Universidad de las Fuerzas Armadas ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería en Software

Calidad de Software

NRC: 27894

Análisis Estático de Código y Mejora Continua con SonarQube

Estudiante:

Axel Lenin PULLAGUARI CEDEÑO

Docente:

Angel Geovanny CUDCO POMAGUALLI

Sangolquí, Ecuador

8 de enero de 2026

Resumen

Este informe presenta los resultados del laboratorio práctico sobre **Análisis Estático de Código** utilizando **SonarQube**. El objetivo principal fue auditar y mejorar la calidad de una aplicación web ("Parking App") que presentaba inicialmente una alta deuda técnica y vulnerabilidades críticas.

Se desplegó un entorno local mediante **Docker Compose** para orquestar el servidor de análisis y su base de datos. A través de un ciclo iterativo de escaneo, refactorización y pruebas unitarias (Test Driven Development), se logró:

- Eliminar vulnerabilidades de seguridad (SQL Injection).
- Reducir drásticamente los Çode Smells".
- Alcanzar una cobertura de código superior al 80 % mediante pruebas con **Jest**.

El resultado final es una aplicación robusta que cumple con un **Quality Gate** estricto, demostrando la importancia de integrar herramientas de calidad continua en el ciclo de vida del desarrollo de software.

Repositorio del Proyecto: https://github.com/ALPullaguariSW/LabSonarQube

Índice general

1 Introducción **3**

2 Objetivos **4**

2.1 Objetivo General 4

2.2 Objetivos Específicos 4

3 Marco Teórico **5**

3.1 Análisis Estático de Código (SAST) 5

3.2 SonarQube 5

3.2.1 Arquitectura 5

3.3 Métricas de Calidad de Software 5

3.3.1 Fiabilidad (Reliability) 6

3.3.2 Seguridad (Security) 6

3.3.3 Mantenibilidad (Maintainability) 6

3.4 Cobertura de Código (Code Coverage) 6

3.5 OWASP Top 10 7

4 Metodología **8**

4.1 Fase 1: Configuración del Entorno (Setup) 8

4.2 Fase 2: Análisis Inicial (Diagnóstico) 8

4.3 Fase 3: Configuración de Calidad y Refactorización 10

4.3.1 Backend 11

4.3.2 Frontend 12

4.4 Fase 4: Testing y Cobertura 12

5 Resultados **14**

5.1 Mejora de Métricas de Calidad 14

5.2 Cobertura de Código 14

5.3 Estado Final del Quality Gate 15

6 Discusión **16**

7 Conclusiones **17**

Introducción

La calidad del software no es un atributo accidental, sino el resultado de un proceso deliberado de diseño, codificación y verificación. En el desarrollo moderno, la deuda técnica —entendida como el costo implícito de retrabajar código debido a soluciones rápidas o malas prácticas— puede acumularse rápidamente, comprometiendo la seguridad, mantenibilidad y escalabilidad de los proyectos.

El análisis estático de código (SAST) se ha convertido en una práctica estándar para identificar estos problemas en etapas tempranas del ciclo de vida de desarrollo de software (SDLC). Herramientas como **SonarQube** permiten automatizar la revisión de código, detectando çode smells", vulnerabilidades de seguridad y errores lógicos que podrían pasar desapercibidos en revisiones manuales.

En el presente laboratorio, se aborda la auditoría y refactorización de una aplicación web heredada ("Vulnerable Parking App"), la cual presenta múltiples deficiencias de calidad. A través de la configuración de un entorno de análisis con Docker y SonarQube, se busca no solo identificar fallos críticos (como inyecciones SQL o exposiciones de datos sensibles), sino también implementar una cultura de Çlean Code"mediante la corrección sistemática de errores y la implementación de pruebas unitarias robustas para asegurar una cobertura de código aceptable.

El informe detalla el flujo de trabajo seguido: desde el despliegue de la infraestructura de análisis, pasando por la identificación y remediación de cientos de incidencias, hasta la consecución de un "Quality Gate.^{ex}itoso.

