Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

INSPECCIÓN Y MEJORAMIENTO DE LA CALIDAD DE CÓDIGO SEGÚN MÉTRICAS EN SONARQUBE BASADAS EN COBERTURA DE PRUEBAS UNITARIAS, PATRONES Y ESTÁNDARES DE LENGUAJES DE PROGRAMACIÓN IMPLEMENTADOS

Jorge Roberto Godínez Barrios
Asesorado por el Ing. Herman Igor Veliz Linares

Guatemala, julio de 2023

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

INSPECCIÓN Y MEJORAMIENTO DE LA CALIDAD DE CÓDIGO SEGÚN MÉTRICAS EN SONARQUBE BASADAS EN LA COBERTURA DE PRUEBAS UNITARIAS, PATRONES Y ESTÁNDARES DE LENGUAJES DE PROGRAMACIÓN IMPLEMENTADOS

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

JORGE ROBERTO GODÍNEZ BARRIOS
ASESORADO POR EL ING. HERMAN IGOR VELIZ LINARES

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, JULIO DE 2023

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO a.i.	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Kevin Armando Cruz Lorente
VOCAL V	Br. Fernando José Paz González
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANA	Inga. Aurelia Anabela Cordova Estrada
EXAMINADOR	Ing. Álvaro Giovanni Longo Morales
EXAMINADOR	Ing. Alvaro Obrayan Hernández García
EXAMINADOR	Ing. Edgar Estuardo Santos Sutuj
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

INSPECCIÓN Y MEJORAMIENTO DE LA CALIDAD DE CÓDIGO SEGÚN MÉTRICAS EN SONARQUBE BASADAS EN LA COBERTURA DE PRUEBAS UNITARIAS, PATRONES Y ESTÁNDARES DE LENGUAJES DE PROGRAMACIÓN IMPLEMENTADOS

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha 10 de agosto de 2021.

Jorge Roberto Godinez Barrios

Guatemala, 23 de febrero de 2023

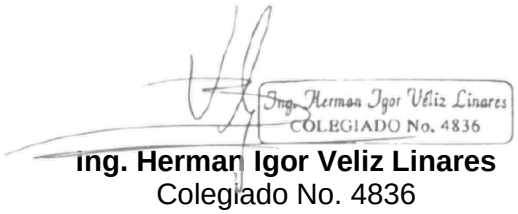
Ingeniero
Carlos Alfredo Azurdia
Coordinador de Privados y Trabajos de Tesis
Escuela de Ingeniería en Ciencias y Sistemas
Facultad de Ingeniería - USAC

Respetable Ingeniero Azurdia:

Por este medio hago de su conocimiento que en mi rol de asesor del trabajo de investigación realizado por el estudiante **JORGE ROBERTO GODÍNEZ BARRIOS** con carné **201020588 y CUI 2313 46654 1202** titulado **"INSPECCIÓN Y MEJORAMIENTO DE LA CALIDAD DE CÓDIGO SEGÚN MÉTRICAS EN SONARQUBE BASADAS EN COBERTURA DE PRUEBAS UNITARIAS, PATRONES Y ESTÁNDARES DE LENGUAJES DE PROGRAMACIÓN IMPLEMENTADOS",** luego de corroborar que el mismo se encuentra finalizado, lo he revisado y doy fé de que el mismo cumple con los objetivos propuestos en el respectivo protocolo, por consiguiente, procedo a la aprobación correspondiente.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,



Ing. Herman Igor Veliz Linares
Colegiado No. 4836

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala 20 de marzo de 2023

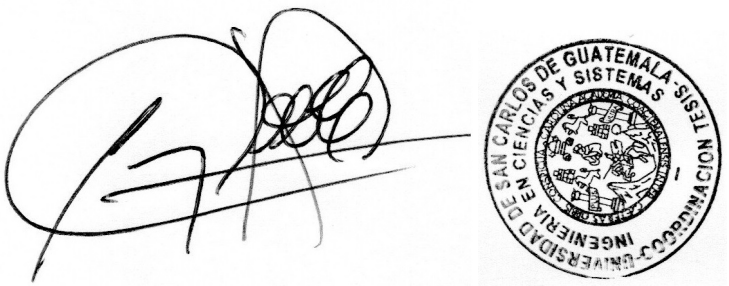
Ingeniero
Carlos Gustavo Alonzo
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Alonzo:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **JORGE ROBERTO GODÍNEZ BARRIOS** con carné **201020588** y CUI **2313 46654 1202** titulado **"INSPECCIÓN Y MEJORAMIENTO DE LA CALIDAD DE CÓDIGO SEGÚN MÉTRICAS EN SONARQUBE BASADAS EN COBERTURA DE PRUEBAS UNITARIAS, PATRONES Y ESTÁNDARES DE LENGUAJES DE PROGRAMACIÓN IMPLEMENTADOS",** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo aprobado.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,



Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

SIST.LNG.DIRECTOR.7.EICCSS.2023

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del Asesor, el visto bueno del Coordinador de área y la aprobación del área de lingüística del trabajo de graduación titulado: **INSPECCION Y MEJORAMIENTO DE LA CALIDAD DE CÓDIGO SEGÚN MÉTRICAS EN SONARQUBE BASADAS EN COBERTURA DE PRUEBAS UNITARIAS, PATRONES Y ESTÁNDARES DE LENGUAJES DE PROGRAMACION IMPLEMENTADOS**, presentado por: **Jorge Roberto Godinez Barrios**, procedo con el Aval del mismo, ya que cumple con los requisitos normados por la Facultad de Ingeniería.

"ID Y ENSEÑAD A TODOS"

Ingeniero Carlos Gustavo Alonzo
DIRECTOR
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, julio de 2023

Decanato
Facultad e Ingeniería

24189101- 24189102

LNG.DECANATO.OIE.45.2023

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería En Ciencias Y Sistemas, al Trabajo de Graduación titulado: **INSPECCION Y MEJORAMIENTO DE LA CALIDAD DE CÓDIGO SEGÚN MÉTRICAS EN SONARQUBE BASADAS EN COBERTURA DE PRUEBAS UNITARIAS, PATRONES Y ESTÁNDARES DE LENGUAJES DE PROGRAMACION IMPLEMENTADOS.**, presentado por: **Jorge Roberto Godinez Barrios** después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

Firmado electrónicamente por: José Francisco Goméz Rivera
Motivo: Orden de impresión
Fecha: 25/07/2023 19:37:52
Lugar: Facultad de Ingeniería, USAC.

Ing. José Francisco Gómez Rivera
Decano a.i.

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANO a.i.
Facultad de Ingeniería

Guatemala, julio de 2023

ACTO QUE DEDICO A:

Dios	Por ser una fuente de sabiduría, vida y salud para lograr cada una de mis metas trazadas.
Mis padres	Jorge Godínez y Elvy Barrios, por su apoyo incondicional y valores que hoy me hacen ser una gran persona.
Mi esposa	Priscila Vásquez, por apoyarme en cada paso y brindarme su amor incondicional.
Mi ángel	José David, por enseñarme amar, vivir, brillar y ser feliz cada minuto como si fuera el último,
Mis hermanas	Michelle y Estefani Godínez, por estar conmigo y apoyarme en todo momento.

AGRADECIMIENTOS A:

Universidad de San Carlos de Guatemala	Por permitir desarrollarme académicamente y ser hoy en día un profesional que puede servir a los demás.
Ing. Herman Veliz	Por ser parte importante de mi desarrollo académico y profesional, brindándome en todo momento su apoyo.
Familia Sandoval Barrios	Por ser mi familia en todo momento y apoyarme incondicionalmente, por ser parte de todos mis logros.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

LISTA DE SÍMBOLOS

Símbolo	Significado
%	Porcentaje

GLOSARIO

Azure	Plataforma en la nube de Microsoft.
Black box	Sistema o componente cuyo funcionamiento interno no es conocido. Los usuarios solo pueden interactuar con él a través de interfaces y observar las entradas y salidas sin conocer los detalles internos de su funcionamiento.
Checkstyle	Herramienta de análisis estático de código para Java.
Ciclomática	Métrica que indica la complejidad de un programa de software.
CI	Integración continua.
Clean as You Code	Práctica de desarrollo de s*oftware* que promueve el mantenimiento y limpieza continuos del código durante el proceso de programación.
CLI	Interfaz de línea de comando.
Dashboard	Tablero de control visual que muestra información relevante de manera intuitiva.
Default	Valor o configuración preestablecida.

Deliver quickly	Entrega rápida y eficiente del desarrollo de *software*.
DevOps	Enfoque que combina el desarrollo de *software* (Dev) y las operaciones (Ops) para mejorar la colaboración y la entrega de *software*.
Docker	Plataforma de contenedores que permite empaquetar, distribuir y ejecutar aplicaciones de forma portátil y aislada.
Estandarización	Proceso de establecer y seguir normas o estándares para garantizar la consistencia y la calidad en el desarrollo de *software*.
IDE	Ambiente de desarrollo integrado.
Interoperabilidad	Capacidad de sistemas o componentes de *software* para comunicarse y trabajar juntos de manera efectiva.
JavaScript	Lenguaje de programación utilizado, principalmente en el desarrollo web para crear interactividad y dinamismo.
Kanban boards	Tableros visuales utilizados en la metodología Kanban para gestionar y visualizar el flujo de trabajo.
Linux	Sistema operativo de código abierto basado en Unix.

Localhost	Nombre de dominio utilizado para hacer referencia a la propia máquina local en una red.
Métrica	Medida cuantitativa utilizada para evaluar y comparar características o desempeño de *software.*
Metodologías	Conjunto de prácticas, técnicas y enfoques utilizados para guiar y gestionar el desarrollo de *software*.
OpenJDK	Versión libre de la plataforma de desarrollo java.
Oracle JRE	Conjunto de utilidades que permite la ejecución de aplicaciones de Java.
Pipeline	Conjunto de pasos o etapas secuenciales que conforman el flujo de trabajo en el desarrollo y despliegue de *software*.
PMD	Detector de errores de programación.
QA	Aseguramiento de calidad.
Quality profile	Determina qué reglas se deben cumplir y cómo se deben evaluar para garantizar la consistencia y mejorar la calidad del código.
Quality gates	Criterios o estándares que deben cumplirse para avanzar en un proceso de desarrollo de *software*.

Repositorio	Lugar donde se almacenan y gestionan los archivos y versiones de un proyecto de *software*.
Riguroso	Caracterizado por seguir procedimientos estrictos y meticulosos.
Server	Servidor, computadora o sistema que provee servicios o recursos a otros dispositivos o usuarios.
Software	Conjunto de programas, instrucciones y datos utilizados por una computadora para realizar tareas específicas.
SonarLint	Herramienta de análisis de código estático que busca y reporta problemas en tiempo real durante la programación.
SonarQube	Plataforma de análisis de calidad de código para detectar y corregir problemas en aplicaciones.
SonnarScanner	Herramienta utilizada para analizar el código fuente y enviar los resultados a SonarQube para su evaluación.
SQALE	Evaluación de la calidad del *software* basada en las expectativas del ciclo de vida.
Testabilidad	Característica de un *software* que permite realizar pruebas de forma efectiva y eficiente.

TI	Tecnologías de la información.
Token	Elemento de identificación utilizado en sistemas de autenticación y seguridad.
Umbral	Valor límite o punto de corte.
URL	Localizador uniforme de recursos.
White box	Enfoque en el cual se tiene acceso y conocimiento completo de los detalles internos de un sistema, en el cual pueden examinar y modificar el código fuente, la arquitectura y los procesos internos del sistema. Valor límite o punto de corte.

RESUMEN

En la fase de desarrollo de *software* es importante aplicar análisis de calidad de código, y el uso de SonarQube para inspección de código estático ayuda a mejorar significativamente la calidad del código en esta fase; siendo compatible con diferentes lenguajes de programación en un ambiente de integración continua y local.

Por lo tanto, para cumplir con este objetivo, definir y configurar de manera adecuada las métricas en SonarQube, se evalúan los resultados y se determina su impacto en la eficacia y mantenibilidad del *software*. Se concluye que la mejora continua de la calidad del código a través de SonarQube puede tener un impacto significativo en la eficacia y mantenibilidad del *software*. Se hacen recomendaciones para establecer un proceso claro para la mejora continua de la calidad del código basado en los reportes brindados durante la ejecución de los análisis realizados con SonarQube.

OBJETIVOS

General

Mejorar la calidad del código mediante la inspección y análisis de código estático utilizando SonarQube para diferentes lenguajes de programación, estableciendo métricas para crear un estándar de calidad de código que debe cumplirse en un ambiente de integración continua, ambiente local y en un entorno de desarrollo integrado.

Específicos

1. Integrar SonarQube con herramientas de integración continua Azure DevOps.

2. Configurar pipeline en Azure DevOps que integre SonarQube y ejecute análisis de código de forma automática mediante solicitudes de cambios realizados a un repositorio.

3. Integrar el análisis estático de código con SonarQube en un ambiente local donde el desarrollador se encuentre codificando.

4. Realizar análisis estático de código en tiempo real, utilizando *SonarLint* integrado a un entorno de desarrollo integrado.

5. Establecer métricas y rangos de valores para evaluar la calidad del código y mejorarla a lo largo del tiempo.

6. Realizar mejoras en el código basándose en los resultados de los análisis y métricas obtenidos en SonarQube.

INTRODUCCIÓN

El desarrollo de *software* es un proceso complejo, muchas veces se encuentra con problemas en cuanto a la calidad del código que puede afectar la seguridad y el desempeño del *software*. Para abordar estos problemas, se han desarrollado herramientas de análisis de código como SonarQube, que proporciona métricas y análisis detallados para mejorar la calidad del código. La integración continua es una práctica que busca integrar y verificar el código de forma constante en todo el ciclo de vida del desarrollo del *software*.

