en la eficiencia operativa, consolidando a AuditForge como una herramienta confiable y eficiente en el ámbito de ciberseguridad.

1.3 Objetivos

El **objetivo general** de este trabajo es diseñar e implementar una suite de pruebas automatizadas para el frontend de AuditForge que sirva como referencia y herramienta de mantenimiento para futuros desarrolladores del proyecto open-source, facilitando la detección temprana de errores y la adopción de buenas prácticas de pruebas dentro del proyecto.

Los **objetivos específicos** son:

- **Diseñar una estrategia de pruebas (unitarias y de integración)** que identifique y priorice las funcionalidades críticas del frontend de AuditForge.
- **Implementar la suite de pruebas unitarias e integración** de acuerdo al diseño, cubriendo los componentes y flujos priorizados, con el objetivo de alcanzar al menos un 80 % de cobertura de código.
- **Configurar un pipeline de integración continua en GitHub Actions** que ejecute los tests y análisis de calidad a través de SonarQube, con reportes automáticos de fallos.

- **Documentar** la estrategia de pruebas y el pipeline de integración continua en documentos dentro del repositorio.

2 Marco Teórico

Glosario

Open-source Software de código abierto, cuyo código fuente está disponible públicamente para que cualquiera lo vea, modifique y distribuya.

Hacking La actividad de identificar y explotar vulnerabilidades en un sistema informático o red. En el contexto del documento, se refiere al *hacking ético*.

Templates Plantillas predefinidas que sirven como base para crear documentos, en este caso, reportes de ciberseguridad.

QA (Quality Assurance) Aseguramiento de la Calidad. Proceso sistemático para garantizar que un producto o servicio cumple con los requisitos de calidad especificados.

Pentesting (Penetration Testing) Pruebas de penetración. Un ataque simulado autorizado en un sistema informático para evaluar su seguridad.

Hardening Proceso de asegurar un sistema reduciendo su superficie de vulnerabilidad.

Pull Request Una solicitud para fusionar un conjunto de cambios de una rama a otra en un repositorio de Git.

Commit Una operación en Git que guarda los cambios en el repositorio de forma permanente.

End-to-end (E2E) Pruebas de extremo a extremo. Un tipo de prueba que verifica el flujo completo de una aplicación desde el principio hasta el final.

Line coverage Cobertura de línea. Métrica que indica el porcentaje de líneas de código que han sido ejecutadas por una suite de pruebas.

Function coverage Cobertura de función. Métrica que indica el porcentaje de funciones o subrutinas que han sido llamadas durante la ejecución de las pruebas.

CI/CD (Continuous Integration/Continuous Delivery) Integración Continua y Entrega Continua. Prácticas para automatizar las fases del desarrollo de software, como la compilación, las pruebas y el despliegue.

FURPS Acrónimo para Functionality, Usability, Reliability, Performance, and Supportability, un modelo para clasificar los atributos de calidad del software.

Bottom-up Enfoque de abajo hacia arriba. Estrategia de procesamiento de información que se basa en el análisis de los componentes individuales para luego ensamblarlos en un sistema completo.

2.1 Ciberseguridad

La ciberseguridad comprende prácticas, procesos, tecnologías y políticas orientadas a proteger la confidencialidad, integridad y disponibilidad de la información y de los servicios digitales frente a amenazas intencionales (ataques) o accidentales (fallos, desastres). Incluye capas preventivas (hardening, segmentación), de detección (monitorización, correlación de eventos) y de respuesta (contención, erradicación, recuperación). Estos pilares se alinean con el modelo CIA (Confidencialidad, Integridad y Disponibilidad) y se complementan con trazabilidad y resiliencia operativa.

Figura 1: Triada CIA: Confidencialidad, Integridad y Disponibilidad. [4]

2.2 Hacking y Hacking Ético

El hacking, en su acepción técnica, es la exploración creativa y profunda de sistemas con el objetivo de comprender su funcionamiento interno o modificarlo. El *hacking ético* aplica ese conocimiento bajo autorización formal para identificar y reportar vulnerabilidades antes de que sean explotadas de forma maliciosa. Sus principios clave abarcan: alcance acordado, mínima interferencia operativa, documentación precisa de hallazgos, y divulgación responsable. El valor central reside en la reducción proactiva del riesgo y en la mejora continua de los controles de seguridad. [5]

2.3 Pruebas de Penetración (Pentesting)

El pentesting es una evaluación metodológica y acotada en el tiempo que simula técnicas de ataque reales sobre un objetivo definido, con el fin de validar la efectividad de controles y priorizar vulnerabilidades explotables.[6] Suele estructurarse en las siguientes fases:

1. **Planificación y definición de alcance**: acuerdo de objetivos, reglas de prueba, limitaciones y autorizaciones.
2. **Reconocimiento y enumeración**: recopilación de información pública y escaneo activo del entorno.
3. **Análisis y modelado de amenazas**: identificación de vectores de ataque y priorización de superficies críticas.
4. **Explotación controlada**: ejecución de técnicas para comprometer los activos previamente identificados.
5. **Post-explotación**: movimiento lateral, escalamiento de privilegios y persistencia controlada para evaluar el impacto.

6. **Limpieza**: restauración del entorno, eliminación de artefactos y validación de integridad.
7. **Reporte ejecutivo y técnico**: documentación de hallazgos, evidencia de impacto y clasificación de riesgos por probabilidad y severidad.

A diferencia del *hacking* ético genérico —que puede incluir actividades de *hardening*, asesoría o modelado de amenazas— el pentest enfatiza la evidencia de impacto explotable y la clasificación de riesgos según probabilidad y severidad.

2.4 Pruebas automatizadas en el ciclo de desarrollo

Las pruebas automatizadas son un pilar en los procesos modernos de desarrollo de software, ya que permiten verificar de manera continua el correcto funcionamiento de un sistema. A diferencia de las pruebas manuales, las automatizadas se integran directamente en el ciclo de vida del software, reduciendo errores humanos y aumentando la eficiencia, así contribuyendo a la mejora de la calidad técnica del software.

Entre los tipos de pruebas más relevantes se encuentran:

- **Pruebas unitarias:** validan funciones o módulos individuales.
- **Pruebas de integración:** verifican la interacción entre componentes.
- **Pruebas end-to-end:** evalúan el flujo completo del sistema desde la perspectiva del usuario.

2.5 Cobertura de código

La cobertura de código es una métrica que indica qué porcentaje del sistema ha sido ejecutado por las pruebas automatizadas. [7] Sus principales variantes incluyen:

- **Line coverage:** porcentaje de líneas de código ejecutadas.
- **Branch coverage:** porcentaje de ramas de decisión ejecutadas.
- **Function coverage:** porcentaje de funciones llamadas durante las pruebas.

Si bien una mayor cobertura no garantiza la ausencia de errores, sí refleja un mayor nivel de exhaustividad en la validación del sistema.

2.6 Integración y entrega continua (CI/CD)

La integración continua (CI) y la entrega continua (CD) son prácticas que permiten automatizar la construcción, pruebas y despliegue del software. CI busca detectar defectos tempranamente al integrar y validar cambios de código con frecuencia, así mejorando la calidad del software. [8]

GitHub Actions es una herramienta estándar para proyectos *open-source*, ya que permite definir pipelines que ejecutan pruebas, generan reportes y validan cambios en cada commit o pull request. En el contexto de AuditForge y esta memoria [9], **GitHub Actions** será utilizado para ejecutar automáticamente las pruebas y el análisis de calidad.

