Gobernanza y Auditoría de Deuda Técnica

Spring PetClinic

Proyecto: Modernización de Monolito Empresarial

Plazo: Semana 4 – 22 de febrero de 2026

Rol: Staff Software Engineers

1. Descripción General

Este documento cubre la implementación del Pipeline de Gobernanza para el proyecto legado Spring PetClinic, correspondiente a la Entrega 2 del proyecto del curso. El objetivo es establecer compuertas de calidad y mecanismos de gobernanza automatizados que garanticen los estándares de código, midan cobertura, controlen la complejidad ciclomática, e integren un panel de métricas DORA para visibilidad operativa continua.

El repositorio original de Spring PetClinic ya incluía un workflow de GitHub Actions CI. Como parte de esta entrega, ese workflow fue extendido y complementado con herramientas adicionales para cumplir los requisitos de gobernanza. Los componentes implementados fueron:

- Modificación del workflow CI (GitHub Actions — maven-build.yml)
- Medición y enforcement de Cobertura de Código mediante JaCoCo
- Análisis de Complejidad Ciclomática mediante PMD
- Análisis estático y panel de calidad mediante SonarCloud
- Pipeline de Métricas DORA con recolección automática (dora.yml + dora_collector.py)
- Panel de métricas DORA con visualización HTML en tiempo casi real

2. Pipeline CI de Gobernanza

2.1 Configuración del Workflow

El proyecto incluía originalmente un archivo de workflow en GitHub Actions. Dado que Spring PetClinic utiliza Maven como herramienta de construcción, centramos nuestras modificaciones en el workflow maven-build.yml. El workflow de Gradle se conservó pero no fue el foco principal de la gobernanza.

El workflow de Maven fue extendido para integrar el análisis de SonarCloud como parte del ciclo de vida de build-and-verify. El pipeline se activa tanto en eventos push a main como en pull requests hacia main, garantizando que las compuertas de calidad se ejecuten continuamente durante el desarrollo.

2.2 Archivo maven-build.yml

Características principales del workflow final:

- Se activa en eventos push y pull_request contra la rama main
- Corre en ubuntu-latest con Java 17 (via actions/setup-java@v4 con distribución Adopt)

- Utiliza caché de dependencias Maven para optimizar tiempos de build
- Ejecuta ./mvnw -B -U clean verify sonar:sonar para correr el ciclo completo de verificación y análisis estático
- Pasa la clave de proyecto y organización de SonarCloud como parámetros del build
- Lee el SONAR_TOKEN desde los Secrets de GitHub para autenticación con SonarCloud

Comando principal del pipeline:

```
- name: Build and analyze
  run: |
    ./mvnw -B -U clean verify sonar:sonar \
      -Dsonar.projectKey=sandra381_project_enterprise_monolith_2026 \
      -Dsonar.organization=sandra381 \
      -Dsonar.host.url=https://sonarcloud.io
  env:
    SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

3. Cobertura de Código — JaCoCo

3.1 Herramienta y Justificación

JaCoCo (Java Code Coverage Library) fue seleccionado como herramienta de cobertura de código. Se integra nativamente con Maven, genera reportes en formato HTML y XML, y produce datos de cobertura que SonarCloud puede consumir para visualización en el panel. El plugin fue añadido a la configuración del pom.xml.

3.2 Configuración en pom.xml

Se configuraron tres fases de ejecución para el plugin de JaCoCo:

- prepare-agent: Instrumenta la JVM antes de la ejecución de pruebas
- report (fase prepare-package): Genera el reporte de cobertura en HTML y XML
- check (fase verify): Enforza los umbrales mínimos de cobertura — falla el build si no se cumplen

Las compuertas de calidad definidas en la ejecución check de JaCoCo:

Métrica	Umbral Mínimo	Descripción
Cobertura de instrucciones	**≥ 90%**	Cobertura de instrucciones de bytecode
Cobertura de branch	**≥ 84%**	Cobertura de ramas condicionales

Fragmento del pom.xml para la regla check de JaCoCo:

```
<execution>
  <id>check</id>
  <phase>verify</phase>
  <goals><goal>check</goal></goals>
  <configuration>
    <rules>
      <rule>
        <element>BUNDLE</element>
        <limits>
          <limit>
            <counter>INSTRUCTION</counter>
            <value>COVEREDRATIO</value>
            <minimum>0.90</minimum>
          </limit>
          <limit>
            <counter>BRANCH</counter>
            <value>COVEREDRATIO</value>
            <minimum>0.84</minimum>
          </limit>
        </limits>
      </rule>
    </rules>
  </configuration>
</execution>
```

3.3 Resultados de Cobertura

Tras ejecutar el build completo con Maven y JaCoCo habilitado, se obtuvieron los siguientes resultados de cobertura por paquete. Estos resultados fueron usados para calibrar los umbrales mínimos del pipeline:

Paquete	Cob. Instr.	Instr. Faltantes	Cob. Ramas	Ramas Faltantes	Cxty	Líneas	Métodos	Clases
petclinic.owner	**93%**	~	**82%**	5	106	217	66	9
petclinic (raíz)	**7%**	~	n/a	10	4	11	4	2
petclinic.system	**74%**	~	n/a	2	12	16	12	4
petclinic.vet	**100%**	0	**100%**	0	15	34	13	4

petclinic.model	**100%**	0	**100%**	0	15	18	13	3
TOTAL	**90%**	107/1125	**84%**	14/88	**152**	**296**	**108**	**22**

Los resultados muestran una cobertura global de instrucciones del 90% y cobertura de ramas del 84%. Por ello, los umbrales se fijaron exactamente en esos valores: el pipeline garantiza que la cobertura nunca caiga por debajo de la línea base actual. Los paquetes petclinic.vet y petclinic.model alcanzan cobertura perfecta, mientras que petclinic (raíz) y petclinic.system presentan valores más bajos debido a código de configuración y arranque que es difícil de ejercitar en pruebas unitarias.

4. Complejidad Ciclomática — PMD

4.1 Herramienta y Justificación

PMD fue seleccionado para el análisis estático de complejidad ciclomática. Es un analizador de código fuente ampliamente utilizado que soporta detección basada en reglas de code smells y problemas de diseño en proyectos Java. Se añadió el maven-pmd-plugin al pom.xml y se creó un archivo de reglas personalizado en config/pmd/pmd.xml para definir los umbrales de gobernanza.

4.2 Archivo de Reglas PMD (config/pmd/pmd.xml)

Se creó un archivo de ruleset dedicado para definir los umbrales de complejidad ciclomática del proyecto. El ruleset utiliza la regla estándar CyclomaticComplexity de la categoría de diseño de PMD:

```
<!-- config/pmd/pmd.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<ruleset name="PetClinic PMD Rules"
         xmlns="http://pmd.sourceforge.net/ruleset/2.0.0">

  <description>Gobernanza: Umbral de Complejidad Ciclomática.</description>

  <rule ref="category/java/design.xml/CyclomaticComplexity">
    <properties>
      <property name="methodReportLevel" value="20"/>
      <property name="classReportLevel"  value="80"/>
    </properties>
  </rule>

</ruleset>
```

Justificación de los umbrales:

Propiedad	Valor	Significado
methodReportLevel	**20**	Métodos con complejidad > 20 fallan el build. Promueve métodos simples y testables.
classReportLevel	**80**	Clases con complejidad agregada > 80 fallan el build. Identifica clases dios (God Classes).

4.3 Configuración del Plugin en pom.xml

El maven-pmd-plugin versión 3.25.0 fue añadido al pom.xml. Está configurado para fallar el build ante violaciones (failOnViolation: true) e imprimir los errores específicos para trazabilidad. Se ejecuta durante la fase verify:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <version>3.25.0</version>
  <configuration>
    <failOnViolation>true</failOnViolation>
    <printFailingErrors>true</printFailingErrors>
    <rulesets>
      <ruleset>${project.basedir}/config/pmd/pmd.xml</ruleset>
    </rulesets>
  </configuration>
  <executions>
    <execution>
      <id>pmd-check</id>
      <phase>verify</phase>
      <goals><goal>check</goal></goals>
    </execution>
  </executions>
</plugin>
```

5. Integración con SonarCloud

5.1 Configuración y Cuenta

Para la integración con SonarCloud se creó una cuenta en la plataforma y se vinculó con el repositorio de GitHub. Esto permite que en cada análisis del pipeline, SonarCloud reciba los resultados del build y los muestre en un panel centralizado con métricas de calidad, historial de actividad del proyecto y detección de security hotspots.