En el primer capítulo se presentan los conceptos principales que ayudarán a conocer la herramienta SonarQube y componentes que la integran, conociendo las partes más importantes al realizar el análisis estático de código de proyectos.

En el segundo capítulo se presenta la implementación de SonarQube en un ambiente de integración continua y los pasos que se deben de llevar a cabo para una correcta implementación, teniendo como resultado mejorar la calidad del código de forma efectiva. Dentro de este mismo capítulo se presenta la implementación de SonarQube en un ambiente local, como el ambiente de desarrollo. Y, por último, se muestra el análisis de código en tiempo real integrando *SonarLint* al entorno de desarrollo integrado.

1. MARCO TEÓRICO

1.1. Modelos y metodologías de desarrollo

Para el desarrollo de *software* no existe un marco y reglas específicas que se cumplan en su totalidad, se utilizan modelos, metodologías y estrategias como guías; y en ciertos casos, utilizar varias fuentes y fusionar un modelo de desarrollo tradicional con alguna metodología ágil, tratando de cumplir lineamientos en ciclo de vida del desarrollo del proyecto con el fin de que cumpla con todos los requerimientos establecidos.

1.1.1. Modelos de desarrollo tradicionales

Sommerville define modelo de proceso de *software* como “Una representación simplificada de un proceso de *software*, representada desde una perspectiva específica. Por su naturaleza los modelos son simplificados, por lo tanto, un modelo de procesos del *software* es una abstracción de un proceso real” (Sommerville, 2005, p. 8).

“Los modelos genéricos no son descripciones definitivas de procesos de *software*; son abstracciones útiles que pueden ser utilizadas para explicar diferentes enfoques del desarrollo de software” (Pons, R. y Pérez, 2010, p. 978-950).

Algunos de los modelos más implementados son los siguientes:

- Prototipo

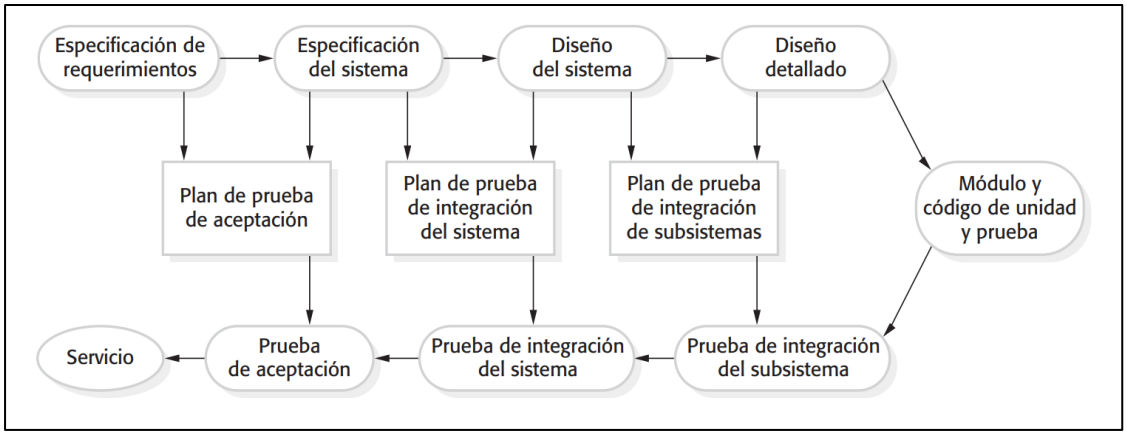
- Desarrollo basado en componentes (reutilización)
- Desarrollo en espiral
- Modelo RAD (*rapid application development*)
- Modelo en cascada

1.1.1.1. Modelo V

Este modelo es muy utilizado en la validación de *software*, verificación y validación (V&V); con este modelo la calidad de software se implementa desde una fase temprana, como la especificación de requerimientos, extendiéndonos hasta la fase final y entrega del *software.*

Con este modelo, se trata de mitigar la mayor cantidad de errores durante todos los procesos y con esto retroalimentar factores para la siguiente fase. A continuación, se presenta en la figura 1 el modelo V.

Figura 1. **Modelo V**



Fuente: Sommerville, I. (2011). *Adaptado de ingeniería de software.* p. 45.

En su estudio sobre el modelo V, (Sommerville, 2011, págs. 42,43) identifica tres etapas cruciales en el proceso de pruebas que merecen una atención especial. Estas etapas se describen a continuación:

1.1.1.1.1. Pruebas de desarrollo

Cada componente se prueba de manera independiente, es decir, sin otros componentes del sistema. Estos pueden ser simples entidades, como funciones o clases de objeto o agrupamientos coherentes de dichas entidades. Por lo general se usan herramientas de automatización de pruebas.

1.1.1.1.2. Pruebas del sistema

Este proceso tiene la finalidad de descubrir errores que resulten de interacciones no anticipadas entre componentes y comprobar que el sistema cumple sus requerimientos funcionales y no funcionales. Para sistemas grandes, esto puede ser un proceso de múltiples etapas, donde los componentes se conjuntan para formar subsistemas que se ponen a prueba de manera individual, antes de que dichos subsistemas se integren para establecer el sistema final.

1.1.1.1.3. Pruebas de aceptación

El sistema se pone a prueba con datos suministrados por el cliente del sistema, en vez de datos de prueba simulados. Las pruebas de aceptación revelan los errores y las omisiones en la definición de requerimientos del sistema, ya que los datos reales ejercitan el sistema en diferentes formas a partir de los datos de prueba.

1.1.2. Metodologías de desarrollo ágiles

(Carvajal Riola, 2008) menciona que la aparición de las metodologías ágiles no puede ser asociada a una única causa, sino a todo un conjunto, bien es cierto que la mayoría de los autores indicarán que son una reacción a las metodologías tradicionales, pero ¿cuáles fueron las causas de esta reacción?

Este autor describe algunas de las causas que impulsaron a la aparición de estas metodologías las cuales son las siguientes:

- *Plumbing*: la traducción al castellano sería pesadez, lentitud de reacción, exceso de documentación, en definitiva, falta de agilidad de los modelos de desarrollo existente.
- Las metodologías existentes no cumplieron las expectativas planteadas inicialmente.
- Explosión de la red y las aplicaciones web.
- Movimiento *open source.*

Jim Highsmith dice que ser *agile* significa ser capaz de *deliver quickly, change quickly, change often*, que significa entregas rápidas, cambios rápidos, cambios frecuentes; que en metodologías ágiles se refiere a la capacidad de entregar de manera rápida y efectiva los resultados en el desarrollo de *software*, destacar la importancia de acelerar los ciclos de entrega, adaptarse rápidamente a los cambios y obtener retroalimentación temprana para maximizar el valor para el cliente. Esto implica adoptar enfoques ágiles y flexibles que prioricen la velocidad y la capacidad de respuesta en el desarrollo y entrega del *software*.

“El factor más importante en el desarrollo de *software* no son las técnicas y las herramientas que emplean los programadores, sino la calidad de los propios programadores” (Glass, 2002, p. 11).

“La calidad de los programadores determinará en un grado muy elevado el éxito del proyecto, parece una frase muy tonta y evidente, pero por alguna extraña razón parece que no todo el mundo la tiene presente” (Carvajal Riola, 2008, pág. 81).

1.1.3. Buenas prácticas de desarrollo de *software*

Fusión de técnicas, principios y metodologías que se implementan durante el desarrollo del software obteniendo resultados que aportan positivamente durante todas las etapas del *software*. Aplicando dichas técnicas estamos entregando un código limpio, reutilizable y escalable, evitando errores mayores e inconveniente en trabajos colaborativos.

Algunas de las buenas prácticas de programación son las siguientes:

- Seguir estándares, como las nomenclaturas y organización de ficheros o estructuras de proyectos.
- documentación de código, evitar comentarios obvios.
- Test obligatorio de código propio.
- Utilizar versionamiento y flujo adecuado.
- Reutilización de código, evitar duplicidad.
- Evitar complejidad o fragmentos de código cortos y concisos.
- Código limpio.

1.1.4. Entornos de desarrollo y despliegue

Existen dos tipos de entornos regularmente utilizados durante el desarrollo de *software.* (EKON, 2020) describe un entorno o ambiente de desarrollo como: Un entorno de desarrollo en el mundo del *software* y la tecnología es un conjunto de procedimientos y herramientas utilizadas por los desarrolladores para codificar, generar, depurar, actualizar, integrar, testear, validar y ejecutar programas. Funciona como un espacio de trabajo en el que los cambios se implementan en diferentes entornos hasta que se ponen en marcha en la versión real (versión del usuario). Actualmente es un entorno de desarrollo al proceso integral de gestión del desarrollo de *software*.

(EKON, 2020) menciona los diferentes tipos de entornos de desarrollo que se ejecutan a medida que va avanzando el proyecto, los cuales son entorno desarrollo, entorno integración, entorno de pruebas (QA), entorno de preproducción y entorno de producción

1.1.5. DevOps

DevOps es una combinación de los términos ingleses *development* (desarrollo) y *operations* (operaciones), designa la unión de personas, procesos y tecnología para ofrecer valor a los clientes de forma constante. DevOps permite que los roles que antes estaban aislados (desarrollo, operaciones de TI, ingeniería de la calidad y seguridad) se coordinen y colaboren para producir productos mejores y más confiables. Al adoptar una cultura de DevOps junto con prácticas y herramientas de DevOps, los equipos adquieren la capacidad de responder mejor a las necesidades de los clientes, aumentar la confianza en las aplicaciones que crean y alcanzar los objetivos empresariales en menos tiempo (Microsoft Azure, 2021).

1.1.5.1. Integración y entregas continuas (CI/CD)

El uso de herramientas de administración de la configuración permite a los equipos distribuir cambios de un modo controlado y sistemático, lo que reduce el riesgo de modificar la configuración del sistema. Los equipos utilizan herramientas de administración de la configuración para hacer un seguimiento del estado del sistema y evitar alteraciones en la configuración, que es como se desvía la configuración de un recurso del sistema a lo largo del tiempo del estado definido para él (Microsoft Azure, 2021).

1.2. Arquitectura de *software*

El diseño arquitectónico es la primera etapa en el proceso de diseño del software. Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y la relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación. (Sommerville, 2011, p. 148)

La arquitectura de *software* es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000).

“En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante” (Sommerville, 2011, p. 149).

1.2.1. Definición de una arquitectura de *software*

El diseño del sistema que cubrirá los requerimientos funcionales y no funcionales de este. Puesto que se trata de un proceso creativo, las actividades dentro del proceso dependen del tipo de sistema que se va a desarrollar, los antecedentes y la experiencia del arquitecto del sistema, así como de los requerimientos específicos del sistema. Por lo tanto, es útil pensar en el diseño arquitectónico como un conjunto de decisiones a tomar en vez de una secuencia de actividades. (Sommerville, 2011, p. 151).

"La arquitectura de un sistema de *software* puede basarse en un patrón o un estilo arquitectónico particular. Un patrón arquitectónico es una descripción de una organización del sistema" (Garlan & Shaw, 1993, pp. 11-39).

1.3. Calidad de *software*

"El grado en que un sistema, componente o proceso cumple con los requisitos especificados; y satisface las necesidades o expectativas del cliente o usuario" (IEEE The Institute of Electrical and Electronics, 1991).

1.3.1. Aseguramiento de calidad de *software*

Un patrón planificado y sistemático de todas las acciones necesarias para proporcionar la confianza adecuada de que un producto se ajusta a los requisitos técnicos establecidos. Basado en un conjunto de actividades diseñadas para evaluar el proceso mediante el cual se desarrollan o fabrican los productos. (IEEE The Institute of Electrical and Electronics, 1991)

1.3.2. Estrategias de pruebas de *software*

- “Realizar pruebas del software en su totalidad, una vez que el paquete completo esté disponible; esta estrategia también es conocida como big bang testing” (Galin, 2003, pág. 182).
- Realizar pruebas del software por partes, en módulos, a medida que se completan (pruebas unitarias); luego, probar grupos de módulos probados integrados con módulos recién completados (pruebas de integración). Este proceso continúa hasta que se hayan probado todos los módulos del paquete. Una vez que se completa esta fase, todo el paquete se prueba como un todo (prueba del sistema). Esta estrategia de prueba generalmente se denomina *incremental testing.* (Galin, 2003, pág. 182)

1.3.3. Clasificación de pruebas de *software*

Hay dos enfoques en los que hay un debate que, si probar la funcionalidad y evaluar los resultados es suficiente, o también la estructura interna del software como lo son los cálculos y matemática subyacente. Con estos dos enfoques se han desarrollado dos tipos de pruebas las cuales son los siguientes

1.3.3.1. *Black box* (funcionalidad)

Identifica errores solo de acuerdo con el mal funcionamiento del software, ya que se revelan en sus salidas erróneas. En los casos en que se determina que los resultados son correctos, las pruebas de caja negra ignoran la ruta interna de los cálculos y el procesamiento realizado. (Galin, 2003, pág. 187)

1.3.3.2. *White box* (estructural)

“Prueba las rutas de cálculo internas para identificar errores. Aunque el término *white* trata de enfatizar el contraste entre este método y las pruebas *black box*. Investigar, conocer y probar la exactitud de la estructura del código” (Galin, 2003, pág. 187).