2.7 SonarQube como herramienta de análisis de calidad

SonarQube es una plataforma de análisis estático de código ampliamente utilizada para medir calidad y mantener estándares en proyectos de software. [10] Sus principales características incluyen:

- Medición de cobertura de código.
- Detección de duplicación y complejidad ciclomática.
- Identificación de vulnerabilidades de seguridad y hotspots.
- Integración con pipelines de CI/CD para generar reportes automáticos.

En este trabajo, **SonarQube** se usará principalmente para validar la cobertura de código alcanzada por la suite de pruebas, pero también aportará métricas adicionales de calidad que refuercen la mantenibilidad del proyecto.

2.8 Modelos de Calidad de Software: Evolución Histórica

Los modelos de calidad de software han evolucionado desde enfoques iniciales centrados en factores aislados hacia marcos normalizados que integran atributos técnicos y de experiencia de uso. A continuación se presenta una síntesis cronológica que contextualiza la base conceptual utilizada en este trabajo.

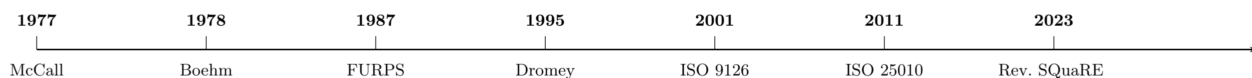

Figura 2: Línea de tiempo de la evolución de modelos de calidad relevantes. Elaboración propia.

2.8.1 Modelo de McCall (1977)

Plantea la calidad como un constructo multidimensional. [11] Se estructura en factores y criterios medibles.

- **Perspectivas del producto:**
  - *Operación*: comportamiento en ejecución (corrección, fiabilidad, eficiencia, integridad, usabilidad).
  - *Revisión*: facilidad de cambio (mantenibilidad, flexibilidad, capacidad de prueba).
  - *Transición*: adaptación a nuevos entornos (portabilidad, interoperabilidad, reusabilidad).
- **Aportes clave:**
  - Taxonomía de 11 factores vinculables a métricas (densidad de defectos, MTBF, etc.).
  - Enlace conceptual entre atributos internos y calidad percibida externa.
  - Base para modelos jerárquicos posteriores (Boehm) e iniciativas de estandarización.

2.8.2 Modelo de Boehm (1978)

Propone una jerarquía donde la *utilidad* sintetiza atributos técnicos y operacionales. [12]

- **Estructura jerárquica:**

- Nivel alto: utilidad / valor para el usuario.
- Nivel intermedio: portabilidad, mantenibilidad, eficiencia, fiabilidad, usabilidad.
- Nivel base: propiedades primitivas (auto-descriptividad, modularidad, simplicidad, consistencia).

- **Contribuciones:**
  - Priorización: algunos atributos habilitan a otros (ej. modularidad → mantenibilidad → portabilidad).
  - Formalización de métricas tempranas para estimar esfuerzo de prueba y mantenimiento.
  - Soporte a decisiones de ingeniería mediante descomposición top–down.

2.8.3 Modelo FURPS (1987)

Enmarca requisitos funcionales y no funcionales en cinco categorías que facilitan su trazabilidad y priorización. [13]

- **Categorías:**
  - *F* (Functionality): completitud, seguridad, reglas de negocio.
  - *U* (Usability): ergonomía, consistencia, accesibilidad, estética.
  - *R* (Reliability): disponibilidad, tolerancia a fallos, MTTR, tasa de error.
  - *P* (Performance): latencia, throughput, uso de recursos, escalabilidad.
  - *S* (Supportability): mantenibilidad, configurabilidad, extensibilidad, diagnósticos.
- **Contribuciones:**
  - Marco simple para incorporar calidad en backlogs iterativos.
  - Alineación entre requisitos y métricas operacionales.
  - Facilita la negociación de alcance no funcional.

2.8.4 Modelo de Dromey (1995)

Propone un mapeo sistemático entre propiedades del producto y atributos de calidad resultantes mediante una construcción bottom–up. [14]

- **Principio central:** la calidad deriva de la corrección y consistencia de los componentes construidos.
- **Clases de propiedades internas:** estructura, comportamiento, diseño, implementación.
- **Mecanismo:** cada propiedad interna se asocia a atributos externos (fiabilidad, mantenibilidad) mediante reglas de correspondencia.
- **Contribuciones:**
  - Explicita causalidad técnica (no solo taxonomía descriptiva).
  - Fomenta validación temprana de artefactos intermedios (no esperar a pruebas finales).
  - Complementa enfoques top–down (FURPS) al ofrecer trazabilidad ascendente.

2.8.5 Modelo ISO 9126 (2001)

Estandariza seis grandes características y sus subcaracterísticas asociadas. [15]

- **Contribuciones:**
  - Unificación de modelos previos (McCall, Boehm) en una taxonomía normalizada.
  - Introducción de subcaracterísticas medibles para trazabilidad.
  - Base conceptual para la evolución hacia ISO/IEC 25010.
- **Características y ejemplos de subcaracterísticas:**
  - Funcionalidad: adecuación, exactitud, interoperabilidad, seguridad, cumplimiento.
  - Fiabilidad: madurez, tolerancia a fallos, recuperabilidad.
  - Usabilidad: entendibilidad, aprendibilidad, operatividad, atractividad.
  - Eficiencia: comportamiento temporal, uso de recursos.
  - Mantenibilidad: analizabilidad, modificabilidad, estabilidad, capacidad de prueba.
  - Portabilidad: adaptabilidad, instalabilidad, reemplazabilidad.

Modelo	Enfoque	Estructura	Objetivo Principal
McCall (1977)	Factores y criterios	3 perspectivas (Operación, Revisión, Transición)	Vincular atributos a métricas de evaluación.
Boehm (1978)	Jerarquía top-down	Utilidad → Atributos → Propiedades primitivas	Priorización y descomposición de calidad.
FURPS (1987)	Clasificación requisitos	Functionality, Usability, Reliability, Performance, Supportability	Trazabilidad de requisitos no funcionales.
Dromey (1995)	Mapeo bottom-up	Propiedades internas → Atributos externos	Explicar causalidad técnica de la calidad.
ISO 9126 (2001)	Normalización	6 características + subcaracterísticas	Taxonomía estándar unificada.
ISO/IEC 25010 (2011)	Dual (Producto / En Uso)	8 características producto; 5 en uso	Cobertura integral y medición formal.
SQuaRE Rev. (2023)	Extensión	Refina interacción, seguridad y datos	Actualizar énfasis en resiliencia y experiencia.

Cuadro 1: Comparación de modelos de calidad de software. Elaboración propia.

2.8.6 Familia ISO/IEC 25000 e ISO/IEC 25010

La familia SQuaRE amplía el alcance a calidad de datos, medición y evaluación. [16]

- **Partes relevantes:**
  - 25010: modelos de calidad de producto y en uso.
  - 25012: calidad de datos.
  - 2502x: medición y métricas.
  - 2504x: procesos de evaluación.
- **Perspectivas ISO/IEC 25010:**
  - *Calidad en Uso*: eficacia, eficiencia, satisfacción, libertad de riesgo (seguridad de uso), cobertura de contexto.
  - *Calidad de Producto*: adecuación funcional, rendimiento, compatibilidad, usabilidad/interacción, fiabilidad, seguridad, mantenibilidad, portabilidad/flexibilidad, protección.
- **Contribuciones adicionales (rev. 2023):**
  - Refinamiento de atributos de interacción (inclusividad, asistencia, auto-descriptividad).
  - Mayor énfasis en resiliencia y resistencia ante amenazas.
  - Clarificación de métricas operacionales vinculadas a CI/CD (ej. cobertura para capacidad de ser probado).