La propiedad sonar.organization fue configurada en la sección de properties del pom.xml, y el pipeline pasa la sonar.projectKey y sonar.host.url en tiempo de build. La autenticación se maneja mediante el Secret de GitHub SONAR_TOKEN:

```
<!-- pom.xml properties -->
<sonar.organization>sandra381</sonar.organization>
```

5.2 Compuertas de Calidad y Resultados

SonarCloud evalúa automáticamente una Quality Gate en cada análisis. Dicha compuerta verifica bugs nuevos, vulnerabilidades, code smells, cobertura y duplicaciones sobre el código nuevo de cada PR.

Resultados obtenidos en el PR #1 (setup CI e importación del legado), rama delivery2 → main:

Indicador	Resultado	Estado
Quality Gate	**Passed ✓**	Aprobado
Nuevos Issues	21	Sin condiciones definidas
Issues Aceptados	0	Sin issues sin resolver
Cobertura (nuevas líneas)	**91.93%**	Requerido ≥ 80.0% — Cumplido
Duplicaciones	**0.0%**	Requerido ≤ 3.0% — Cumplido
Security Hotspots	**0**	Sin hotspots detectados

6. Panel de Métricas DORA

6.1 Contexto y Objetivo

Las métricas DORA (DevOps Research and Assessment) son el estándar de la industria, respaldado por Google, para medir el rendimiento de los equipos de ingeniería de software. Las cuatro métricas clave evalúan tanto la velocidad de entrega como la estabilidad operativa del equipo:

Métrica	¿Qué mide?
Deployment Frequency (DF)	¿Con qué frecuencia el equipo despliega código a producción? Indica cadencia de entrega.
Lead Time for Changes (LT)	Tiempo promedio desde que un PR es mergeado hasta que llega a producción (deploy). Mide agilidad.
Change Failure Rate (CFR)	Porcentaje de despliegues que causan un incidente en producción. Mide estabilidad.
Mean Time to Recovery (MTTR)	Tiempo promedio para recuperarse de un incidente. Mide resiliencia operativa.

6.2 Arquitectura de la Solución DORA

La solución fue construida completamente dentro del repositorio de GitHub, sin dependencias externas de pago. Se compone de tres partes:

- Workflow de recolección automática (.github/workflows/dora.yml)
- Script de recolección de datos (scripts/dora_collector.py)
- Panel de visualización web (docs/index.html + docs/metrics.json)

6.3 Workflow de Recolección — dora.yml

Se creó el archivo .github/workflows/dora.yml que automatiza la recolección y actualización de métricas. El workflow tiene tres triggers:

- push a main: Se ejecuta automáticamente con cada merge, actualizando las métricas
- workflow_dispatch: Permite ejecución manual desde la interfaz de GitHub Actions
- schedule (cron): Corre todos los lunes a las 06:00 UTC para garantizar datos frescos semanalmente

El flujo del workflow es el siguiente:

```
name: DORA Dashboard

on:
  push:
    branches: [main]
  workflow_dispatch:
  schedule:
    - cron: '0 6 * * 1'   # Cada lunes 06:00 UTC

permissions:
  contents: write         # Necesario para hacer commit de metrics.json

jobs:
  generate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          ref: main
          fetch-depth: 0  # Historia completa necesaria
      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'
      - run: pip install requests
      - name: Generate metrics.json
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