Objetivos

2.1 Objetivo General

Evaluar, corregir y asegurar la calidad del código fuente de una aplicación web completa mediante el uso de la plataforma SonarQube, aplicando principios de refactorización y pruebas unitarias para cumplir con estándares estrictos de seguridad y mantenibilidad.

2.2 Objetivos Específicos

- **Despliegue de Infraestructura**: Configurar un entorno de análisis estático local utilizando Docker Compose para orquestar los servicios de SonarQube y PostgreSQL.
- **Diagnóstico**: Ejecutar escaneos iniciales para identificar deudas técnicas, vulnerabilidades de seguridad (Security Hotspots) y malas prácticas en el código Backend (Node.js) y Frontend (ExtJS).
- **Refactorización (Remediation)**: Aplicar técnicas de Çlean Code"para resolver los problemas detectados, incluyendo la migración a estándares modernos (ES Modules) y la mitigación de riesgos de seguridad (SQL Injection, SRI).
- **Aseguramiento de Calidad (Testing)**: Implementar pruebas unitarias e integración con Jest y Supertest, alcanzando una cobertura de código superior al 80 % en la lógica de negocio.
- **Configuración de Quality Gate**: Personalizar las reglas y exclusiones del proyecto para lograr un estado "Passed.[en] el Quality Gate, eliminando falsos positivos.

Marco Teórico

3.1 Análisis Estático de Código (SAST)

El Análisis Estático de Pruebas de Seguridad de Aplicaciones (SAST, Static Application Security Testing) es una metodología de prueba "White-Box"que analiza el código fuente, el código de bytes o los binarios de una aplicación en busca de condiciones de seguridad y calidad indicativas de vulnerabilidades. A diferencia del análisis dinámico (DAST), SAST se realiza sin ejecutar la aplicación, lo que permite identificar errores en fases tempranas del ciclo de desarrollo (Shift-Left Testing).

3.2 SonarQube

SonarQube es una plataforma de código abierto para la inspección continua de la calidad del código. Realiza revisiones automáticas para detectar errores, olores de código (code smells) y vulnerabilidades de seguridad en más de 20 lenguajes de programación.

3.2.1 Arquitectura

- **SonarQube Server**: El núcleo que procesa los informes de análisis, gestiona la base de datos y sirve la interfaz web.
- **SonarScanner**: El agente que se ejecuta en la máquina del desarrollador o en el servidor de CI/CD. Analiza el código fuente línea por línea y envía el informe resultante al servidor.
- **Base de Datos**: Almacena las métricas, problemas y configuraciones (en este laboratorio, PostgreSQL).

3.3 Métricas de Calidad de Software

SonarQube basa su análisis en tres pilares fundamentales:

3.3.1 Fiabilidad (Reliability)

Mide la capacidad del software para realizar sus funciones requeridas bajo condiciones específicas. En SonarQube, esto se traduce en la detección de **Bugs**: errores de programación que pueden provocar fallos en tiempo de ejecución (ej. NullPointerExceptions, bucles infinitos).

3.3.2 Seguridad (Security)

Se divide en dos categorías:

- **Vulnerabilidades**: Fallos confirmados que pueden ser explotados por atacantes (ej. Inyección SQL, XSS). Requieren corrección inmediata.
- **Security Hotspots**: Fragmentos de código sensibles a la seguridad que requieren revisión manual para confirmar si son seguros o no (ej. configuración de CORS, uso de cookies).

3.3.3 Mantenibilidad (Maintainability)

Relacionada con la facilidad con la que el código puede ser modificado. SonarQube utiliza el concepto de **Code Smells** (Olores de Código): patrones que no son necesariamente errores, pero indican debilidades de diseño o dificultades de mantenimiento (ej. código duplicado, funciones demasiado complejas, "magic numbers"). Se mide a través de la **Deuda Técnica** (el tiempo estimado para corregir estos problemas).