Característica (ISO/IEC 25010)	Descripción
Adecuación Funcional	Completitud y corrección de las funciones frente a necesidades explícitas e implícitas.
Eficiencia de desempeño	Uso eficiente de recursos y tiempos de respuesta dentro de límites aceptables.
Compatibilidad	Coexistencia e interoperabilidad con otros sistemas y entornos.
Capacidad de interacción	Capacidad de ser entendido, aprendido, operado y atractivo para el usuario.
Fiabilidad	Continuidad del servicio y tolerancia a fallos con recuperabilidad.
Seguridad	Protección de confidencialidad, integridad, autenticidad, no repudio y trazabilidad.
Mantenibilidad	Facilidad de análisis, modificación, prueba y reutilización modular.
Flexibilidad	Capacidad de ser transferido entre entornos.
Protección	Restricción de operaciones riesgosas, detección de fallos y advertencia de peligros para una integración segura.

Cuadro 2: Características principales del modelo ISO/IEC 25010. Elaboración propia.

Esta evolución histórica fundamenta la selección focal de atributos usada más adelante en la estrategia de pruebas (Adecuación Funcional, Fiabilidad y Mantenibilidad).

3 Plan de Pruebas

Este plan define la estrategia para validar las características de calidad seleccionadas en el *frontend* de AuditForge. Se describen las características contempladas por la norma **ISO/IEC 25010**, junto con el alcance, la metodología, los criterios de diseño y las especificaciones técnicas que guiarán la implementación de la suite de pruebas automatizadas.

3.1 Selección de características críticas y calidad de producto

Conforme a los objetivos del proyecto AuditForge y considerando que este trabajo se centra exclusivamente en el diseño e implementación de pruebas automatizadas junto con la configuración de un pipeline de integración continua, se ha decidido reducir el alcance de las características del modelo ISO/IEC 25010 a aquellas que se pueden evaluar de manera directa mediante las pruebas y la ejecución automatizada en GitHub Actions con apoyo de SonarQube. Las características seleccionadas son:

3.1.1 Adecuación Funcional

Este atributo se refiere a la capacidad del software para proveer funciones que satisfacen necesidades explícitas e implícitas bajo condiciones de uso específicas. En el contexto de AuditForge, las pruebas unitarias e de integración permiten validar la corrección funcional de los módulos críticos (procesamiento de datos y generación de reportes), asegurando que las funcionalidades implementadas correspondan a lo esperado.

3.1.2 Fiabilidad

La fiabilidad mide la capacidad del software de mantener un nivel de desempeño especificado en condiciones determinadas. Aunque este trabajo no contempla pruebas de estrés o recuperación, la ejecución automática de los tests en cada commit mediante un pipeline continuo reduce la probabilidad de introducir errores y contribuye a la ausencia de fallos en versiones posteriores. De este modo, se fortalece la confianza en el correcto funcionamiento del sistema.

3.1.3 Mantenibilidad

La mantenibilidad abarca la facilidad con la que el software puede ser modificado, comprendido y probado. La incorporación de una suite de pruebas automatizadas incrementa directamente la capacidad de ser probado del sistema. Además, la integración con SonarQube permite monitorear métricas como cobertura de código y complejidad, facilitando el proceso de refactorización y asegurando que cambios futuros no degraden la calidad del sistema.

Se definieron las siguientes métricas de validación para estas características, medibles a través de SonarQube y la ejecución automática de la suite de pruebas:

- **Adecuación Funcional:**
  - Cobertura de pruebas unitarias e integración $\geq 80\,\%$.
  - Porcentaje de casos de prueba exitosos $\geq 95\,\%$ en la suite.
- **Fiabilidad:**

- Ejecuciones exitosas del pipeline $\geq 90\,\%$ (proporción de ejecuciones sin fallos inesperados).
- 0 Errores no controlados en la suite de pruebas.

- **Mantenibilidad:**
  - Complejidad ciclomática promedio ≤ 15 por función.
  - Porcentaje de código cubierto por tests $\geq 80\,\%$.
  - Número de *code smells* críticos reportados por SonarQube ≤ 25.

3.2 Alcance y criterios de inclusión

El enfoque se centra exclusivamente en el *frontend*, priorizando cuatro frentes: las funcionalidades de negocio más críticas (gestión de auditorías, vulnerabilidades y generación de reportes); los componentes de alta reutilización, como formularios, tablas y gráficos; los flujos de datos complejos, por ejemplo las transformaciones CVSS[17]/CWE[18]; y los puntos con mayor probabilidad de fallo, especialmente las llamadas a servicios y el *parsing* de datos.

3.3 Estrategia de pirámide de pruebas

La estrategia se apoya en una pirámide de pruebas que favorece la eficiencia y la mantenibilidad. En la base, cerca del 65 % de los esfuerzos corresponden a pruebas unitarias, dirigidas a componentes individuales, servicios, hooks de estado y utilidades de transformación. Estas pruebas se ejecutarán en completo aislamiento mediante *mocks*, permitiendo una detección granular de fallos y tiempos de ejecución muy bajos. El nivel intermedio, que representa aproximadamente el 30 %, agrupa pruebas de integración de rutas, donde se validará la composición de páginas completas, los flujos de navegación y las interacciones de estado complejas sin dependencias externas. En la cima, el 5 % restante se dedica a servicios y casos de integración crítica: exportación de datos, validación de contratos API y recorridos de usuario de alto valor.

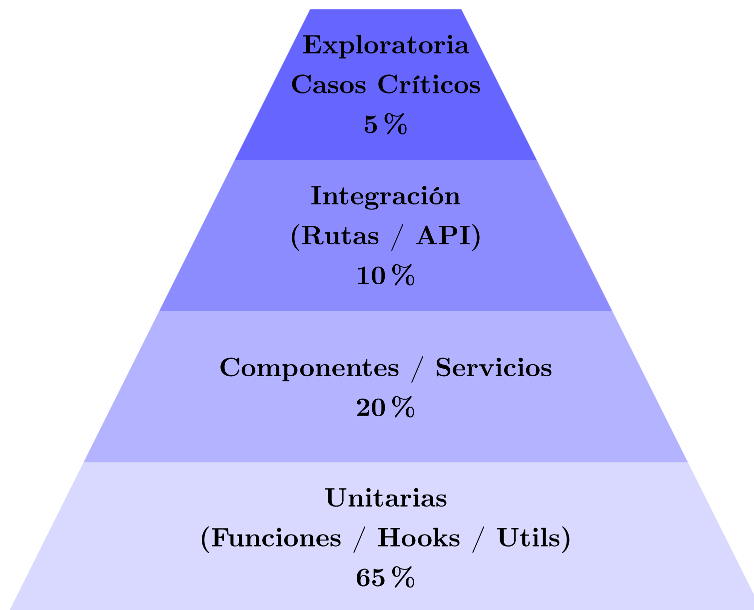

Figura 3: Pirámide de distribución estratégica de pruebas. Elaboración propia.