```
        run: python scripts/dora_collector.py
      - name: Commit y push metrics.json
        run: |
          git config user.name 'github-actions[bot]'
          git add docs/metrics.json
          git commit -m 'chore(dora): update metrics' || echo 'No changes'
          git push origin main
```

6.4 Script de Recolección — dora_collector.py

Se creó la carpeta scripts/ con el archivo dora_collector.py. Este script Python consume la API REST de GitHub para calcular las cuatro métricas DORA a partir de eventos reales del repositorio, utilizando una ventana de 90 días hacia atrás:

Métrica DORA	Fuente de Datos	Lógica de Cálculo
Deployment Frequency	GitHub Releases	Releases publicados (no draft, no prerelease) en los últimos 90 días, dividido entre el número de semanas del período.
Lead Time for Changes	PRs mergeados + Releases	Para cada PR mergeado a main, se busca el siguiente Release publicado. El lead time es la diferencia en horas entre el merge y el deploy.
Change Failure Rate	Releases + Issues con label 'incident'	Un deploy se considera fallido si existe un Issue con label 'incident' creado dentro de las 48 horas siguientes al release. CFR = fallos / total deploys.
MTTR	Issues con label 'incident'	Tiempo promedio entre la creación y el cierre de Issues etiquetados como 'incident'. Requiere que los Issues se cierren al resolver el incidente.

El script genera automáticamente el archivo docs/metrics.json con los valores calculados y lo sube al repositorio mediante un commit automatizado del bot de GitHub Actions.

6.5 Panel de Visualización HTML

Se creó la carpeta docs/ con dos archivos:

- metrics.json: Archivo de datos generado automáticamente por el workflow DORA, sirve como fuente de verdad para el panel
- index.html: Panel web interactivo que consume metrics.json y visualiza las cuatro métricas con indicadores de nivel DORA

El panel clasifica cada métrica según los niveles de referencia oficiales de Google/DORA:

Métrica	Elite	High	Medium	Low
Deploy Freq.	Varias/día	1/sem – 1/día	1/mes – 1/sem	< 1/mes
Lead Time	< 1 hora	1h – 1 día	1 día – 1 sem	> 1 semana
CFR	0 – 5%	5 – 10%	10 – 15%	> 15%
MTTR	< 1 hora	< 1 día	< 1 semana	> 1 semana

6.6 Resultados Actuales del Panel DORA

El panel fue generado con una ventana de análisis de los últimos 90 días. Los valores obtenidos son:

Métrica	Valor Obtenido	Clasificación	Análisis
Deployment Frequency	0.23 deploys/semana	**High**	Aproximadamente 1 deploy cada 4-5 semanas, coherente con un proyecto académico recién iniciado.
Lead Time for Changes	0.4 horas	**Elite**	El tiempo entre merge de PR y el siguiente release es inferior a 1 hora, clasificación máxima.
Change Failure Rate	100.0%	**Low**	Valor refleja que todos los releases coincidieron con Issues de incidente en la ventana de 48h. Requiere revisar el etiquetado de issues.
Mean Time to Recovery	0.7 horas	**Elite**	Tiempo de resolución de incidentes inferior a 1 hora, clasificación máxima de resiliencia.

7. Flujo End-to-End del Pipeline

El flujo completo de gobernanza cuando un desarrollador hace push o abre un PR hacia main:

#	Fase Maven	Herramienta	Acción
1	validate	Spring Java Format	Valida estilo de formato de código
2	validate	Checkstyle (nohttp)	Verifica URLs HTTP (no HTTPS) en el código fuente
3	test	Maven Surefire + JaCoCo	Ejecuta pruebas unitarias e instrumenta JVM para cobertura
4	prepare-package	JaCoCo report	Genera reporte de cobertura XML/HTML
5	verify	JaCoCo check	Verifica ≥90% instrucciones, ≥84% ramas — falla si no cumple
6	verify	PMD check	Verifica complejidad ciclomática — falla si supera umbrales
7	post-verify	SonarCloud	Publica resultados al panel de calidad y evalúa Quality Gate
8	(post-push)	dora.yml → dora_collector.py	Recolecta métricas DORA y actualiza metrics.json en el repo

Los pasos 5 y 6 actúan como compuertas de calidad duras: si cualquiera falla, el build se marca como fallido y el PR no puede mergearse. El paso 7 proporciona observabilidad y seguimiento de tendencias. El paso 8 actualiza continuamente el panel DORA con datos reales del repositorio.

8. Auditoría de Deuda Técnica

8.1 Resumen General

Como parte de la modernización de Spring PetClinic, llevamos a cabo un análisis de deuda técnica usando SonarCloud y PMD integrados en el pipeline de CI. El análisis encontró 21 problemas de mantenibilidad distribuidos en distintos niveles de gravedad, con una complejidad ciclomática total de 101 y una complejidad cognitiva de 47.