3.4 Cobertura de Código (Code Coverage)

Es una métrica que indica el porcentaje de código fuente que es ejecutado durante las pruebas automatizadas (Unit Testing).

$$\text{Cobertura} = \frac{\text{Líneas ejecutadas por tests}}{\text{Total de líneas de código}} \times 100$$

Una alta cobertura reduce la probabilidad de regresiones (bugs introducidos por nuevos cambios) y es un requisito común en los Quality Gates.

3.5 OWASP Top 10

El proyecto OWASP (Open Web Application Security Project) publica periódicamente una lista de los 10 riesgos de seguridad más críticos en aplicaciones web. En este laboratorio, nos centramos especialmente en:

- **A03:2021 – Injection**: Ocurre cuando datos no confiables son enviados a un intérprete (como SQL) como parte de un comando o consulta. La remediación estándar es el uso de declaraciones preparadas (Prepared Statements).
- **A05:2021 – Security Misconfiguration**: Configuraciones de seguridad faltantes o incorrectas (ej. no usar cabeceras de seguridad o exponer trazas de error detalladas al usuario).

Metodología

La metodología aplicada sigue el ciclo de mejora continua (Plan-Do-Check-Act) adaptado al análisis de código estático.

4.1 Fase 1: Configuración del Entorno (Setup)

Se utilizó **Docker Compose** para levantar los servicios necesarios. El archivo `docker-compose.yml` provisionó:

- **SonarQube Community Edition**: Servidor de análisis.
- **PostgreSQL**: Base de datos persistente para SonarQube y para la aplicación "Parking App".

Se generó un token de seguridad y se configuró el archivo `sonar-project.properties` para definir el alcance del análisis.

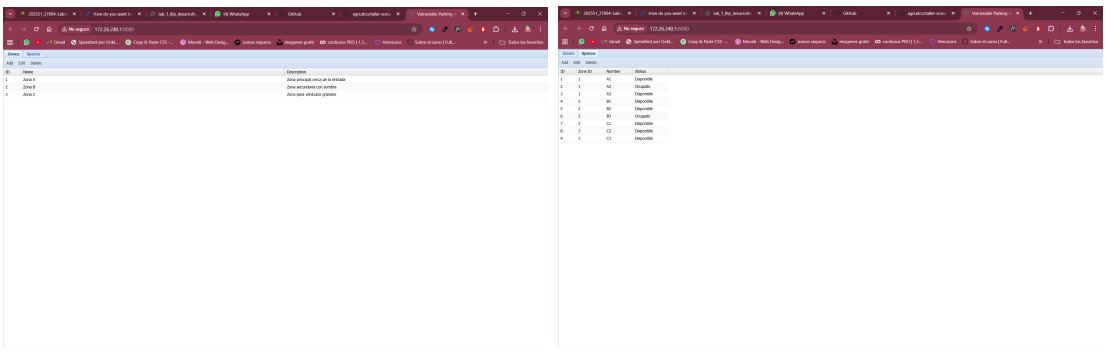

Figura 1 *Aplicación "Parking App.^en funcionamiento (Zonas y Espacios) previo al análisis.*

4.2 Fase 2: Análisis Inicial (Diagnóstico)

Se ejecutó el comando `sonar-scanner` sin configuraciones avanzadas inicialmente, y luego con el archivo de propiedades básico.

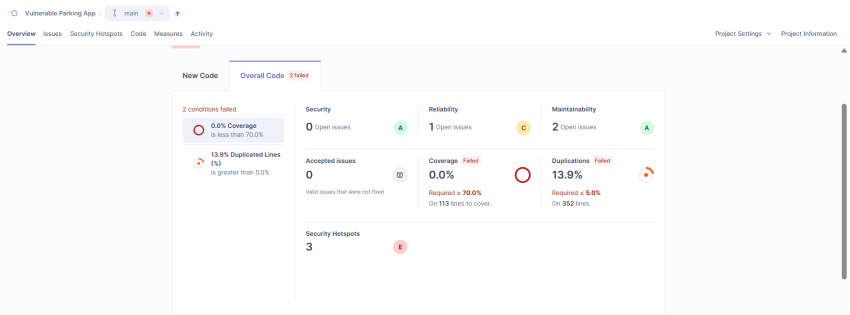

Figura 2 *Resultado del primer escaneo sin archivo de propiedades configurado.*

Posteriormente, con la configuración aplicada, se obtuvo un estado inicial de **FAILED**.