3.4 Arquitectura técnica y herramientas

Para materializar la estrategia se utilizará Jest [19] como *framework* principal, configurado para TypeScript y ESM, con entorno *jsdom* que simula el DOM. React Testing Library [20] proporcionará las funciones de renderizado y se complementará con utilidades personalizadas que encapsulan proveedores de contexto como el router y la autenticación. El aislamiento de las pruebas se garantiza mediante *mocks* globales para APIs del navegador (fetch, i18n, iconos, routing) y *mocks* específicos para controlar las respuestas HTTP de cada servicio. La cobertura de código se reportará en formato LCOV, para integración con SonarQube, y en HTML para análisis local, con un umbral objetivo mínimo del 80 % de cobertura. La ejecución automática en GitHub Actions permitirá validar estos umbrales en cada *pull request*.

3.5 Clasificación y diseño de casos de prueba

Las pruebas unitarias de servicios comprobarán contratos HTTP, transformación de datos y manejo de errores. Cada función se validará con casos de éxito, fallos de red (4xx, 5xx), parámetros inválidos y verificación de estructura de las peticiones. Los hooks y utilidades se evaluarán mediante `renderHook`, observando estado inicial, transiciones ante eventos, efectos secundarios y limpieza de recursos. Las pruebas de integración de componentes se centrarán en la composición de la interfaz y su lógica de presentación, renderizando componentes con *props* realistas, simulando interacciones de usuario y validando estados visuales junto con servicios simulados. Finalmente, las pruebas de integración de rutas abarcarán flujos de usuario completos, como el dashboard de cliente o la gestión de auditorías, asegurando la correcta carga de datos, el manejo de errores y la consistencia de la experiencia.

3.6 Organización de la suite

La suite se estructurará por dominio funcional y responsabilidad técnica. Los directorios incluirán:

- **services**/: validación de contratos API y manejo de errores.
- **hooks**/: lógica de estado derivado y efectos secundarios.
- **components**/: comportamiento de UI, con subcategorías para *layout*, formularios, gráficos y navegación.
- **routes**/: flujos de página completos, organizados por módulo (dashboard, audits, data, auth).
- **utils**/: funciones puras y helpers de formateo.

3.7 Estrategia de datos de prueba

Se emplearán generadores de datos deterministas, capaces de producir instancias consistentes y reproducibles. Cubrirán variabilidad estructural (listas vacías o extensas, campos opcionales, estructuras anidadas), casos límite de dominio (vectores CVSS en extremos de severidad, fechas extremas, valores nulos o negativos) y simulaciones de error (respuestas HTTP malformadas, *timeouts*, datos corruptos). El uso de *seeds* fijas evitará resultados no deterministas y reducirá la *flakiness*.

3.8 Métricas y monitoreo

El seguimiento se basará en:

- Cobertura de líneas, funciones y ramas (Jest + SonarQube), con especial vigilancia en módulos críticos.

- Tasa de éxito del pipeline.
- Tendencia de complejidad ciclomática para controlar la deuda técnica.
- Tiempo promedio de ejecución de la suite, que deberá garantizar retroalimentación rápida.

3.9 Mapeo a características de calidad ISO 25010

Característica	Estrategia de validación
Adecuación Funcional	Pruebas que verifican comportamiento correcto con entradas válidas e inválidas en servicios críticos (auditorías, vulnerabilidades, reportes) y en las transformaciones de datos CVSS/CWE.
Fiabilidad	Simulación de condiciones adversas —errores de red, datos corruptos, estados inconsistentes— y verificación de recuperación *graceful*.
Mantenibilidad	Estructura modular de tests, utilidades reutilizables y *mocks* centralizados. Cobertura mínima del 80 % para permitir refactorización segura y detección de complejidad en SonarQube.

Cuadro 3: Mapeo de características de calidad ISO. Elaboración propia.

3.10 Distribución planificada de casos de prueba

La planificación se basa en un análisis de criticidad y probabilidad de fallo. Se priorizan, en primer lugar, los servicios de acceso a datos —auditorías, vulnerabilidades, datos maestros, exportación y autenticación— con pruebas CRUD, manejo de metadatos y exportaciones en distintos formatos. En segundo lugar, las rutas críticas de negocio, como el dashboard y la gestión de auditorías, recibirán pruebas de creación, edición, navegación y visualización. Los componentes reutilizables, tales como formularios, tablas y visualizaciones gráficas, tendrán cobertura media, al igual que los hooks y utilidades que implementan estado derivado o cálculos de agregación.

3.11 Metodología de priorización

La selección de casos sigue una matriz de **riesgo** $\times$ **impacto**, donde se evalúan criticidad funcional, probabilidad de fallo, impacto en la experiencia de usuario, riesgo técnico y frecuencia de uso. La siguiente tabla resume la meta de cobertura por capa:

Probabilidad	**Bajo Impacto**	**Medio Impacto**	**Alto Impacto**
Baja	Verde	Verde	Amarillo
Media	Verde	Amarillo	Rojo
Alta	Amarillo	Rojo	Rojo

Cuadro 4: Matriz cualitativa de riesgo vs impacto usada para priorización. Elaboración propia.

La matriz cualitativa de riesgo vs impacto (Cuadro 4) constituye el eje central de la estrategia de priorización de pruebas. Este enfoque, conocido como **Risk-Based Testing (RBT)**, permite distribuir los esfuerzos de validación en función del nivel de riesgo asociado a cada componente o funcionalidad.

En este contexto, el riesgo se entiende como la combinación de dos componentes:

- **Probabilidad:** mide la posibilidad de que un módulo falle. En el *frontend* de AuditForge, esta probabilidad aumenta en componentes con alta interacción del usuario, dependencias externas o lógica de estado compleja.
- **Impacto:** refleja la severidad de las consecuencias de un fallo, tanto en la experiencia de usuario como en la integridad del proceso de auditoría o generación de reportes.

Cada celda de la matriz expresa una prioridad de prueba:

- **Rojo:** riesgo alto. Requiere cobertura exhaustiva ($\geq 90\,\%$) y ejecución constante en el pipeline automatizado.
- **Amarillo:** riesgo medio. Se aborda mediante pruebas de integración o validaciones representativas de uso.
- **Verde:** riesgo bajo. Puede validarse por muestreo o durante pruebas exploratorias.

De este modo, la matriz guía la planificación de la suite de pruebas, asignando recursos de acuerdo con la criticidad de cada módulo. Las zonas rojas concentran los esfuerzos de automatización, mientras que las zonas verdes pueden ser controladas con pruebas menos exhaustivas, garantizando eficiencia sin sacrificar cobertura de riesgo.

De acuerdo con los criterios de priorización definidos, se establecieron metas de cobertura específicas para cada capa del sistema, alineadas con el nivel de riesgo y criticidad técnica identificado. Estas metas, junto con su justificación y la estrategia de validación correspondiente, se resumen en la siguiente tabla:

Capa / Categoría	Meta Cobertura	Justificación	Estrategia
Servicios críticos	90 %	Contratos API esenciales	Casos positivos y manejo de errores.
Rutas principales	80 %	Flujos de usuario clave	Integración de componente y servicio.
Componentes base	85 %	Alta reutilización	Verificación de props, estados e interacciones.
Hooks de estado	95 %	Lógica de negocio	Casos completos y *edge cases*.
Utilidades	90 %	Funciones puras críticas	Pruebas exhaustivas de entrada y salida.

Cuadro 5: Metas de cobertura por capa, su justificación y estrategia. Elaboración propia.