1.3.3.3. Pruebas de usuario

“Las pruebas de usuario o del cliente son una etapa en el proceso de pruebas donde los usuarios o clientes proporcionan entrada y asesoría sobre las pruebas del sistema” (Sommerville, 2011, pág. 228).

1.3.3.3.1. Pruebas alfa

“Donde los usuarios del *software* trabajan con el equipo de diseño para probar el software en el sitio del desarrollador” (Sommerville, 2011, pág. 228).

1.3.3.3.2. Pruebas beta

“Donde una versión del software se pone a disposición de los usuarios, para permitirles experimentar y descubrir problemas que encuentran con los desarrolladores del sistema” (Sommerville, 2011, pág. 228).

1.3.3.3.3. Pruebas de aceptación

“Donde los clientes prueban un sistema para decidir si está o no listo para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente” (Sommerville, 2011, pág. 228).

1.3.4. Atributos de calidad de *software*

Los requerimientos para un sistema son descripciones de lo que el sistema debe hacer: el servicio que ofrece y las restricciones en su operación. Tales requerimientos reflejan las necesidades de los clientes Al proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se le llama ingeniería de requerimientos (IR). (Sommerville, 2011, p. 83)

1.3.4.1. Requerimientos funcionales

Enunciados acerca de servicios que el sistema debe proveer, de cómo debería reaccionar el sistema a entradas particulares y de cómo debería comportarse el sistema en situaciones específicas. En algunos casos, los requerimientos funcionales también explican lo que no debe hacer el sistema. (Sommerville, 2011, p. 85)

1.3.4.2. Requerimientos no funcionales

“Los requerimientos no funcionales se suelen aplicar al sistema como un todo, más que a características o a servicios individuales del sistema” (Sommerville, 2011, p. 85).

Los requerimientos no funcionales pueden variar según su clasificación y la arquitectura en la que se evalúen, estos serían algunos:

- Exactitud
- Fiabilidad
- Eficiencia
- Integridad
- Usabilidad
- Mantenibilidad
- Flexibilidad
- Testabilidad
- Portabilidad
- Reutilización
- Interoperabilidad

1.4. SonarQube

SonarQube ofrece la capacidad no solo de mostrar el estado de una aplicación, sino también de resaltar problemas recientemente introducidos. Con una *quality gate*, se puede ir inspeccionando y limpiando a medida que se codifica y, por lo tanto, mejorar la calidad del código de forma sistemática (SonarQube SA., 2021). En esta sección haremos una recapitulación resumida de los componentes importantes en SonarQube y cada una de su descripción, información extraída de página oficial de SonarQube (SonarQube SA., 2021).

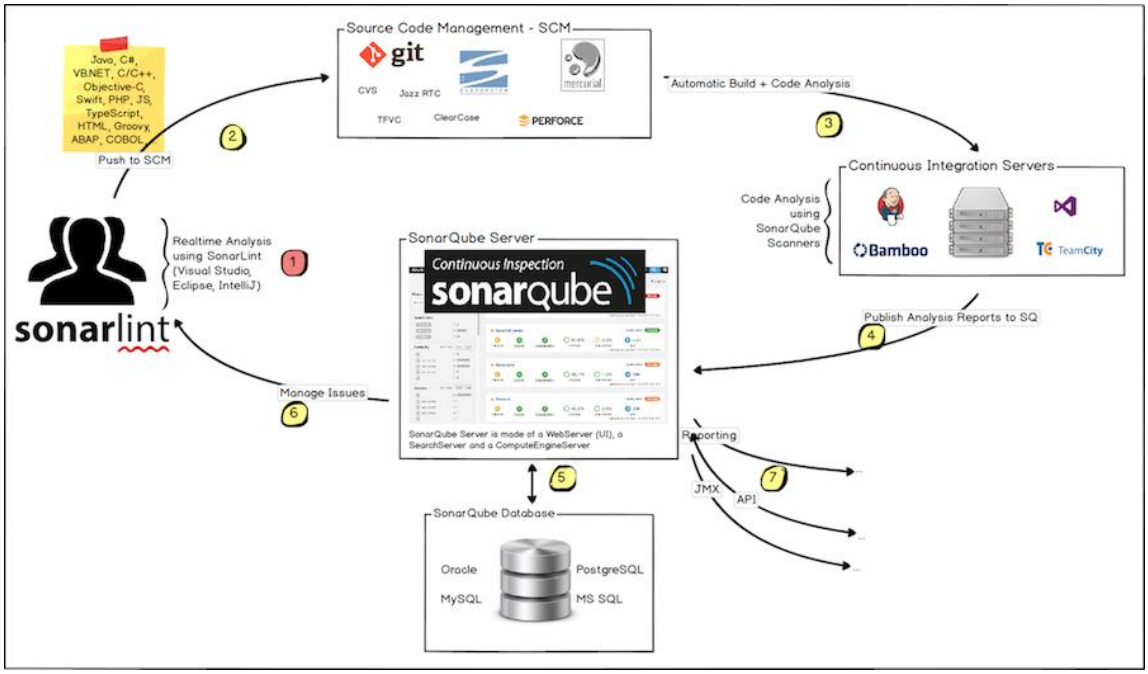
1.4.1. Requerimientos de instalación

El único requisito previo para ejecutar SonarQube es tener Java (Oracle JRE 11 u OpenJDK 11).

1.4.2. Componentes

A continuación, se muestra la figura 2 con la arquitectura de SonarQube.

Figura 2. **Componentes SonarQube**



Fuente: SonarQube (2020*). Adaptado de arquitectura e integración*, SonarQube documentación. Consultado el 2 de agosto 2022. Recuperado de https://sonarqube.inria.fr/sonarqube/documentation/.

1.4.3. Tipos de problemas

Al ejecutar un análisis, SonarQube detecta un problema cada vez que un fragmento de código no cumple con una regla de codificación. El conjunto de reglas de codificación se define a través del perfil de calidad (*quality profile*) asociado para cada proyecto (SonarQube SA., 2021).

En la inspección existen tres tipos de inconvenientes los cuales son: *bug*, vulnerabilidad, *code smell.*

1.4.4. *Quality gate*

Es un conjunto de condiciones que indica si el proyecto está listo para su lanzamiento o no. Tener un mismo *quality gate* y *quality profile* para todos los proyectos de la empresa, garantiza un estándar de calidad dentro de la compañía y también hace más liviano la movilidad de los desarrolladores entre proyectos, aunque esto no siempre se cumple.

1.4.5. *Quality profile*

Son un componente central de SonarQube donde usted define conjuntos de reglas que, cuando se violan, son detectados como problemas en el código.

1.4.6. Métricas en SonarQube

Las métricas en SonarQube son valores numéricos que se utilizan para medir y evaluar la calidad del código en un proyecto de software. En la documentación técnica oficial (SonarQube SA., 2021) y de la herramienta SonarQube se extrae dicha información relevante con respecto a métricas.

1.4.6.1. Complejidad

Es la complejidad ciclomática calculada con base en el número de caminos a través del código. Siempre que el flujo de control de una función se divide, el contador de complejidad se incrementa en uno.

Cada función tiene una complejidad mínima de 1. Este cálculo varía ligeramente según el lenguaje de programación y las palabras reservadas y las funcionalidades de este.

1.4.6.1.1. Complejidad cognitiva

Puede estar influenciada por varios factores, como la estructura del código, la lógica de programación, la arquitectura del *software* y la forma en que se organizan los componentes. Cuanto más complicada sea la estructura y la lógica del código, mayor será la complejidad cognitiva y más difícil será para los desarrolladores comprender y mantener ese código.

1.4.6.2. Duplicidad

- Bloques duplicados
- Archivos duplicados
- Líneas duplicadas
- Líneas duplicadas (%)

1.4.6.3. Problemas de código

- Nuevos problemas
- Problemas con falsos positivos

1.4.6.4. Mantenibilidad

- *Code Smells.*

- Calificación de mantenibilidad: la escala de calificación de mantenibilidad con base en la remediación por medio de la clasificación SQALE.
- Deuda técnica.
- Relación de deuda técnica: relación entre el costo de desarrollar el *software* y el costo de repararlo.

1.4.6.5. Fiabilidad

- Errores
- Errores nuevos
- Esfuerzo de corrección de confiabilidad

1.4.6.6. Seguridad

- Vulnerabilidades
- Calificación de seguridad
- Esfuerzo de corrección de seguridad
- Puntos de acceso de seguridad

1.4.6.7. Pruebas

- Cobertura de condición.
- Aciertos de cobertura de afecciones.
- Condiciones cubiertas por línea.
- Cobertura: es una combinación de cobertura de línea y cobertura de condición. Su objetivo es proporcionar una respuesta aún más precisa a la siguiente pregunta: ¿Cuánto del código fuente ha sido cubierto por las pruebas unitarias?
- Densidad de éxito de la prueba unitaria (%).

1.4.7. Herramientas de análisis

SonarQube utiliza diversas herramientas de análisis estático.

1.4.7.1. *Checkstyle*

Verifica el cumplimiento de las reglas de codificación, algunos de estos son las siguientes:

- Convenciones de nombres
- Encabezados, *imports*
- Espacios en blanco, formateo
- Comentarios
- Buenas prácticas
- Complejidad ciclomática

1.4.7.2. PMD

Es el motor de análisis estático de código que utiliza la herramienta PMD para identificar problemas y violaciones de las mejores prácticas de programación en los proyectos analizados. Identifica los problemas potenciales.

- Posibles errores
- Código muerto
- Código duplicado
- Complejidad ciclomática
- Legibilidad

1.4.7.3. FindBugs

Herramienta que ayuda a la detección de errores.

- Posibles errores
- Defectos de diseño
- Malas prácticas
- Corrección multiproceso (correcto manejo sincronía)
- Vulnerabilidad de código

1.5. *SonarLint*

Es una extensión IDE gratuita y de código abierto que identifica y le ayuda a solucionar problemas de calidad y seguridad mientras codifica.

Como un corrector ortográfico, *SonarLint* analiza las fallas y proporciona comentarios en tiempo real y una guía clara de corrección para entregar un código limpio desde el principio. (SonarLint.org, 2021)

Fácil de usar, no necesita de ninguna configuración: solo debe ser instalado en el IDE que esté trabajando y permite codificar tranquilamente mientras *SonarLint* hace su trabajo. (SonarLint.org, 2021)

1.5.1. Características de *SonarLint*

SonarLint tiene ciertas características que ayudan a corregir errores en tiempo real, reduciendo el tiempo de codificación, así como los procesos de calidad sean más eficientes en tiempo de ejecución.

1.5.1.1. Detección de errores

SonarLint contiene miles de reglas para diferentes lenguajes de programación, estas reglas están basadas en los *quality profiles*; estas reglas permiten detectar errores comunes, errores engañosos y vulnerabilidades conocidas.

1.5.1.2. Retroalimentación instantánea

Una de las ventajas y beneficios que se obtenie con *SonarLint* es que los problemas se detectan y notifican a medida que se codifica, como un corrector ortográfico, con esto obteniendo un *feedback* instantáneo.

1.5.1.3. Cómo solucionar los errores

SonarLint identifica con precisión dónde está el problema y brinda recomendaciones sobre cómo solucionarlo, basadas en buenas prácticas de programación muy efectivas que han sido implementadas en *SonarLint*, por lo que la mayoría de las soluciones son muy efectivas.

2. FASE DE INVESTIGACIÓN E IMPLEMENTACIÓN

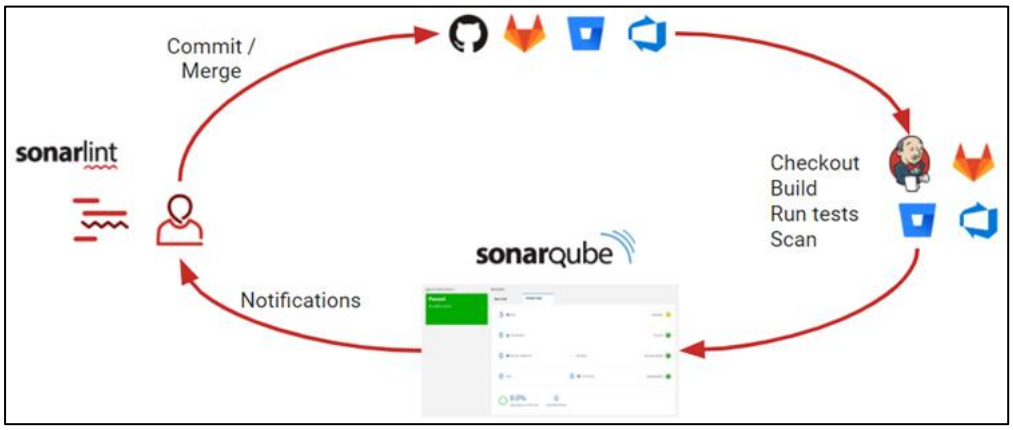
La implementación de dicha herramienta conlleva algunos pasos previos y configuraciones que ayudarán desplegar SonarQube.

2.1. Diagrama SonarQube

La arquitectura de SonarQube para la inspección de calidad de código dependerá del ambiente en el que se vaya a implementar. Si esta será realizada en un ambiente DevOps en el que hará una integración de SonarQube con CI, se necesitarán varios componentes previo, ya que cumpliendo con CI cada cambio que se haga al código y sea integrado a una rama de desarrollo, se ejecutará automáticamente la inspección; y si será en un ambiente local en el cual el desarrollador utiliza para mejoramiento de buenas prácticas de desarrollo y aprendizaje, utilizar únicamente el servidor de SonarQube localmente en el cual se ejecutará manualmente.