A partir de estos datos, identificamos los 3 archivos con mayor riesgo dentro del módulo owner, que concentran los problemas más críticos del proyecto. Para cada uno se propone un plan de mejora progresivo usando el patrón Strangler Fig, que permite modernizar el sistema paso a paso sin detener la funcionalidad existente.

8.2 Métricas Generales del Proyecto

Métrica	Valor	Estado
Complejidad Ciclomática Total	**101**	⚠ Alta — recomendado ≤ 10 por método
Complejidad Cognitiva Total	**47**	⚠ Moderada
Total de Issues	**21**	Requiere atención
Issues Blocker	**3**	Crítico
Issues High	**3**	Alto
Issues Medium	**9**	Moderado
Issues Low	**2**	Bajo
Issues Info	**4**	Informativo
Calificación de Mantenibilidad	**A**	Aceptable

8.3 Top 3 Archivos con Mayor Deuda Técnica

Los archivos fueron seleccionados combinando tres criterios objetivos obtenidos directamente de SonarCloud: complejidad ciclomática (cantidad de caminos posibles en el código), complejidad cognitiva (qué tan difícil es entender el código) y problemas reales detectados por el análisis estático.

Hotspot #1 — PetController.java

Riesgo: ALTO | Impacto: ALTO | Prioridad: 1

`Ubicación: src/main/java/org/springframework/samples/petclinic/owner/PetController.java`

Métrica	Valor	Estado
Complejidad Ciclomática	**27**	La más alta del módulo
Complejidad Cognitiva	**15**	La más alta del módulo (empatada)
Problema detectado (L69)	Code Smell – Low	Variable temporal asignada solo para ser retornada de inmediato

¿Por qué es un problema?

Este archivo tiene tantos caminos posibles de ejecución que es muy difícil probarlo bien y es fácil que falle en situaciones poco comunes. En cuanto al problema de la línea 69, lo que pasa es que se guarda un valor en una variable y en la siguiente línea se devuelve sin usarla para nada, como meter algo en una caja solo para sacarlo de inmediato. Es código innecesario que ensucia la lectura.

Plan de mejora con Strangler Fig:

Fase	Sprint	Acciones
Fase 1	Sprint 1	Quitar la variable innecesaria en L69 y devolver el valor directamente. Revisar todos los métodos con complejidad mayor a 5 y anotar cómo funcionan actualmente.
Fase 2	Sprint 2	Crear un archivo PetService que se encargue de la lógica de mascotas. El controller solo recibe la petición y la pasa al service, sin hacer el trabajo él mismo. Meta: complejidad ≤ 10 por método.
Fase 3	Sprint 3	Asegurar que el nuevo PetService tenga al menos 80% de cobertura de pruebas. Verificar en SonarCloud que la complejidad bajó de 27 → ≤ 12 y la cognitiva de 15 → ≤ 8.

Hotspot #2 — Owner.java

Riesgo: ALTO | Impacto: ALTO | Prioridad: 2

`Ubicación: src/main/java/org/springframework/samples/petclinic/owner/Owner.java`

Métrica	Valor	Estado
Complejidad Ciclomática	**22**	Segunda más alta del módulo
Complejidad Cognitiva	**15**	La más alta del módulo (empatada)
Problema detectado (L139)	Code Smell – Medium	Condición if anidada innecesariamente dentro de otro if

¿Por qué es un problema?

En la línea 139 hay dos condiciones if una dentro de la otra cuando en realidad se pueden escribir juntas en una sola línea, lo que hace el código más fácil de leer. Pero el problema más grande es que este archivo intenta hacer demasiadas cosas a la vez: guarda datos en la base de datos, tiene lógica de negocio y maneja la lista de mascotas, todo mezclado en un mismo lugar. Esto hace que cualquier cambio en la base de datos afecte directamente la lógica de negocio, algo que complica mucho el mantenimiento a futuro.

Plan de mejora con Strangler Fig:

Fase	Sprint	Acciones

Fase 1	Sprint 1	Unir las dos condiciones if en L139 en una sola línea. Listar todo lo que hace actualmente esta clase para tener claro qué hay que mover.
Fase 2	Sprint 2	Dividir el archivo en tres partes: una que solo hable con la base de datos (OwnerEntity), otra con la lógica de negocio (OwnerDomain) y una que las conecte (OwnerService).
Fase 3	Sprint 3	Actualizar poco a poco todas las partes del sistema que usaban Owner.java. Verificar en SonarCloud que la complejidad bajó de 22 → ≤ 10 sin romper las pruebas existentes.

Hotspot #3 — OwnerController.java

Riesgo: MEDIO | Impacto: ALTO | Prioridad: 3

`Ubicación: src/main/java/org/springframework/samples/petclinic/owner/OwnerController.java`

Métrica	Valor	Estado
Complejidad Ciclomática	**21**	Tercera más alta del módulo