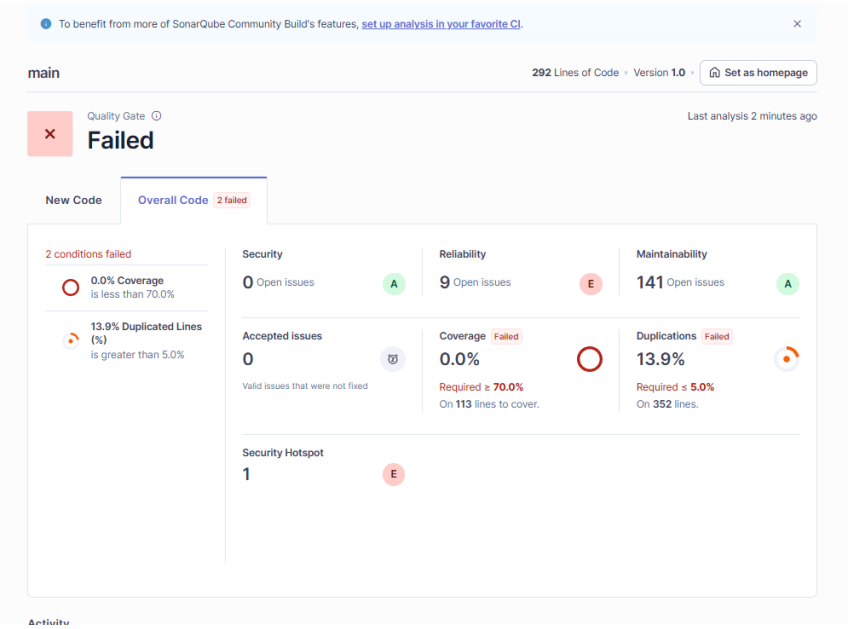

Figura 3 *Vista general de errores encontrados en el primer análisis completo.*

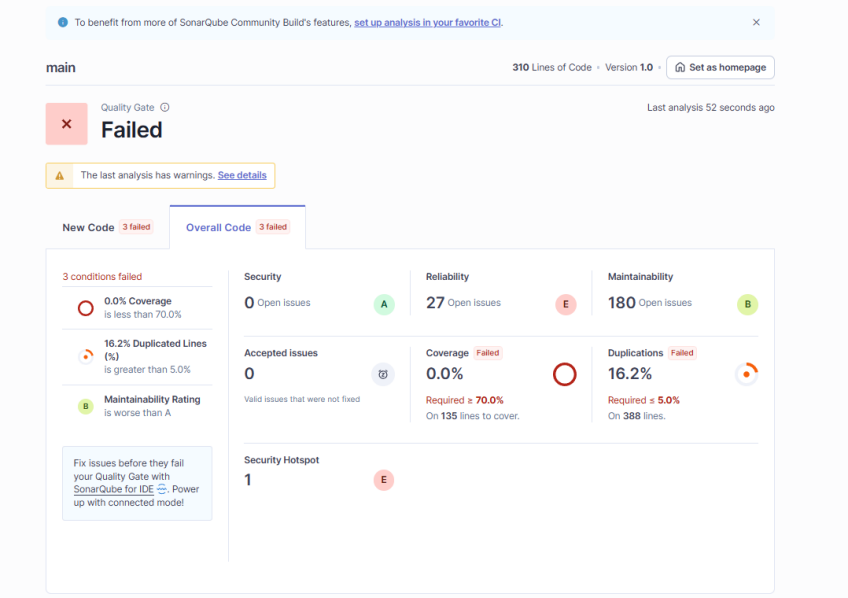

Figura 4 *Detalle del Quality Gate fallido debido a métricas insuficientes.*

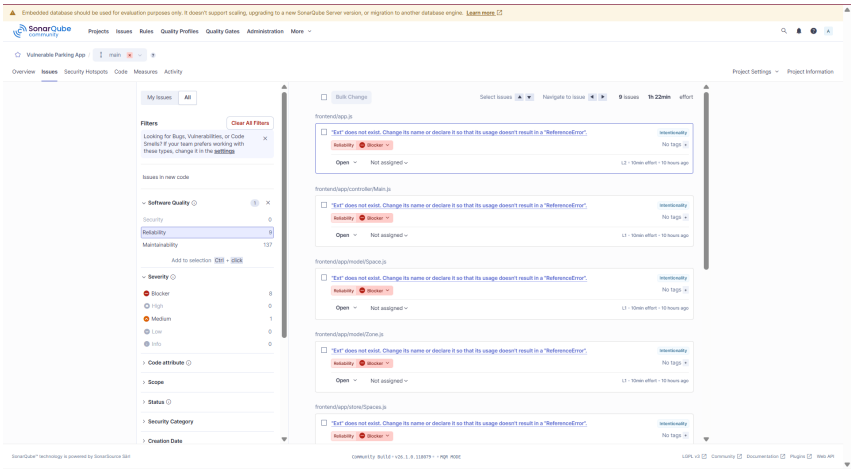

Figura 5 *Errores de fiabilidad (Reliability) detectados inicialmente.*

Issues críticos detectados:

- 500+ Code Smells (principalmente estilo y consistencia).