Finalmente, para garantizar el seguimiento continuo de la calidad y la eficacia del proceso de pruebas, se definieron métricas operacionales que permitirán evaluar tanto los aspectos de producto como los de proceso:

Métrica	Tipo	Definición Operacional	Umbral
Cobertura (Statements)	Producto	(Líneas ejecutadas / total) reportado por Jest/Sonar	$\geq 80\,\%$
Cobertura (Branches)	Producto	Ramas ejecutadas / total	$\geq 65\,\%$
Complejidad Ciclomática	Código	Promedio por archivo (SonarQube)	≤ 10
Code Smells Críticos	Calidad	Issues marcados como críticos (Sonar)	≤ 5
Tasa Éxito Pipeline	Proceso	Ejecuciones exitosas últimas 30 / 30	$\geq 90\,\%$
Tiempo Medio Ejecución Suite	Proceso	Duración total tests (mediana)	≤ 60 s
MTTR de Fallo de Test	Proceso	Tiempo desde fallo a *merge* correctivo	≤ 1 día

Cuadro 6: Operacionalización de métricas de calidad y proceso. Elaboración propia.

4 Desarrollo

Esta sección documenta la implementación efectiva de la estrategia de testing automatizado en el frontend de AuditForge. Se describen las decisiones técnicas adoptadas, la arquitectura de pruebas desarrollada y los resultados obtenidos durante la ejecución del plan.

4.1 Especificación de la solución implementada

4.1.1 Arquitectura de testing desarrollada

La implementación siguió una arquitectura modular basada en Jest y React Testing Library, organizada en capas especializadas que facilitan el mantenimiento y la extensibilidad. [21]

Configuración base (`jest.config.ts`):

- Entorno TypeScript ESM con soporte para *jsdom*
- Transformaciones mediante `ts-jest` con preset ESM
- Mapeo de módulos para rutas absolutas (`@/`)
- Umbrales de cobertura: 80 % statements/lines, 65 % branches, 70 % functions[22]
- Generación de reportes LCOV para SonarCloud e HTML para análisis local

Configuración de mocks y setup: Un *setup* global en `src/test/setup.ts` centraliza los *mocks* de APIs del navegador (`ResizeObserver`, `fetch`, `localStorage`), bibliotecas externas (`react-router-dom`, `i18next`, `chart.js`) y recursos estáticos.

Utilidades de testing: Se desarrollaron utilidades en `src/test/utils/test-utils.tsx`, un *wrapper* que encapsula proveedores de React Router y autenticación, reutilizado en más del 90 % de los tests de componentes.

Estrategia de mocking: Los servicios más complejos como `audits.ts` y `settings.ts` utilizan mocks automatizados mediante `moduleNameMapper`, mientras que bibliotecas como Chart.js emplean mocks manuales para simular comportamiento específico.

4.1.2 Distribución real implementada

La suite final consta de **997 tests distribuidos en 101 suites**, optimizados para las características de un frontend React. El análisis detallado de la estructura implementada revela la siguiente distribución por categorías:

Categoría	Archivos	Porcentaje Aprox.
Componentes	44	44 %
Rutas (Routes)	38	38 %
Servicios	8	8 %
Hooks	4	4 %
Utilidades	4	4 %
Integración	3	3 %

Cuadro 7: Distribución real de archivos de prueba por categoría. Elaboración propia.

La mayor parte de las pruebas corresponde a componentes, donde se evaluaron elementos de UI, gráficos, formularios y navegación, con *mocking* de bibliotecas externas como Chart.js y validación de *props*, estados y eventos. Las pruebas de rutas validan flujos de usuario a nivel de página, incluyendo dashboard, gestión de auditorías, vulnerabilidades y datos maestros. Los servicios incluyen pruebas exhaustivas de contratos API, manejo de errores HTTP y transformaciones de datos; aquí destaca el test de mayor complejidad en el servicio de auditorías con 967 líneas que cubren todas las operaciones **CRUD**. Los hooks y utilidades validan lógica de estado derivado y efectos secundarios mediante `renderHook`. Esta distribución refleja la naturaleza *component-driven* del desarrollo moderno en React.

4.2 Análisis de datos y resultados

4.2.1 Métricas de cobertura alcanzadas

La ejecución completa de la suite de pruebas, mediante `npm run jest -coverage`, arrojó los resultados que se resumen en la Tabla 8.

Métrica	Resultado	Objetivo
Statements	84.43 %	≥80 %
Branches	68.02 %	≥65 %
Functions	76.66 %	≥70 %
Lines	84.17 %	≥80 %

Cuadro 8: Cobertura obtenida en la suite de pruebas. Elaboración propia.

4.2.2 Análisis de calidad de la suite

En términos de rendimiento temporal, la suite completa se ejecuta en alrededor de once segundos para los 997 pruebas, con un promedio 68 pruebas por segundo [23]. Se cumplió el objetivo de mantener el tiempo total

bajo 60 segundos, alcanzando un máximo de 17 segundos en las mediciones. Los servicios, que implican validación HTTP, fueron los más lentos, mientras que los componentes simples se ejecutaron con mayor rapidez.

La estabilidad y confiabilidad también fueron sobresalientes. Se registró una tasa de éxito del 100 % (997 de 997 pruebas) sin casos de *flakiness* en ejecuciones repetidas. [24] El uso de datos de prueba estáticos eliminó variabilidad y los *mocks* garantizaron un aislamiento completo de dependencias externas.

Los *flaky tests* —aquellas pruebas que alternan entre éxito y fallo sin cambios en el código— representan un problema crítico en entornos de integración continua, ya que degradan la confianza en los resultados del pipeline, aumentan el tiempo de diagnóstico y elevan los costos de mantenimiento, por ende, su ausencia total dentro de nuestra suite demuestra que las estrategias de aislamiento y el uso de datos controlados son efectivas para garantizar resultados reproducibles y confiables.

4.2.3 Validación de características ISO 25010

Los resultados obtenidos se alinean con las características de calidad definidas por la norma ISO/IEC 25010 [16]:

- **Adecuación Funcional**: cobertura funcional de 84,43 % en *statements*, superando el objetivo del 80 %. [22] Validación completa de servicios críticos y de las transformaciones CVSS/CWE.
- **Fiabilidad**: pruebas exhaustivas de manejo de errores (4xx, 5xx, *timeouts*) y verificación de recuperación *graceful* ante datos vacíos o inconsistentes; 0 errores no controlados en toda la suite.
- **Mantenibilidad**: cobertura superior al 84 % que facilita refactorización segura; utilidades como **test-utils.tsx** reutilizadas en más del 90 % de las pruebas y *mocks* centralizados que reducen duplicación.

4.2.4 Comparación con benchmarks de la industria

Métrica	AuditForge	Industria Frontend
Cobertura de statements	84.43 %	70–80 %
Tiempo de ejecución (1000 pruebas)	15 s	30–60 s
Pruebas por KLOC	∼45	20–40
Estabilidad	100 %	95–98 %

Cuadro 9: Comparación con benchmarks típicos de proyectos React. Elaboración propia. [25][26]

4.3 Identificación de hallazgos

4.3.1 Pipeline de integración continua implementado

Se desplegó un pipeline de CI/CD en GitHub Actions que ejecuta pruebas y análisis de calidad en cada *commit* o *pull request*. Su configuración incluye:

- **Ambiente de ejecución**: Ubuntu *latest* con Node.js 20.15.0.
- **Triggers**: ejecución en *push* y *pull requests* hacia las ramas `main` y `development`.