Complejidad Cognitiva	**7**	Moderada
Problema detectado (L80)	Code Smell – High	El texto literal "error" aparece repetido 3 veces en el código

¿Por qué es un problema?

La palabra "error" está escrita directamente en el código tres veces. Si algún día hay que cambiarla, hay que buscarla y modificarla en tres lugares distintos, con el riesgo de olvidar alguno y que el sistema se comporte de forma inconsistente. Es como tener el mismo número de teléfono anotado en tres papeles distintos: si cambia, hay que actualizar los tres. Además, con una complejidad de 21, el controller hace demasiadas cosas a la vez cuando debería enfocarse solo en recibir y responder peticiones.

Plan de mejora con Strangler Fig:

Fase	Sprint	Acciones
Fase 1	Sprint 1	Definir la constante una sola vez al inicio del archivo: private static final String VIEW_OWNER_FORM = "error" y reemplazar los tres lugares donde aparece el texto repetido. Esto resuelve directamente el issue High de SonarCloud.
Fase 2	Sprint 2	Crear un archivo OwnerService con los métodos: findOwner(), saveOwner() y searchByLastName(). Mover la lógica de búsqueda y guardado al service para que el controller solo reciba y responda peticiones. Meta: complejidad ≤ 8.

Fase 3	Sprint 3	Escribir pruebas unitarias para OwnerService con al menos 80% de cobertura. Confirmar en SonarCloud que la complejidad bajó de 21 → ≤ 8 y que no quedan issues de severidad alta.

8.4 Tabla de Priorización por Riesgo e Impacto

Resumen comparativo de los tres archivos identificados, ordenados por prioridad de atención:

#	Archivo	CC	Cognitiva	Problema (línea)	Severidad	Riesgo	Impacto	Prioridad
1	**PetController.java**	**27**	**15**	Variable temporal — L69	Low	Alto	Alto	**1**
2	**Owner.java**	**22**	**15**	If anidado — L139	Medium	Alto	Alto	**2**
3	**OwnerController.java**	**21**	**7**	Literal repetido — L80	High	Medio	Alto	**3**

8.5 ¿Por qué el patrón Strangler Fig?

El patrón Strangler Fig es la estrategia de mejora más adecuada para este proyecto porque permite modernizar el sistema poco a poco, sin apagar nada ni arriesgarse a romper todo de golpe:

- El sistema sigue funcionando: cada fase saca una parte del código y la mejora mientras el resto continúa operando con normalidad.
- Menos riesgo de romper cosas: los cambios son pequeños y se pueden revisar al final de cada sprint antes de continuar.
- Los resultados son medibles: SonarCloud y PMD muestran de forma concreta si la complejidad bajó después de cada fase.
- Se adapta al equipo: dos personas pueden repartirse las fases y avanzar al mismo tiempo sin bloquearse entre sí.

8.6 Reducción Esperada de Deuda Técnica

Estimación de mejora tras aplicar el plan de refactoring completo en los tres archivos:

Métrica	Antes	Después (estimado)	Mejora
Complejidad Ciclomática Total	**101**	~65	**~35%**
Complejidad Cognitiva Total	**47**	~30	**~36%**
Issues High / Critical	**3**	0	**100%**
Issues Medium	**9**	~3	**~67%**

9. Conclusiones

El pipeline de gobernanza implementado transforma el proceso de build de Spring PetClinic de un flujo simple de compilación y empaquetado en un sistema que cuida la calidad de manera automática. Los resultados clave obtenidos son:

- Cualquier PR que reduzca la cobertura de instrucciones por debajo del 90% o la cobertura de ramas por debajo del 84% fallará automáticamente, evitando retrocesos.
- Cualquier método con complejidad ciclomática superior a 20 o clase superior a 80 hará fallar el build, promoviendo código más simple y fácil de probar.
- SonarCloud ofrece un panel de calidad actualizado con el historial del proyecto, lo que permite al equipo ver la evolución de la deuda técnica con el tiempo.
- El panel DORA da visibilidad continua sobre el rendimiento del equipo, usando datos reales del repositorio de GitHub.
- Las compuertas de calidad y la recolección de métricas DORA corren de forma automática en cada push y PR, sin ningún paso manual.
- El plan de refactoring con Strangler Fig sobre los 3 archivos más críticos reduce la complejidad ciclomática total en aproximadamente un 35%, eliminando por completo los issues de severidad alta.

El código base actual cumple todos los umbrales definidos, lo que establece una buena línea de partida. A medida que el sistema crezca, el pipeline funcionará como un guardián automático de calidad, asegurando que el trabajo de modernización no genere nueva deuda técnica en el camino.