Basados en el diagrama mostrado en la figura 3, se observan las partes que componen SonarQube.

Figura 3. **Diagrama SonarQube**



Fuente: SonarQube. (2021). *Adaptado de arquitectura e integración*. SonarQube documentación. Consultado el 2 de agosto de 2022. Recuperado de https://docs.sonarqube.org/8.9/.

2.2. Arquitectura SonarQube

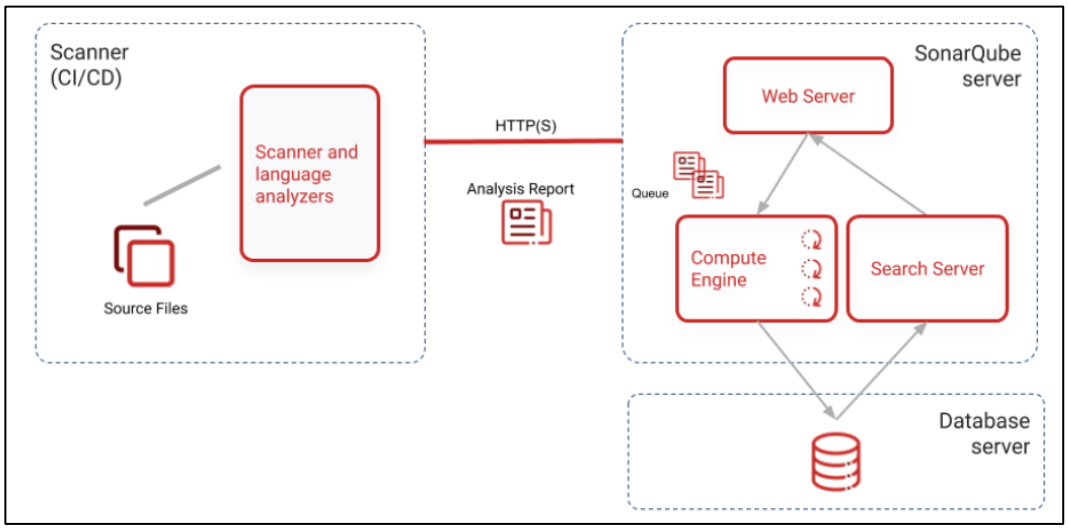
Para llevar a cabo la inspección de código se requiere de tres componentes que se muestran en la figura 4.

- Base de datos: donde se almacenará la configuración, métricas, reportes y problemas que SonarQube pudiera encontrar en el código.

- SonarQube Server: contiene la interfaz gráfica en la cual se mostrará la información almacenada en la base de datos, este servidor también se encarga de realizar el análisis de código por medio a la información brindada por el *scanner* y ejecutar búsquedas en la base de datos.

- Se requiere del *scanner* de SonarQube, se ejecutará de forma manual o automática en el ambiente CI por medio de una tarea Azure DevOps.

Figura 4. **Arquitectura SonarQube**



Fuente: SonarQube S. A., (2021). *Adaptado de arquitectura e integración*. SonarQube documentación. Consultado el 2 de agosto de 2022. Recuperado de https://docs.sonarqube.org/8.9/setup/install-server/.

2.3. Implementación de SonarQube en un ambiente de integración continua

La integración de SonarQube en un ambiente de CI permite automatizar el proceso de análisis, obteniendo retroalimentación del análisis de una forma más rápida y aplicar buenas prácticas de programación.

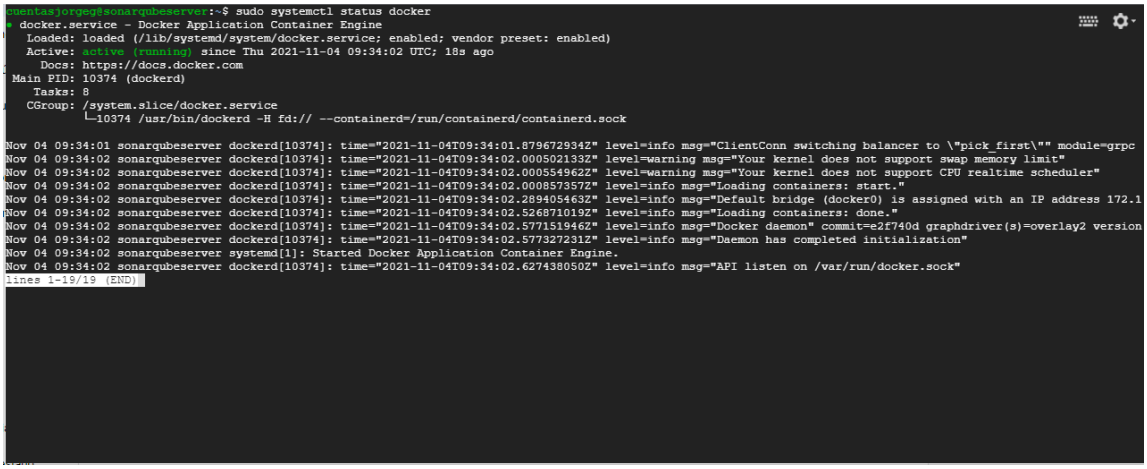
2.3.1. Sonarqube server en *docker*

El servidor en *docker* es importante saber que se ejecuta sin importar la infraestructura con la que se esté utilizando una imagen *docker*.

Por lo que se utiliza una máquina virtual con sistema Linux, para esta implementación emplear la distro Debian 10.

Se conecta a la misma por medio de SSH, luego proceder a la instalación de *docker*. Finalmente, verificar que el servicio *docker* se esté ejecutando, ver figura 5.

Figura 5. **Instalación *docker* en Linux**

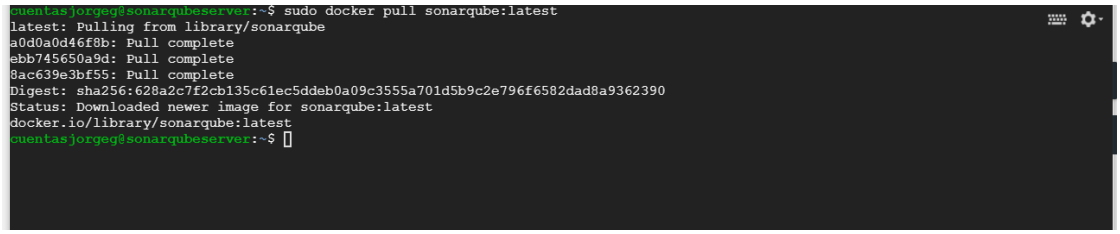


Fuente: elaboración propia, captura de pantalla durante instalación de *docker* en sistema Linux sobre una máquina virtual en la nube.

- Descargar la imagen *docker* oficial para SonarQube, estas imágenes tienen soporte por la comunidad SonarSource para todas las imágenes desde tipo Community hasta Enterprise.

  Estas imágenes están ubicadas en la página oficial de *docker hub* en el repositorio oficial de SonarQube, estas imágenes tienen soporte de SonarSource. Utilizar el *tag latest* para descargar la última versión disponible, ver figura 6.

Figura 6. **Imagen *docker* SonarQube**



Fuente: elaboración propia, captura de pantalla durante descarga de Imagen Docker en sistema Linux desde el repositorio Docker Hub.

- Proceder a instalar la imagen *docker*, esto se realiza con el siguiente comando y luego la instalación.

Figura 7. **Instalación imagen *docker de* SonarQube**

```
docker run -d --name sonarqube -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true -p
9000:9000 sonarqube:latest
```

Fuente: elaboración propia, captura de pantalla durante instalación de *docker* en sistema Linux sobre una máquina virtual en la nube.

La instalación de *SonarQube*, también se realiza por medio de *docker Compose*, en el que se elabora un archivo de extensión YML en donde hay que definir las propiedades del servidor SonarQube, ver figura 8.

Figura 8. **Estructura archivo YML *docker compose* para SonarQube**

```
version: "3"

services:
  sonarqube:
    image: sonarqube:community
    depends_on:
      - db
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    volumes:
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
    ports:
      - "9000:9000"
  db:
    image: postgres:12
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data

volumes:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_logs:
  postgresql:
  postgresql_data:
```

Fuente: elaboración propia, captura de pantalla durante instalación de *docker* en sistema Linux sobre una máquina virtual en la nube.

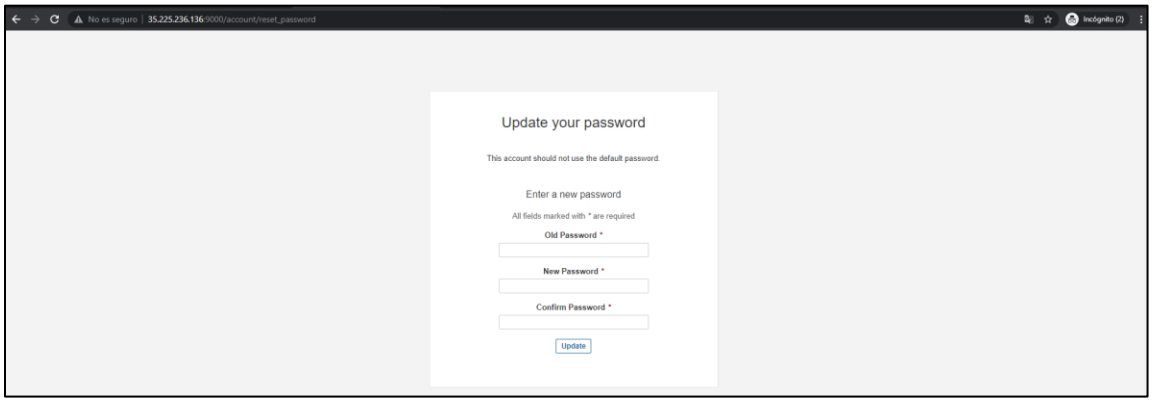
- Verificar que el servidor SoarQube se esté ejecutando, ver figura 9.

Figura 9. **Listar nodos *docker***

Fuente: elaboración propia, captura de pantalla durante instalación de imagen *docker* SonarQube en sistema Linux en la nube.

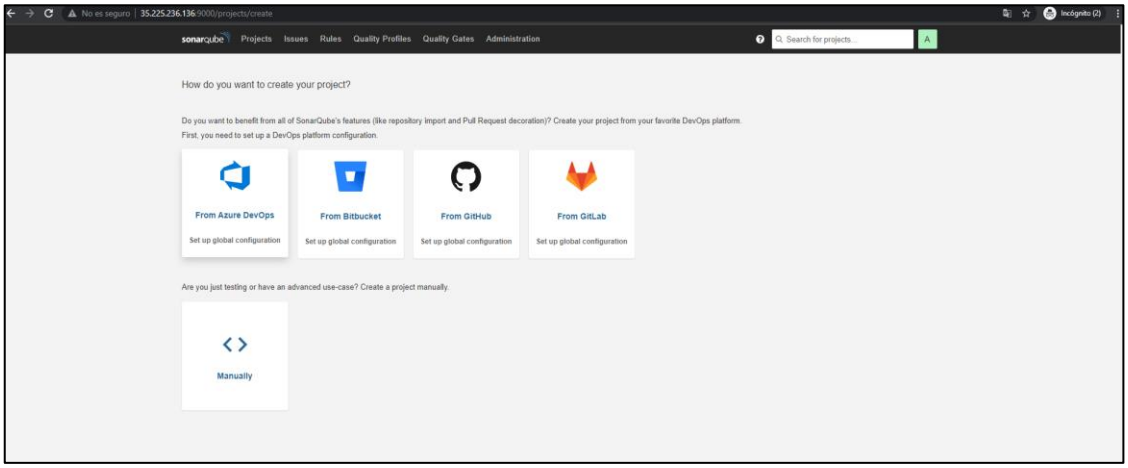
- Ingresar SonarQube con la IP del servidor y puerto configurado utilizando las credenciales por *default,* cambiar credenciales e ingresar al *dashboard* principal, ver figuras 10 y 11.

Figura 10. **Actualización de credenciales SonarQube**



Fuente: elaboración propia, captura de pantalla durante instalación de SonarQube en un servidor en la nube.

Figura 11. **Portal de administración SonarQube**



Fuente: elaboración propia, captura de pantalla durante instalación de SonarQube en un servidor en la nube.

2.3.2. Ambiente de integración continua en Azure DevOps

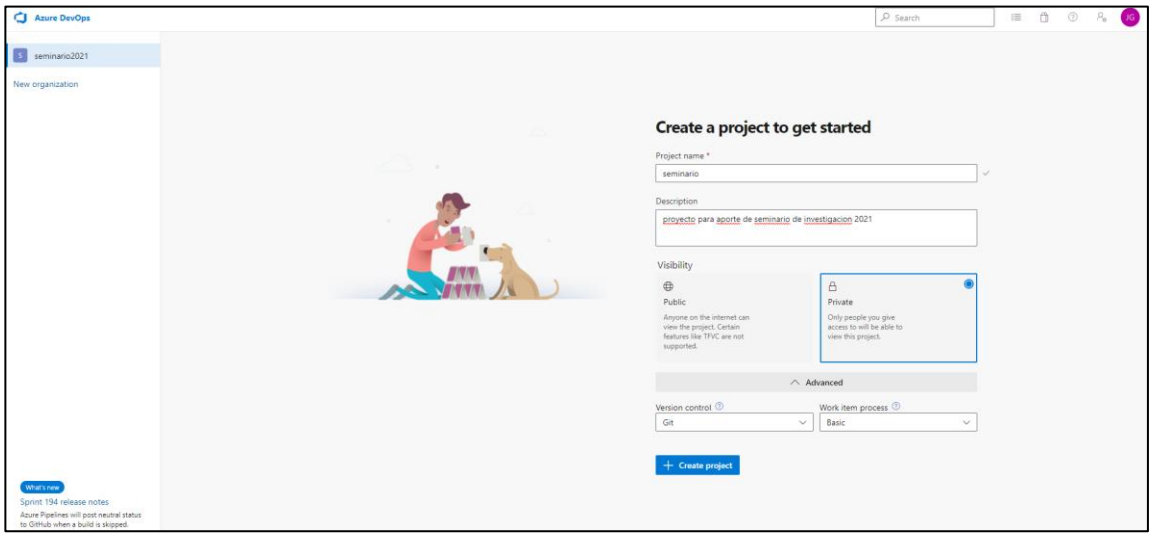
Es importante en la integración con SonarQube porque automatiza el análisis de código, proporciona información integrada de calidad del código, ofrece *feedback* rápido y continuo, mejora la colaboración y se integra con otros flujos de trabajo y herramientas de desarrollo.