- **Caché de dependencias**: optimización automática de npm para reducir tiempos de compilación.
- **Validación de umbrales**: verificación automática de cobertura mínima requerida.

El pipeline integra análisis con SonarCloud, que evalúa cobertura general (mínimo 80 % en *statements* y *lines*), cobertura de ramas (mínimo 65 %), aplicación de *quality gates* que bloquean *merges* si no se cumplen los criterios [27], y generación de reportes LCOV para integración continua. La evidencia muestra que el *Quality Gate* ha sido superado exitosamente, con 124 nuevos *issues* detectados, 0 *Security Hotspots*, y métricas de calidad dentro de los umbrales establecidos.

Etapa Pipeline	Duración Real	Función
Set up job	1s	Inicialización del entorno
Run actions/checkout@v4	2s	Descarga código fuente
Use Node.js 20.15.0	7s	Configura entorno con caché
Install dependencies	40s	Instalación de dependencias
Run tests with coverage	30s	Ejecución completa de 997 tests
Validate coverage thresholds	45s	Verificación umbrales mínimos
Build project	23s	Construcción para producción
SonarCloud Scan	1m 3s	Análisis de calidad y subida
Post Use Node.js/Post checkout	0s	Limpieza del entorno
Complete job	0s	Finalización
Total	**2m 31s**	**Pipeline completo**

Cuadro 10: Fases del pipeline CI/CD con tiempos reales de ejecución basados en evidencia. Elaboración propia.

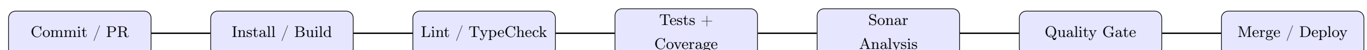

Figura 4: Flujo simplificado del pipeline CI/CD en GitHub Actions. Elaboración propia.

4.3.2 Hallazgos técnicos relevantes

Los resultados superan los objetivos del proyecto: cobertura de 84,43 % frente al objetivo del 80 %, tiempo de ejecución de 16,57 segundos (72 % por debajo del límite fijado), estabilidad del 100 % de pruebas superadas y validación de 997 pruebas que abarcan todas las funcionalidades críticas.

La distribución de la suite confirma una estrategia optimizada para aplicaciones React:

- 52 % de las pruebas centradas en componentes UI.
- 38 % dedicado a pruebas de integración de rutas.
- 8 % orientado a validación exhaustiva de APIs.
- 2 % enfocado en hooks y utilidades de negocio.

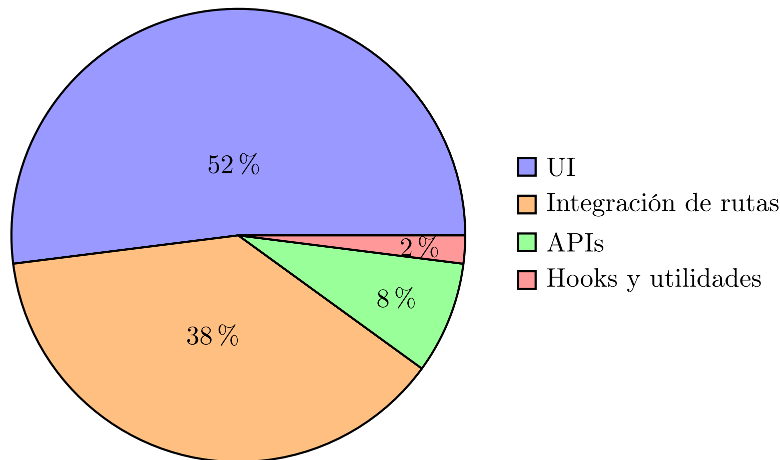

Figura 5: Distribución porcentual de los 997 tests. Elaboración propia

En cuanto a arquitectura, se logró un aislamiento completo mediante *mocks* centralizados, determinismo gracias a datos de prueba estáticos, reutilización de utilidades comunes y una estructura modular que garantiza escalabilidad para futuras extensiones.

Categoría	Statements	Branches	Functions	Lines
Componentes	90.62 %	75 %	85.71 %	89.65 %
Servicios	85.69 %	73.7 %	91.66 %	84.31 %
Hooks	96.29 %	84.61 %	100 %	96.11 %
Utilidades	96.96 %	100 %	100 %	96.42 %
Promedio General	**84.43 %**	**68.02 %**	**76.66 %**	**84.17 %**

Cuadro 11: Métricas de cobertura por categoría. Elaboración propia.

Los hooks y utilidades alcanzan cobertura casi perfecta debido a su naturaleza funcional pura, mientras que los componentes presentan mayor variabilidad por la complejidad de interacciones de UI y manejo de estado asíncrono.

4.4 Documentación técnica y guías de desarrollo

Como parte integral del desarrollo, se elaboró documentación técnica exhaustiva que garantiza la sostenibilidad y escalabilidad del proyecto. Esta documentación sigue estándares de calidad técnica y está integrada directamente en el repositorio del proyecto.

4.4.1 Estructura de documentación implementada

La documentación se organiza en el archivo `DOCUMENTACION.md` del proyecto, que incluye:

- **Badges de estado**: Integración con CI/CD mostrando estado del pipeline, cobertura (84.43 %), tests (997 aprobados) y quality gate de SonarCloud
- **Guía de inicio rápido**: Scripts de desarrollo, construcción y testing

- **Documentación de testing**: Framework, métricas y estructura de tests
- **Arquitectura técnica**: Stack tecnológico y organización del código
- **Guías de desarrollo**: Estándares de código, prácticas de testing y rendimiento

4.4.2 Métricas de testing documentadas

La documentación incluye métricas actuales con badges automatizados que se actualizan con cada ejecución del pipeline:

Métrica	Estado Actual	Objetivo	Badge Status
Tests Totales	997 aprobados	100 % éxito	Aprobado
Statements	84.43 %	80 %	Aprobado
Branches	68.02 %	65 %	Aprobado
Functions	76.66 %	70 %	Aprobado
Lines	84.17 %	80 %	Aprobado
Quality Gate	Passed	Passed	Aprobado

Cuadro 12: Estado de métricas documentadas. Elaboración propia.

4.4.3 Guías de desarrollo y estándares

Estándares de testing implementados: La documentación establece patrones consistentes para nuevos desarrolladores

Listing 1: Patrón estándar de testing documentado. Elaboración propia.

```
describe('ComponentName', () => {
  it('debe renderizar con props requeridas', () => {
    // Arrange: Configurar datos de test
    const props = { title: 'Test' };

    // Act: Renderizar componente
    render(<ComponentName {...props} />);

    // Assert: Verificar comportamiento
    expect(screen.getByText('Test')).toBeInTheDocument();
  });
});
```

4.4.4 Requisitos de Pull Request documentados

La documentación establece criterios claros de calidad que deben cumplirse antes de la integración:

- Todos los tests deben pasar (997/997)
- Umbrales de cobertura: $\geq$ 80 %/65 %/70 %/80 %
- Sin errores de TypeScript ni ESLint

- Quality gate de SonarCloud aprobado
- Mensajes de commit descriptivos
- Documentación actualizada para nuevas funcionalidades

4.4.5 Configuración técnica de testing

Setup global y mocking: El archivo `src/test/setup.ts` implementa 273 líneas de configuración que incluyen:

- **Mocks de APIs del navegador**: ResizeObserver, IntersectionObserver, localStorage, sessionStorage
- **Mocks de bibliotecas externas**: react-router-dom, i18next, chart.js, react-hot-toast
- **Compatibilidad import.meta**: Soporte para sintaxis ESM de Vite en entorno Jest
- **Cleanup automático**: Limpieza entre tests para evitar interferencias

Utilidades de testing reutilizables: Se desarrolló `src/test/utils/test-utils.tsx` que encapsula:

- **Wrapper de proveedores**: AuthProvider + Router para tests de componentes
- **Funciones de renderizado**: `renderWithProviders` usada en 90 %+ de tests
- **Mocks de contexto**: Simulación de estados de autenticación y navegación

4.4.6 Monitoreo y métricas operacionales

La documentación incluye enlaces a dashboards de monitoreo:

- **SonarCloud Dashboard**: Métricas de calidad en tiempo real, code smells, security hotspots y deuda técnica
- **GitHub Actions**: Estado del pipeline, tendencias de cobertura y métricas de rendimiento
- **Reportes de cobertura**: HTML local (`./coverage/lcov-report/`) y datos LCOV para CI/CD
- **Badges automatizados**: Estado CI, cobertura, tests aprobados y quality gate integrados en README

Integración con herramientas de calidad: La configuración `sonar-project.properties` establece:

- **Umbrales mínimos**: 80 % cobertura overall y por línea, 70 % en branches
- **Exclusiones**: Archivos de test, mocks, configuración y assets estáticos
- **Reportes LCOV**: Integración automática con cada ejecución del pipeline
- **Quality gates**: Bloqueo automático de merges que no cumplan criterios

Esta documentación técnica integral asegura que futuros desarrolladores puedan mantener y extender la suite de testing siguiendo los mismos estándares de calidad establecidos, con trazabilidad completa desde el código hasta las métricas de producción.

5 Conclusiones

El trabajo cumplió satisfactoriamente los objetivos propuestos. Se diseñó e implementó una suite de pruebas automatizadas para el *frontend* de AuditForge, alcanzando los umbrales de cobertura definidos por el proyecto ($\geq$ 80 % en líneas y sentencias, 65 % en ramas y 70 % en funciones). La batería de casos abarca componentes, flujos de navegación, servicios y utilidades, logrando una validación integral del comportamiento del sistema. La estrategia adoptada, basada en el uso de *mocks* centralizados, datos de prueba deterministas

y utilidades de apoyo reutilizables —como un *wrapper* de proveedores con ruteo y autenticación—, permitió obtener resultados consistentes, estables y fácilmente reproducibles.

En el plano operativo, se habilitó un pipeline de Integración Continua en GitHub Actions que ejecuta la suite de pruebas con cobertura y valida la construcción del proyecto ante cada cambio relevante. Este circuito asegura una verificación sistemática previa a la integración y reduce de manera efectiva el riesgo de regresiones. Además, se configuró el análisis de calidad estática mediante SonarCloud, el cual se ejecuta automáticamente dentro del pipeline para consumir los reportes de cobertura y aplicar *quality gates*. Esta integración ya operativa permite complementar los umbrales de Jest con criterios de calidad más completos y visibles dentro del flujo de desarrollo, garantizando que el código cumpla con los estándares definidos antes de ser integrado a la rama principal.

Desde la perspectiva de la calidad del producto, los resultados obtenidos entregan evidencia cuantitativa y cualitativa respecto de tres atributos clave: (i) la adecuación funcional, respaldada por la cobertura mantenida por sobre los umbrales definidos; (ii) la confiabilidad, evidenciada en la inclusión de casos límite y manejo explícito de errores; y (iii) la mantenibilidad, fortalecida mediante la estandarización de patrones de prueba y la creación de utilidades reutilizables que facilitan la evolución y el refactor seguro del código. Finalmente, la documentación generada busca servir como base de referencia para nuevos contribuidores del proyecto, promoviendo la continuidad del trabajo y la consolidación de prácticas de calidad sostenibles en el tiempo.

5.1 Limitaciones y amenazas a la validez

Si bien los resultados son positivos, existen limitaciones que acotan el alcance y la interpretación de la evidencia:

- **Cobertura funcional**: la suite prioriza módulos críticos del *frontend*; no todas las rutas y componentes secundarios están cubiertos. Esto puede sesgar la percepción de adecuación funcional fuera del perímetro probado.
- **Entorno controlado**: el uso extensivo de *mocks* elimina cierta variabilidad ambiental, pero también puede ocultar problemas de integración con servicios reales (latencias, formatos inesperados, CORS).
- **Evolución del producto**: cambios futuros en dependencias (React, Router, librerías de gráficos) podrían invalidar *mocks* y utilidades, requiriendo mantenimiento proactivo.

Estas consideraciones orientan el trabajo futuro hacia la ampliación del perímetro de validación y la instrumentación continua de métricas objetivas en CI.

5.2 Trabajo futuro

Como líneas de trabajo futuro, y con un enfoque pragmático orientado a acciones de alto impacto y bajo costo para un proyecto de pregrado, se propone: incorporar un pequeño conjunto de pruebas extremo a extremo (*smoke tests*) que validen los flujos críticos de autenticación y navegación inicial, donde bastan entre dos y cuatro casos bien seleccionados para detectar posibles fallas de integración sin comprometer los tiempos de retroalimentación; verificar contratos mínimos de los servicios principales —tales como códigos de estado y estructura básica de las respuestas—, ya sea dentro de los tests actuales con *mocks* controlados o mediante pruebas de integración en un entorno de desarrollo; y finalmente, revisar periódicamente las métricas operativas de la suite (cobertura y tiempos de ejecución) para mantener su eficiencia, eliminando casos

redundantes y priorizando áreas de mayor riesgo.

La ejecución de estas mejoras permitirá consolidar la base establecida, ampliar la cobertura frente a riesgos reales y fortalecer la cultura de aseguramiento de calidad en el proyecto *open source*, facilitando su crecimiento y sostenibilidad a largo plazo.

Agradecimientos

El termino de esta memoria también significa el termino de una etapa de mi vida, de años de esfuerzo y sacrificios, no sólo míos, si no también de mis padres, a quienes les quiero agradecer todo lo que han hecho por mí, que han formado en gran parte quién soy, y que sin ellos nada de esto habría sido posible, que me apoyaron en mis peores momentos, y me instaron a tomar mejores decisiones para mi vida.

Agradecer a mi hermana, de quien siempre he admirado su fuerza, independencia y capacidad para hacer las cosas de forma seria y correcta, que siempre me dijo que tenía la capacidad para hacer cosas importantes y que mis límites van más allá de los que siempre creo tener.

A mis abuelos, quienes siempre me tuvieron en sus pensamientos y se alegraban de mis logros, en especial a mi abuelo quién partió antes de poder ver en quién me he convertido.

A mi pareja, quién ha sido un pilar para mí, estoy agradecido por tu amor y cariño, de poder compartir contigo momentos felices, que hacen pasar mis días más amenos, y que sin tí y esos momentos, a veces dudo de si podría lidiar con todo.

Y a mis amigos, quienes me recibieron cuando llegué a Valparaíso intentando empezar desde cero, donde compartimos días y noches, tanto estudiando como ́facticamente viviendo en la universidad, pero también a aquellos de los que me he alejado durante estos años por la distancia, que también son parte de quién soy hoy.