- Vulnerabilidades de Inyección SQL en rutas de API.
- Falta total de Cobertura de Código (0 %).

4.3 Fase 3: Configuración de Calidad y Refactorización

Para solucionar los problemas, primero se definieron las reglas del juego configurando perfiles de calidad y Quality Gates.

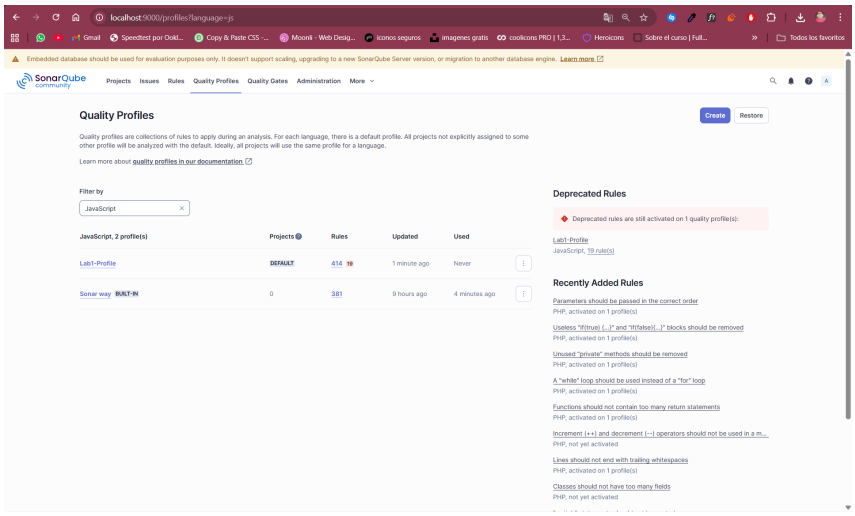

Figura 6 *Configuración de perfiles de calidad (Quality Profiles) para definir las reglas de análisis.*