2.3.2.1. Creación de organización en Azure DevOps

Se procede a crear una organización en Azure DevOps, y dentro de la organización crear un proyecto, asignar un nombre, en donde se administran los roles, repositorios, *kanban boards*, *pipelines* y artefactos.

Para esta implementación utilizar, principalmente, la organización y proyecto creado para la administración de repositorios *git* y *pipelines* para el ambiente de integración continua.

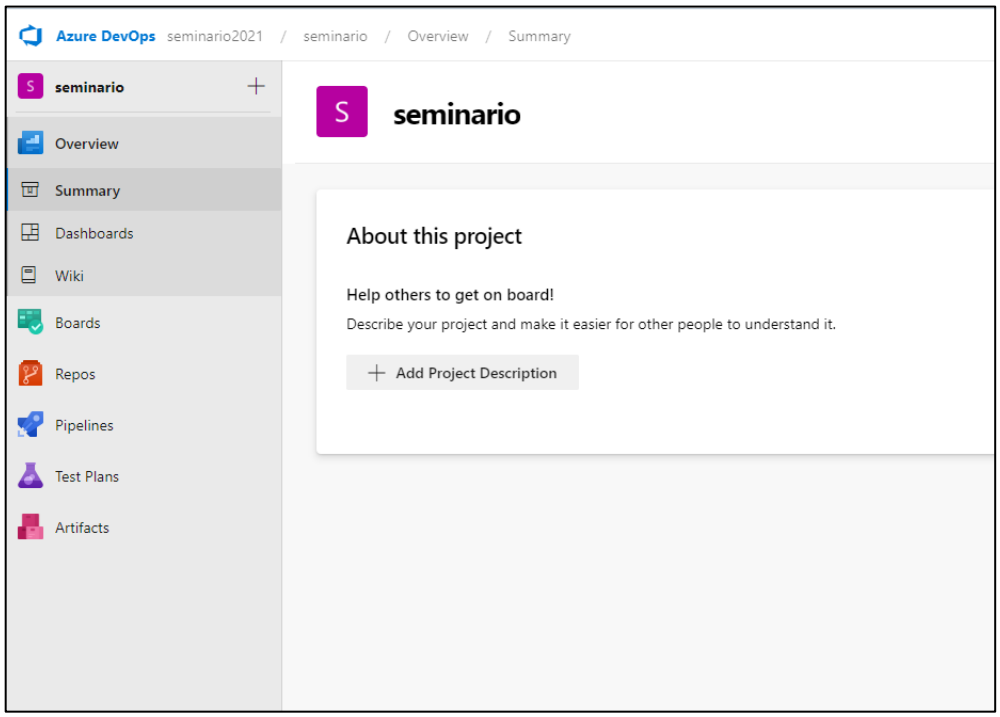
Figura 12. **Organización y proyecto en Azure Devops**



Fuente: elaboración propia, captura de pantalla, crear organización y proyecto SonarQube.

La organización tendrá el siguiente nombre: https://dev.azure.com/seminario2021, aquí será donde se gestionará todo lo relacionado al proyecto.

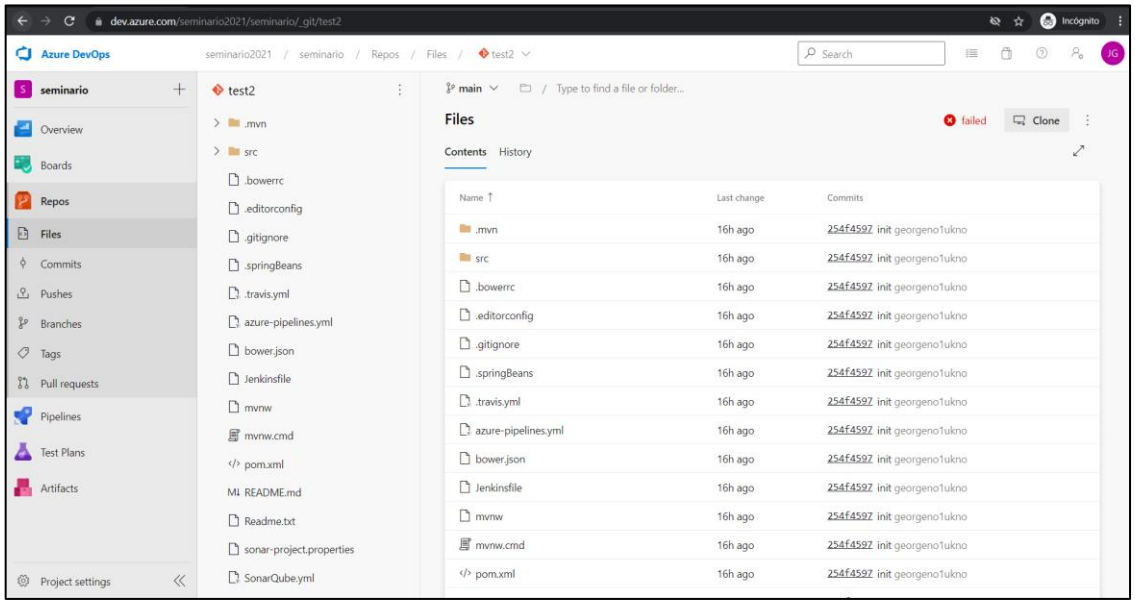
Figura 13. ***Dashboard* principal, proyecto en Azure Devops**



Fuente: elaboración propia, captura de pantalla, crear organización y proyecto SonarQube.

Hay que clonar un repositorio a utilizar para analizar con SonarQube, y para crear el *pipeline* de *build* para el lenguaje. En la figura 14, se muestra el repositorio con el código que se analizará.

Figura 14. **Repositorio para análisis de código**

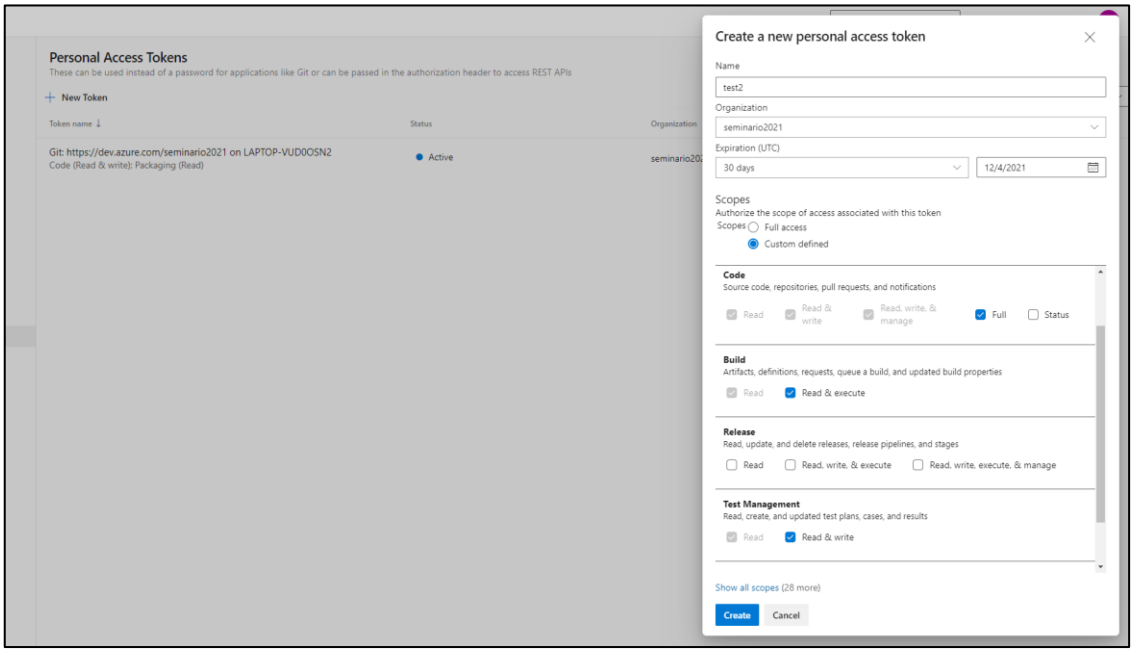


Fuente: elaboración propia, captura de pantalla. Repositorio en Azure.

2.3.2.2. Configuración de *tokens* en Azure Devops

Para que el servidor en SonarQube pueda analizar el repositorio, se necesita gestionar y configurar *tokens* de conexión en la organización de Azure DevOps en la cual se están creando los repositorios y *pipelines*. En la figura 15 se observa la generación del *token*.

Figura 15. **Generación de *token* en Azure DevOps**

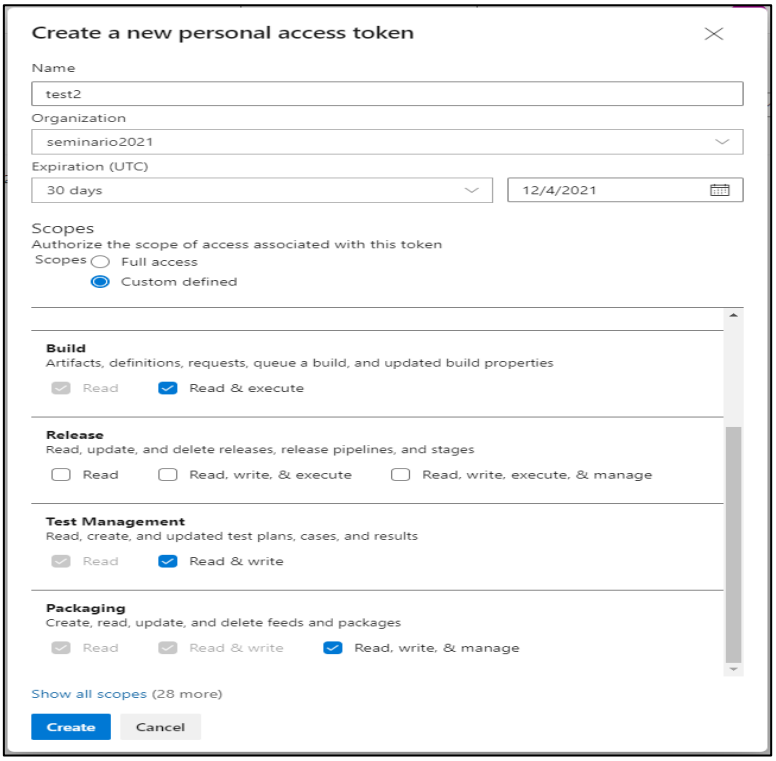


Fuente: elaboración propia, captura de pantalla, personal *tokens* Azure.

Las características, vigencia y permisos que tendrá el servicio que utilizará el *token* serán *code, build, test management y packing.*

En la figura 15, se muestran los tipos de permisos mencionados y su configuración en Azure DevOps.

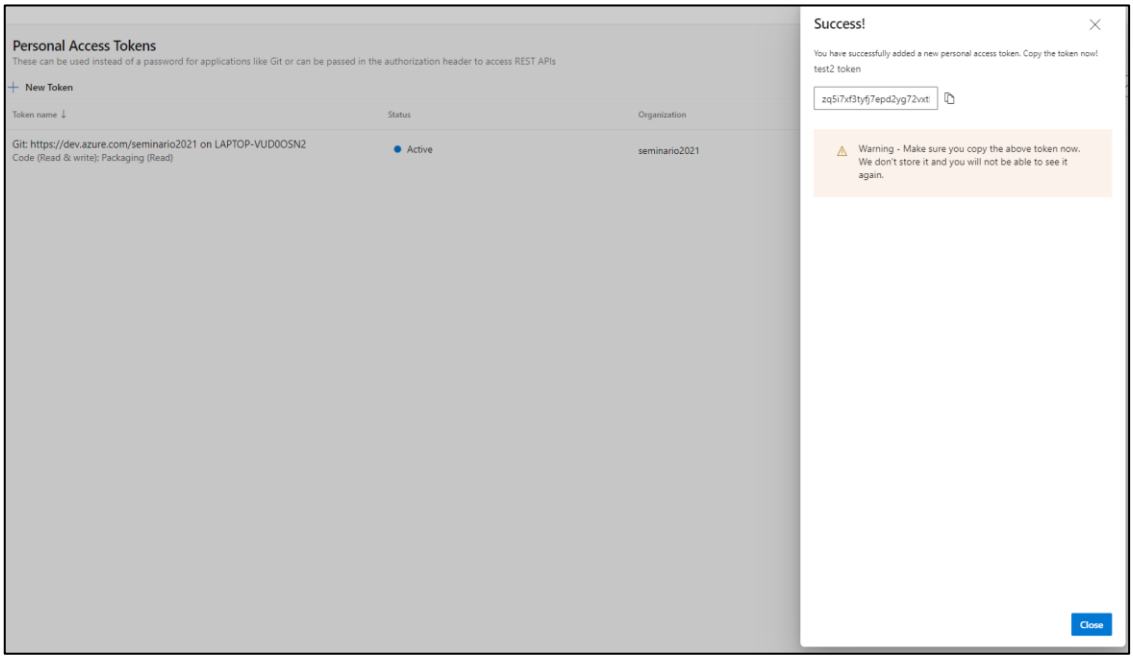
Figura 16. **Tipos de permisos en generación de *token***



Fuente: elaboración propia, captura de pantalla, personal *tokens* Azure.