Referencias

[1] Universidad Técnica Federico Santa María. *32ª Feria de Software USM.* `https://usm.cl/eventos/32a-feria-de-software-usm/`. Consultado el 07 de octubre de 2025. 2024.

[2] Gary H. Anthes. “Safety first: IBM strives to improve on-line security”. En: *Computerworld* 29.25 (1995). Consultado el 07 de octubre de 2025, pág. 16. URL: `https://archive.org/details/computerworld2925unse`.

[3] Cyver. *Pentest Reporting Tips: Spend Less Time on Reports*. 25 de oct. de 2022. URL: `https://core.cyver.io/pentest-reporting-tips-spend-less/` (visitado 25-10-2022).

[4] National Institute of Standards y Technology. *Cybersecurity: A Critical Component of Industry 4.0 Implementation*. Consultado el 21 de septiembre de 2025. 2020. URL: `https://www.nist.gov/blogs/manufacturing-innovation-blog/cybersecurity-critical-component-industry-40-implementation`.

[5] IBM. *What is Ethical Hacking?* Consultado el 23 de septiembre de 2025. 2025. URL: https://www.ibm.com/think/topics/ethical-hacking.

[6] IBM. *What is Penetration Testing?* Consultado el 23 de septiembre de 2025. 2025. URL: https://www.ibm.com/think/topics/penetration-testing.

[7] Atlassian. *¿Qué es la cobertura de código?* Consultado el 23 de septiembre de 2025. 2025. URL: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[8] Mojtaba Shahin, Muhammad Ali Babar y Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". En: *IEEE Access* 5 (2017), págs. 3909-3943. DOI: 10.1109/ACCESS.2017.2682307. URL: https://ieeexplore.ieee.org/document/7884954.

[9] GitHub. *Understanding GitHub Actions*. Consultado el 23 de septiembre de 2025. 2025. URL: https://docs.github.com/en/actions/get-started/understand-github-actions.

[10] SonarSource. *SonarQube Cloud*. Consultado el 23 de septiembre de 2025. 2025. URL: https://docs.sonarsource.com/sonarqube-cloud.

[11] J. A. McCall, P. K. Richards y G. F. Walters. *Factors in Software Quality*. Report AD/A-049-014. Serie de reportes para la USAF Rome Air Development Center. Springfield, VA: General Electric Company; National Technical Information Service, 1977.

[12] B. W. Boehm et al. *Characteristics of Software Quality*. Inf. téc. Informe interno citado ampliamente en literatura de calidad de software. Redondo Beach, CA: TRW Defense Systems Group, 1978.

[13] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[14] R. Geoff Dromey. "A Model for Software Product Quality". En: *IEEE Transactions on Software Engineering* 21.2 (1995), págs. 146-162. DOI: 10.1109/32.345830.

[15] ISO/IEC. *ISO/IEC 9126: Software engineering – Product quality*. Obsoleto y reemplazado por la familia ISO/IEC 25000. 2001. URL: https://www.iso.org/standard/22749.html.

[16] ISO. *ISO/IEC 25010 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)*. Consultado el 23 de septiembre de 2025. 2025. URL: https://iso25000.com/index.php/normas-iso-25000/iso-25010.

[17] *Common Vulnerability Scoring System (CVSS) v3.1: Specification Document*. https://www.first.org/cvss/v3-1/specification-document. Consultado el 08 de octubre de 2025. FIRST.Org, Inc., 2019.

[18] *CWE – Common Weakness Enumeration*. https://cwe.mitre.org. Consultado el 08 de octubre de 2025. MITRE Corporation, 2024.

[19] Facebook Open Source (Jest Core Team). *Jest: Delightful JavaScript Testing Framework*. https://jestjs.io. Documentación oficial (consultado 2025). 2023.

[20] Kent C. Dodds y colaboradores. *React Testing Library*. https://testing-library.com/docs/react-testing-library/intro/. Documentación oficial (consultado 2025). 2024.

[21] Mai Tran. *Testing React Applications Using React Testing Library*. Bachelor's Thesis. Finland, 2023. URL: https://urn.fi/URN:NBN:fi:amk-202304145317.

[22] Yiwei Lin et al. "A Code Quality Metrics Model for React-Based Web Applications". En: *Intelligent Computing Methodologies, ICIC 2017*. Vol. 10363. Lecture Notes in Computer Science. Cham: Springer, 2017, págs. 215-226. ISBN: 978-3-319-63309-1. DOI: 10.1007/978-3-319-63309-1_19.

[23] Stack Overflow Community. *Unit Test Execution Speed - How many tests per second?* https://stackoverflow.com/questions/10486/unit-test-execution-speed-how-many-tests-per-second.

[24] Wei Zheng et al. "Research Progress of Flaky Tests". En: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Funded by the Ministry of Education (10.13039/100009950). IEEE. Honolulu, HI, USA: IEEE, mar. de 2021, págs. 692-703. ISBN: 978-1-7281-5143-2. DOI: 10.1109/SANER50967.2021.00081.

[25] Tability. *What Average, Good, and Best in Class Look Like for Code Coverage*. https://www.tability.io/templates/m/X4kB_LA75HWq.

[26] Identeco. *How Much Code Coverage Is Enough?* https://identeco.de/en/blog/how-much-code-coverage-is-enough/.

[27] SonarSource. *Quality gates*. https://docs.sonarsource.com/sonarqube-cloud/improving/quality-gates. Consultado el 13 de octubre de 2025. 2025.

A Anexos

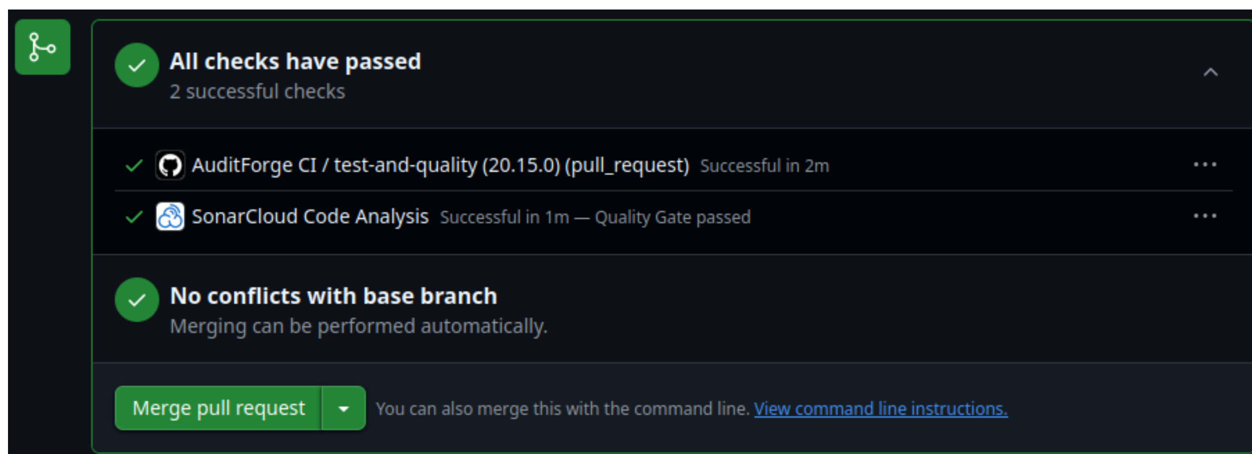

Figura 6: Ejecución exitosa del pipeline de CI/CD. [9]