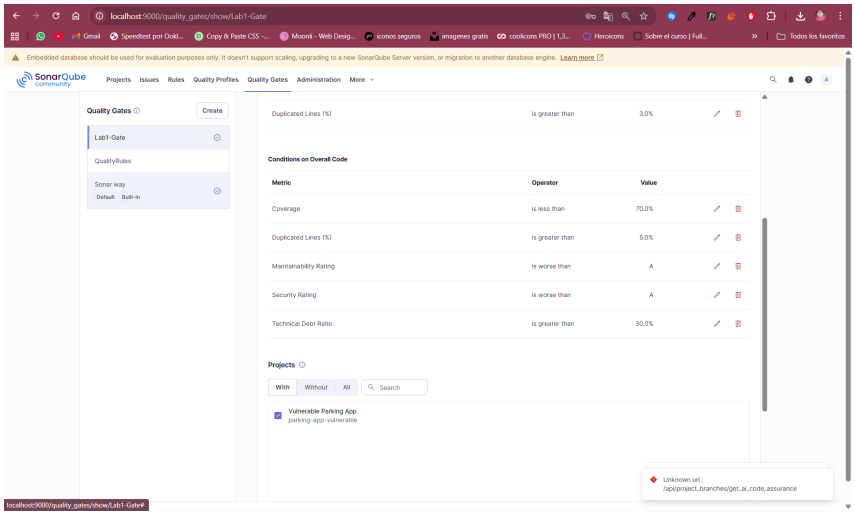

Figura 7 *Establecimiento de Quality Gates personalizados.*

Se procedió a limpiar el código:

4.3.1 Backend

- **Migración a ES Modules**: Cambio de `require` a `import/export`.
- **Seguridad**: Uso de consultas parametrizadas en `db.query` para prevenir SQL Injection.
- **Mantenibilidad**: Creación de constantes para códigos HTTP en `utils/httpStatus.js` y eliminación de números mágicos.

4.3.2 Frontend

- Corrección de variables globales indefinidas (`EXT`).
- Adición de atributos de accesibilidad (`lang`) y seguridad (`integrity`) en HTML.

4.4 Fase 4: Testing y Cobertura

Dado que el Quality Gate exigía un 80 % de cobertura, se implementaron pruebas con **Jest**.

```
// Ejemplo de Test Unitario
it('should return all zones', async () => {
  const mockZones = [{ id: 1, name: 'Zone A' }];
  db.query.mockResolvedValue({ rows: mockZones });
  const res = await request(app).get('/zones');
  expect(res.body).toEqual(mockZones);
});
```

Se configurarion exclusiones en `sonar-project.properties` (`sonar.coverage.exclusions`) para que la métrica se centrara en la lógica de negocio.

Tras las correcciones y la implementación de pruebas, se realizaron nuevos análisis.