Se procede a generar el *token* y almacenarlo en un lugar seguro, ya que este será utilizado posteriormente y brindado al servidor de SonarQube para conectarse a Azure DevOps. En la figura 16, se muestra lo mencionado anteriormente.

Figura 17. **Personal *access token***

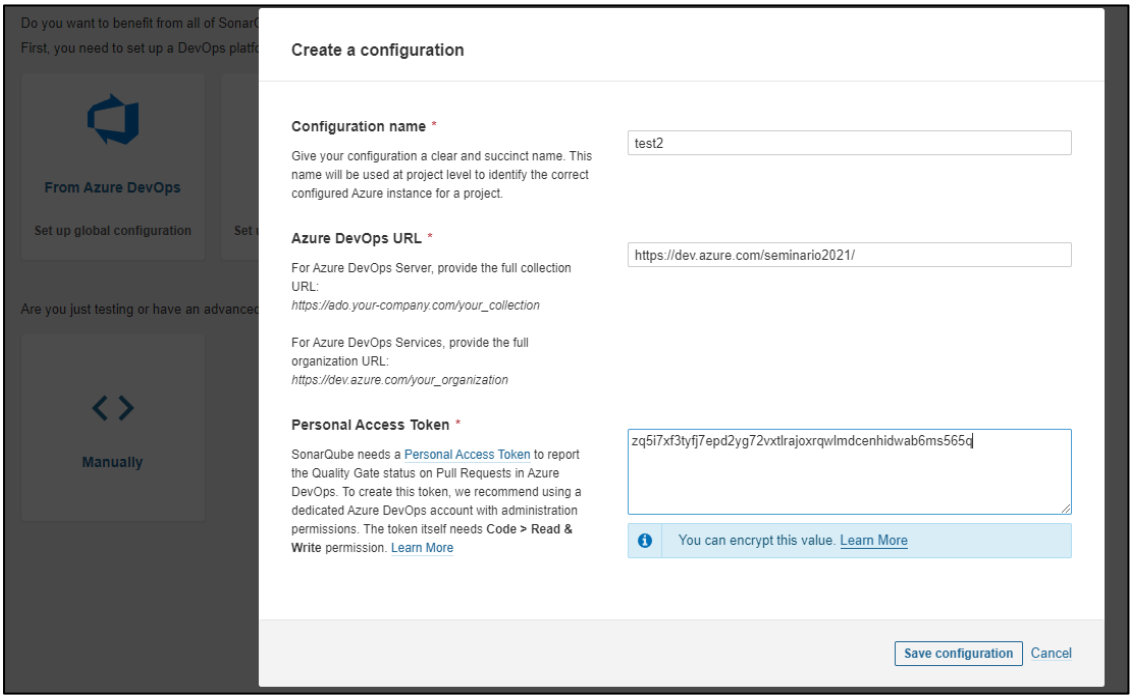


Fuente: elaboración propia, captura de pantalla, generación de personal *tokens* Azure.

2.3.2.3. Configuracion SonarQube

- Utilizar la opción integrar SonarQube con Azure DevOps, ver figura 11
- Proceder a ingresar el nombre del repositorio
- Ingresar la URL de la organización creada en Azure DevOps
- Ingresar el *token* de acceso generado en Azure DevOps, ver figura 18

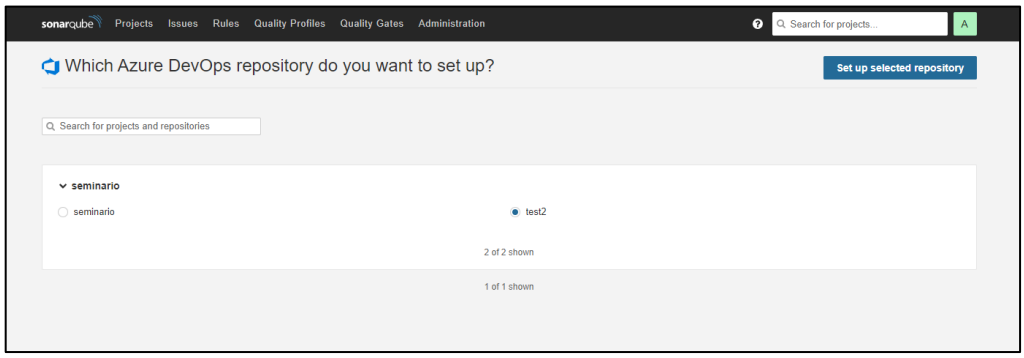
Figura 18. **Integrar Azure Devops a SonarQube**



Fuente: elaboración propia, captura de pantalla durante integración Azure DevOps y SonarQube.

Pon medio del acceso que brinda el *token*, elegir el repositorio a analizar, ver figura 19.

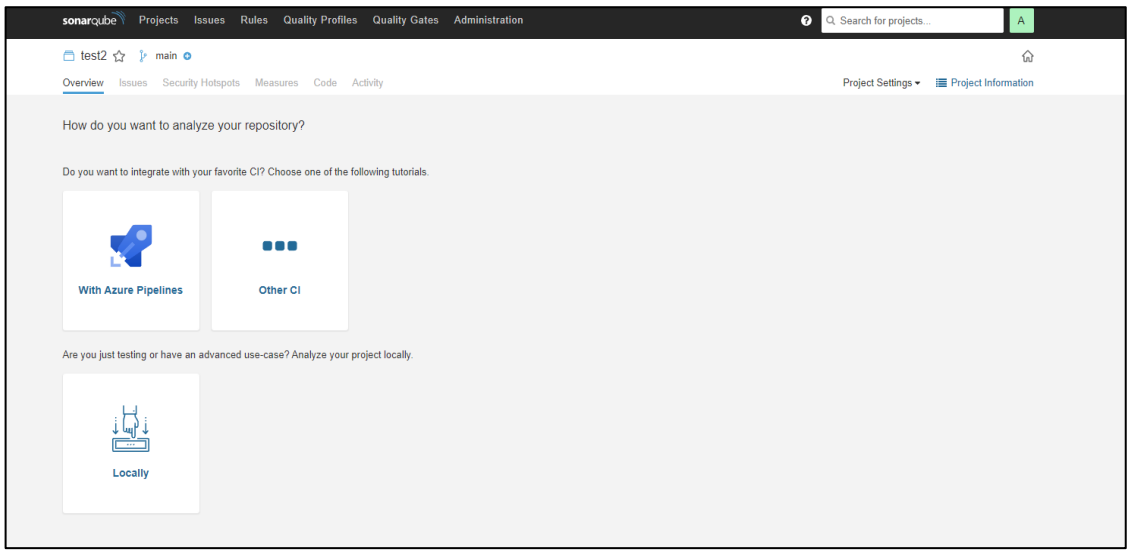
Figura 19. **Vincular repositorio a SonarQube**



Fuente: elaboración propia, captura de pantalla durante integración Azure DevOps y SonarQube

Proceder a elegir el tipo de análisis a realizar, en este caso se hace un análisis de código automatizado, por lo que posterior a vincular el repositorio, que será por medio de un *pipeline* de Azure, ver figura 20.

Figura 20. **Tipo de análisis en SonarQube**

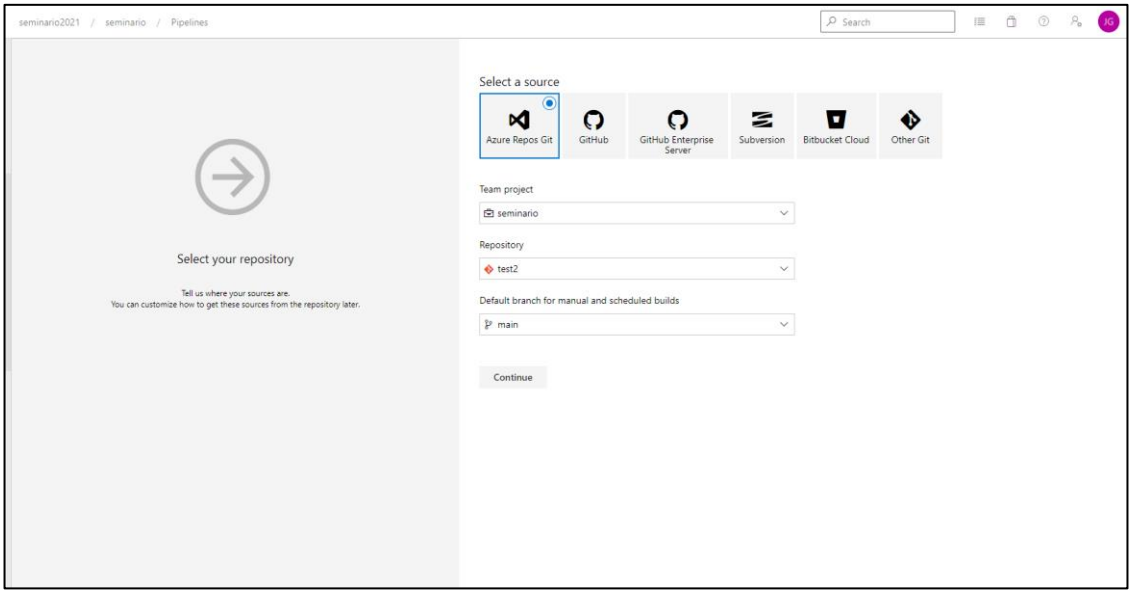


Fuente: elaboración propia, captura de pantalla durante configuración de SonarQube.

2.3.2.4. Creación de pipeline en azure devops

Proceder a la creación del *pipeline* en el proyecto de Azure DevOps, este *pipeline* será vinculado al servidor de SonarQube. Como se muestra en la figura 21, el código a analizar estará gestionado en un repositorio *git* en la cuenta de Azure, elegir el repositorio y la rama de la cual se creará el artefacto.

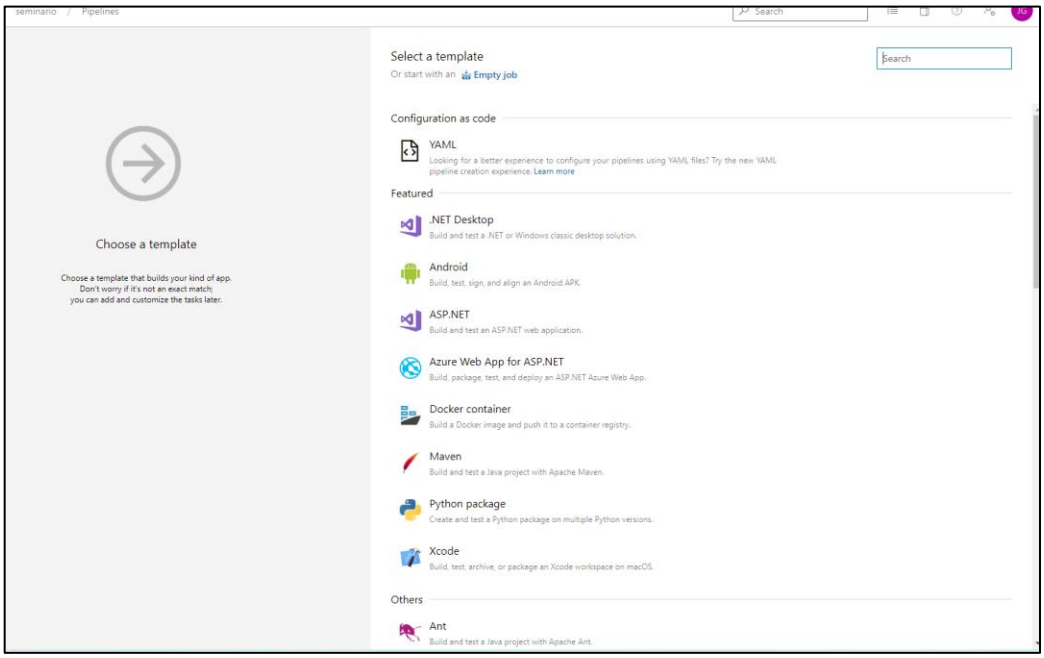
Figura 21. **Fuente del código para *pipeline***



Fuente: elaboración propia, captura de pantalla durante creación *Azure Pipeline*.

Azure *pipelines* ya contiene plantillas por *default*, basadas en el tipo de aplicación o lenguaje; para este ejemplo, se está utilizando un código Java ejecutado en un servidor Maven, este será el punto de referencia para elegir el *template*, como se muestra en la figura 22.