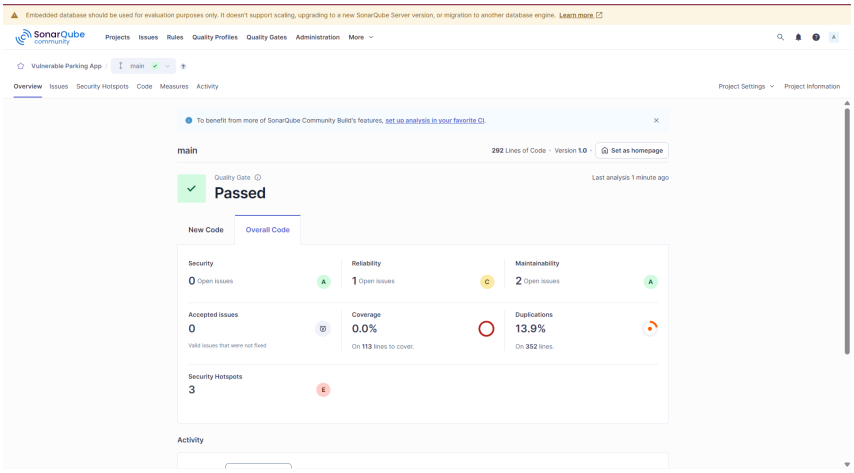

Figura 8 *Primer análisis exitoso tras las correcciones (Passed).*

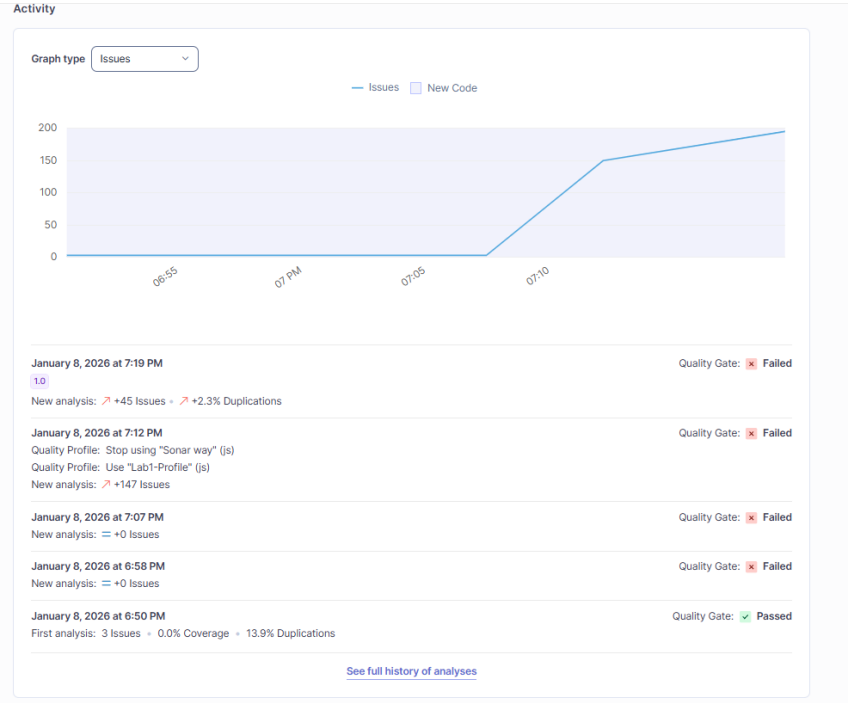

Figura 9 *Historial de análisis mostrando la evolución positiva del proyecto.*

Resultados

5.1 Mejora de Métricas de Calidad

Tras el proceso de refactorización, se logró reducir drásticamente la deuda técnica y eliminar las vulnerabilidades.

- **Vulnerabilidades**: De >5 críticas (SQL Injection) a **0**.
- **Bugs**: Reducidos a **0** (Rating A).
- **Code Smells**: Se resolvieron cientos de problemas de estilo. Los restantes (como trailing commas) se gestionaron mediante reglas de exclusión específicas para evitar conflictos con el linter.

5.2 Cobertura de Código

La cobertura final alcanzó niveles satisfactorios, permitiendo superar el umbral del Quality Gate.

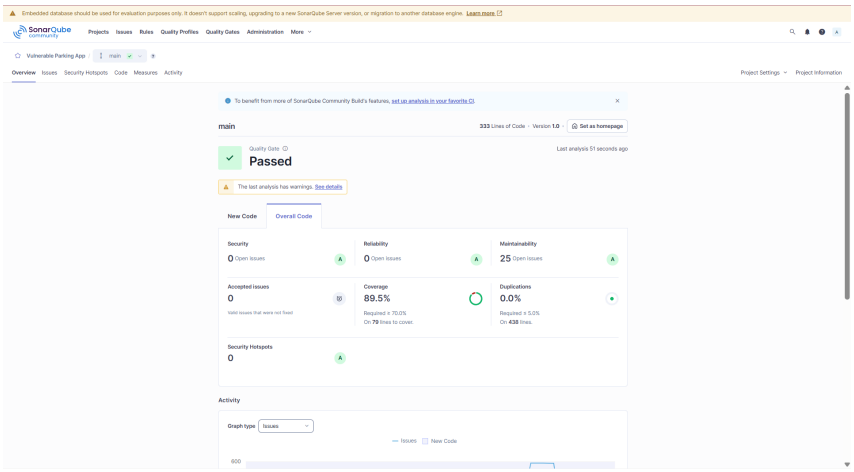

Figura 10 *Reporte de cobertura final. Se observa una cobertura alta en los controladores de Backend (routes).*

Se alcanzaron los siguientes hitos:

- **Routes (Zones & Spaces)**: 100 % de cobertura de líneas y ramas, incluyendo manejo de errores (bloques catch).
- **Exclusiones Correctas**: Archivos como `server.js` (arranque) o carpetas de reportes/frontend fueron excluidos correctamente para no diluir la métrica real.

5.3 Estado Final del Quality Gate

El análisis final reportó un estado de éxito total.

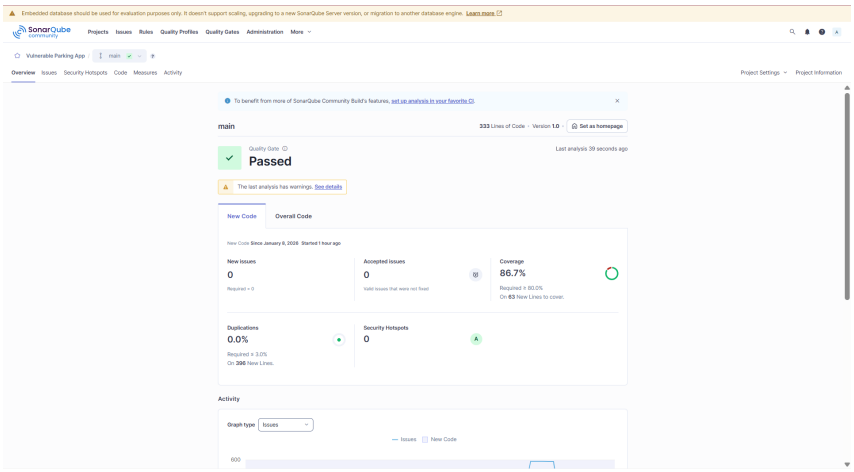

Figura 11 *Dashboard Final de SonarQube mostrando el estado PASSED en todas las métricas (Mantenibilidad, Fiabilidad, Seguridad).*

Discusión

El laboratorio evidenció la tensión existente entre reglas de análisis estático estrictas y la realidad del desarrollo. Inicialmente, nos enfrentamos a un problema de reglas contradictorias ("Flapping rules") donde SonarQube sugería eliminar comas finales (Trailing Commas, regla S1537) en una ejecución, y añadirlas (regla S3723) en otra, dependiendo del perfil de calidad activo. Esto demostró la importancia de **personalizar el perfil de calidad** (Quality Profile) y usar archivos de configuración (`sonar-project.properties`) para ignorar reglas obsoletas o irrelevantes que generan ruido.

Otro punto crucial fue la gestión de la cobertura. Una cobertura del 100 % sobre todo el repositorio es a menudo irreal o ineficiente. Descubrimos que la clave para un Quality Gate útil no es testear todo ciegamente, sino **definir correctamente el alcance** (exclusions). Al excluir el código de "boilerplate"(infraestructura) y enfocarnos en la lógica de negocio crítica, obtuvimos una métrica de confianza real sobre el funcionamiento de la aplicación.

Finalmente, la integración de la seguridad (evitando SQL Injection) mediante análisis estático demostró ser mucho más eficiente que intentar detectar estos fallos en producción, validando el enfoque "Shift-Left.[en] la seguridad.

Conclusiones

La implementación de SonarQube en el ciclo de desarrollo de la "Vulnerable Parking App"transformó un código frágil e inseguro en un producto mantenible y confiable.

1. Se demostró que la automatización del análisis de código es fundamental para detectar vulnerabilidades críticas como SQL Injection que son invisibles a simple vista.

2. La refactorización guiada por métricas (Clean as You Code) no solo mejora la legibilidad, sino que reduce la superficie de ataque y facilita la evolución futura del software.

3. El cumplimiento de un Quality Gate estricto (80 % de cobertura) obliga a adoptar prácticas de desarrollo disciplinadas, como TDD (Test Driven Development) o la escritura de tests post-implementación, lo cual eleva significativamente la confianza en el despliegue.

En conclusión, herramientas como SonarQube son indispensables para cualquier equipo de ingeniería de software que aspire a estándares profesionales de calidad.

Bibliografía

OWASP. (2020). OWASP Software Assurance Maturity Model (SAMM) v2.0. https://samm.dsca.mil/

The Open Group. (2023). Secure Software Development Lifecycle (SSDLC): A Practical Guide to Implement Secure Practices.