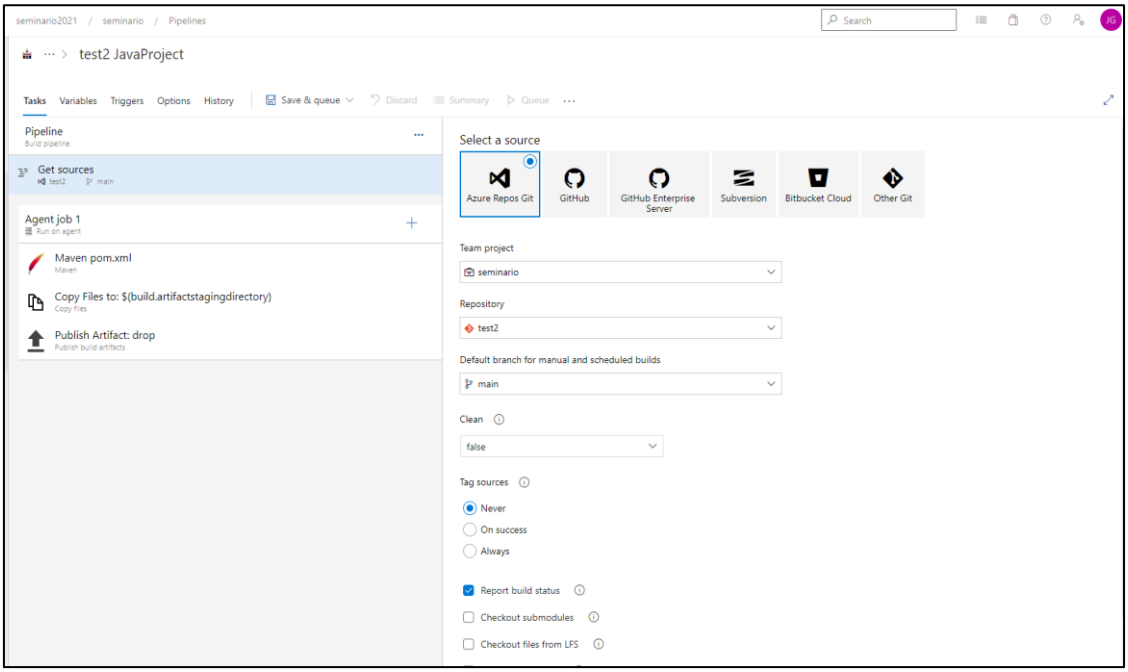
Figura 22. ***Template pipeline***

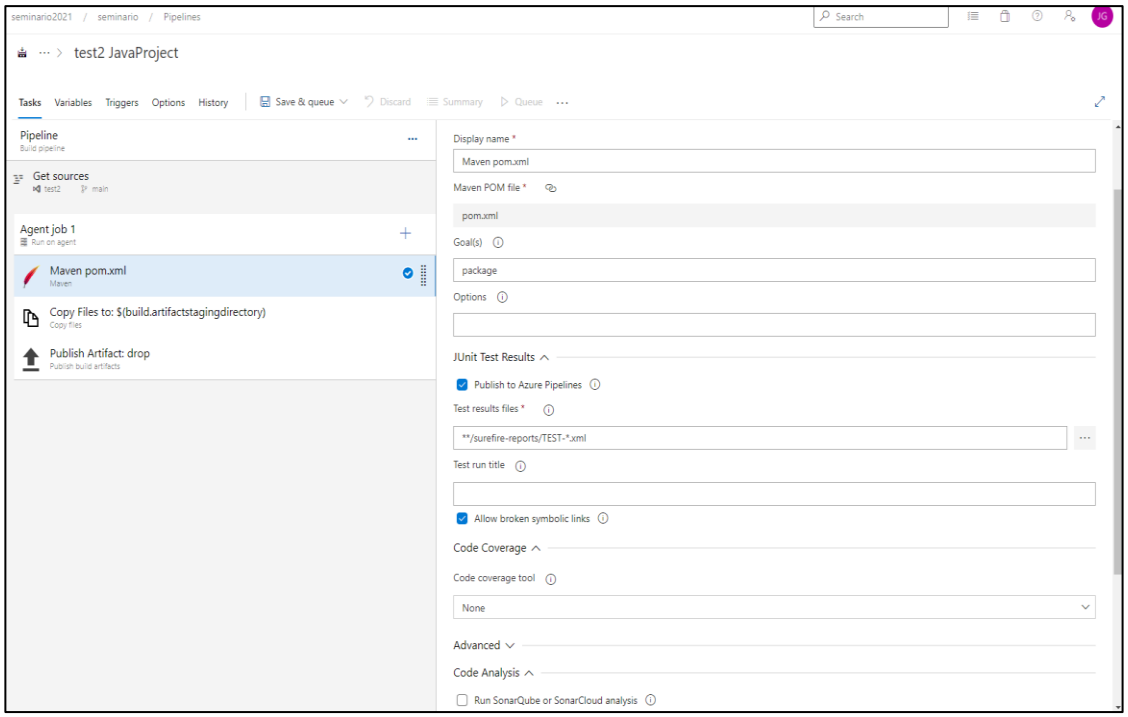


Fuente: elaboración propia, captura de pantalla. *Pipeline* Azure.

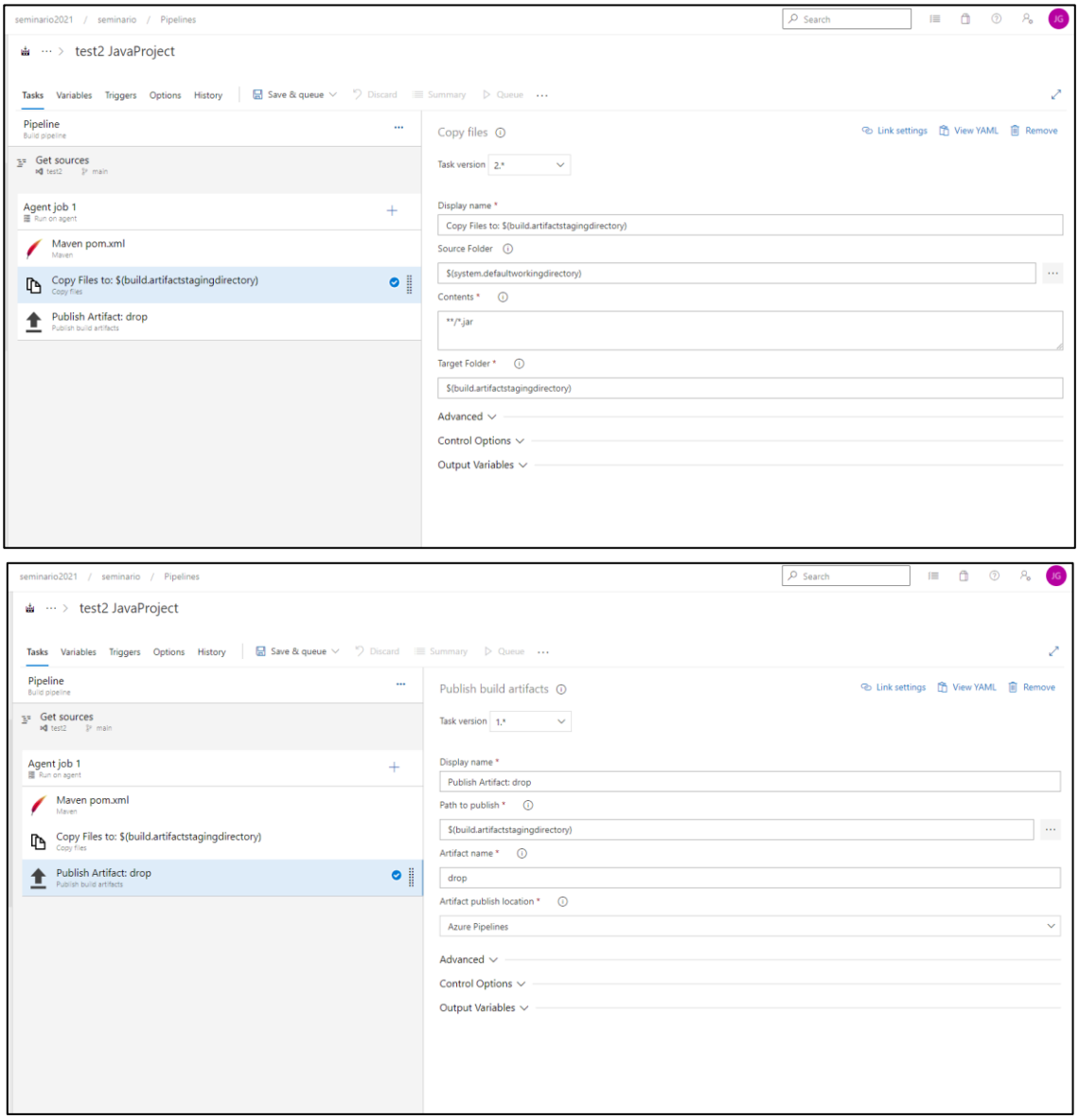
Se procede a configurar los parámetros necesarios para que el mismo pueda ejecutarse y las tareas de compilación, ver de la figura 23 a la 26.

Figura 23. ***Pipeline* Azure Devops**





Continuación de la figura 23.

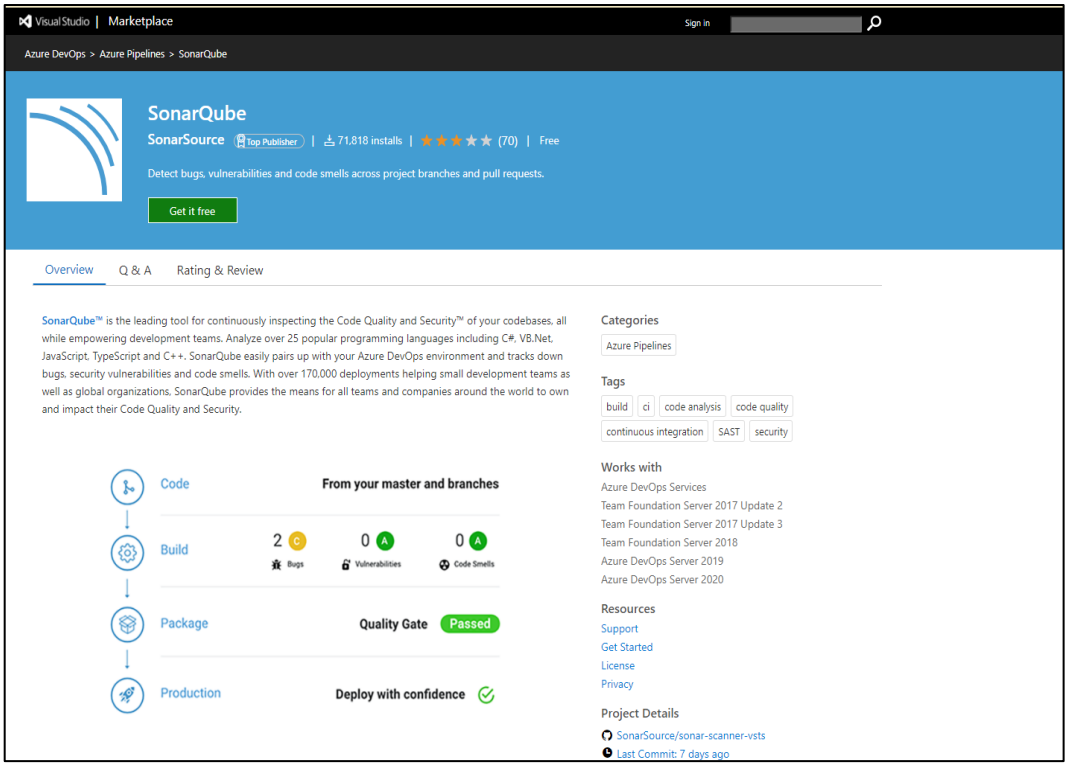


Fuente: elaboración propia, captura de pantalla. *Pipeline* Azure.

2.3.2.5. Instalación SonarQube en organización Azure

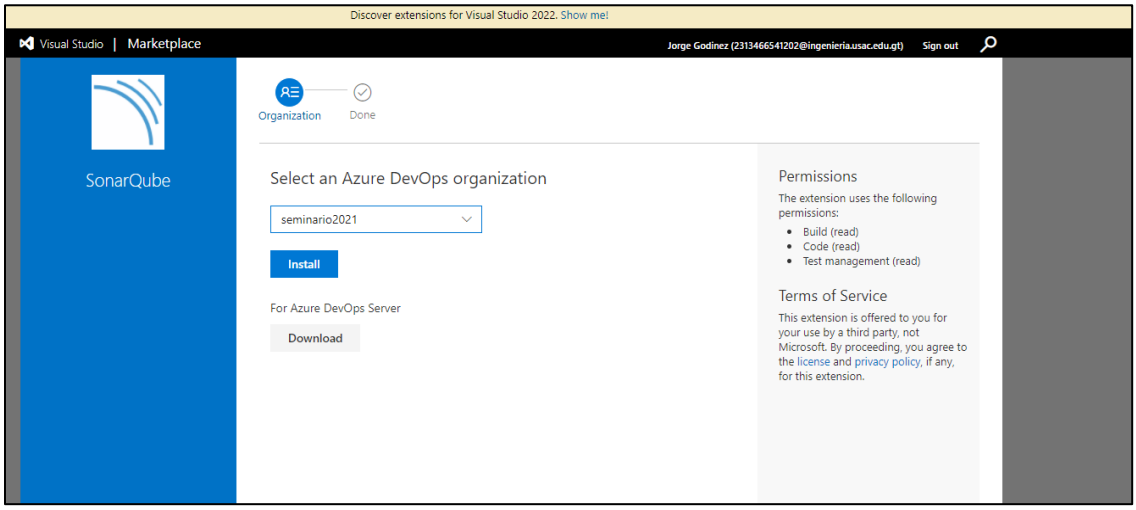
Se procede a instalar SonarQube en la organización de Azure y así crear tareas en *pipelines* referentes a SonarQube para el análisis de código. Ver figuras 24 y 25.

Figura 24. **SonarQube en Azure *marketplace***



Fuente: elaboración propia, captura de pantalla. Instalar SonarQube en Azure organización.

Figura 25. **Instalar SonarQube en organización Azure**



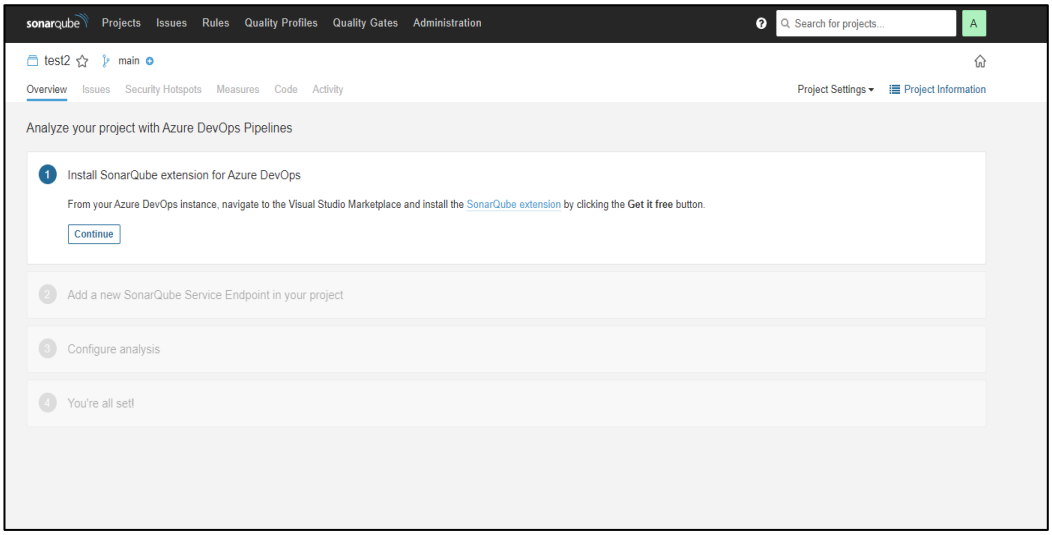
Fuente: elaboración propia, captura de pantalla. SonarQube en Azure Organization.

2.3.2.6. Conectar SonarQube Server a Azure DevOps

El servidor de SonarQube brindará ciertos parámetros que se ingresarán en la cuenta de *Azure* y así vincular completamente de forma bidireccional SonarQube y Azure DevOps.

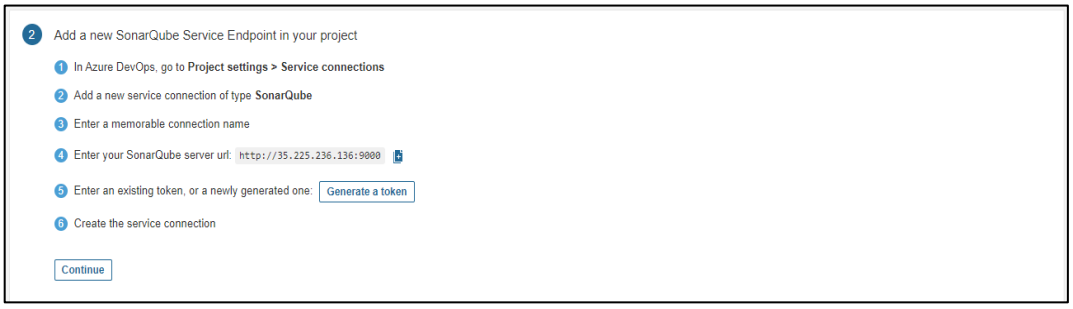
Ingresar al proyecto creado en SonarQube y proceder a la vinculación con los *pipelines* de Azure. SonarQube mostrará los pasos a seguir y los parámetros generados, en las figuras 26 a la 28 se visualiza el proceso.

Figura 26. **Vincular SonarQube en Azure DevOps**



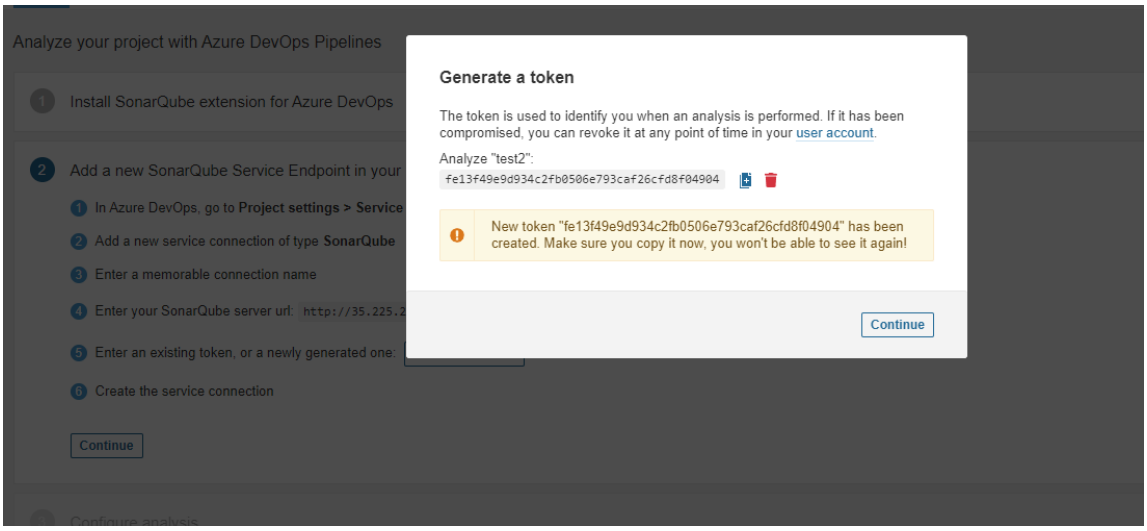
Fuente: elaboración propia, captura de pantalla. Configuración SonarQube.

Figura 27. **Parámetros de configuración de SonarQube**



Fuente: elaboración propia, captura de pantalla. Configuración SonarQube.

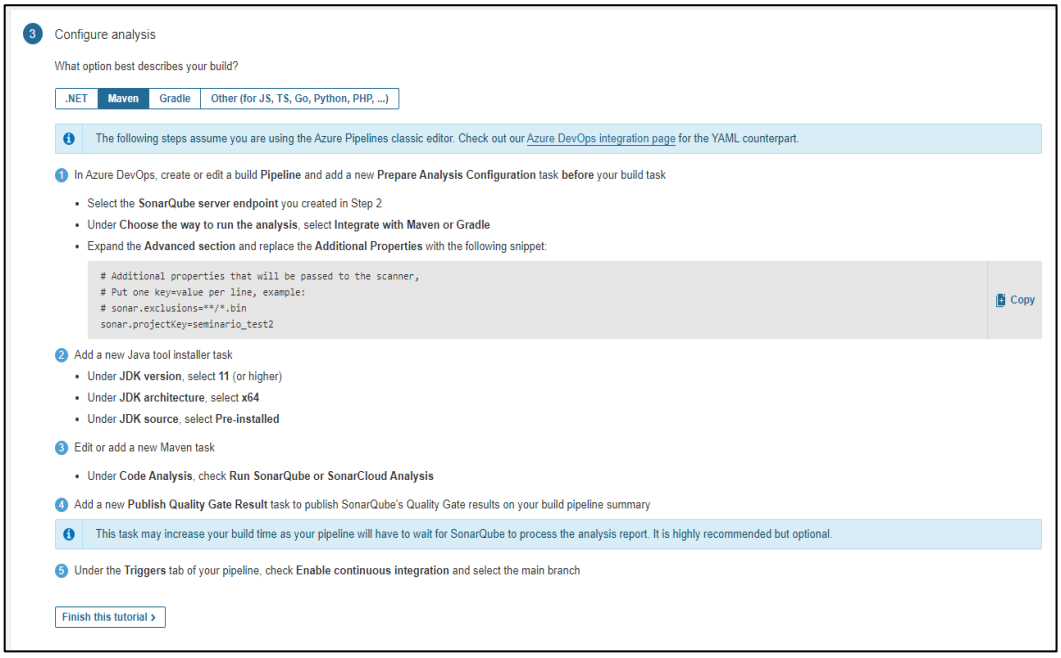
Figura 28. ***Token* generado en SonarQube**



Fuente: elaboración propia, captura de pantalla. Configuración SonarQube.

Configurar el análisis en el proyecto de SonarQube, se brinda el *script* a configurar en Azure *pipeline* y pasos a seguir. Ver figura 29.

Figura 29. **Configuración de análisis en SonarQube**



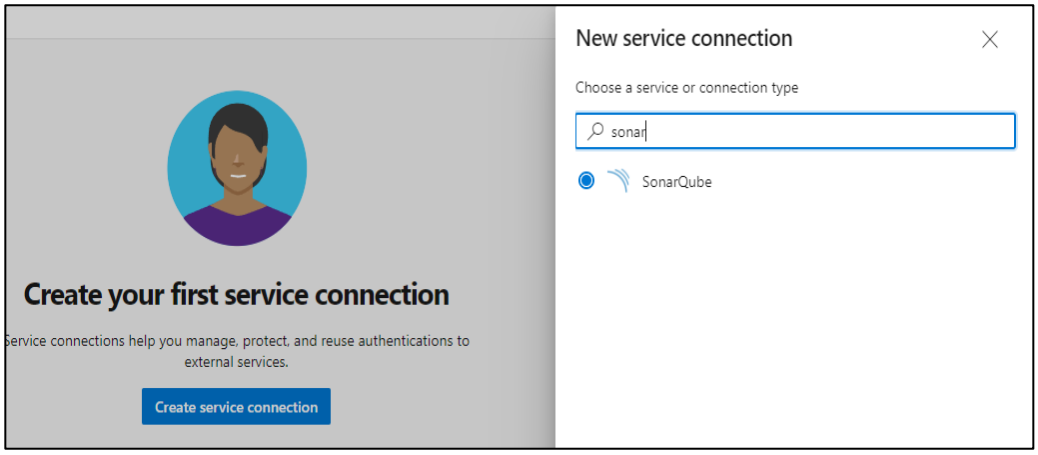
Fuente: elaboración propia, captura de pantalla. Configuración SonarQube.

2.3.2.7. Creación de *service connection* en Azure DevOps

Para que las tareas creadas en Azure DevOps, relacionadas con SonarQube, y estas puedan utilizar la información almacenada en este servidor, es necesario crear un servicio de conexión.

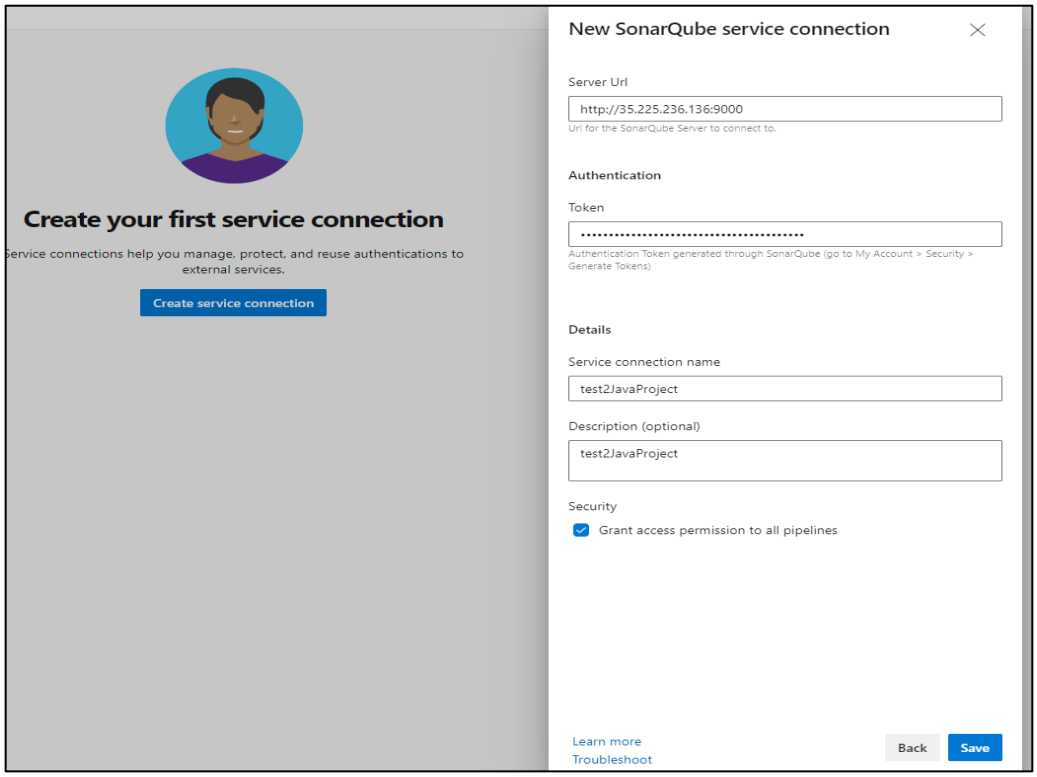
Ingresar a la organización y seleccionar el proyecto en el cual se trabajará, luego ingresar a las configuraciones del proyecto y dirigirse al apartado de *pipelines* donde se ingresará a *Service Connections.* Proceder a la creación del servicio de conexión de SonarQube. Ver figuras 30 y 31.

Figura 30. **Creación de Azure *Service Connection***



Fuente: elaboración propia, captura de pantalla. Azure *Service Connection.*

Figura 31. **Ingreso de parámetros generados en SonarQube**



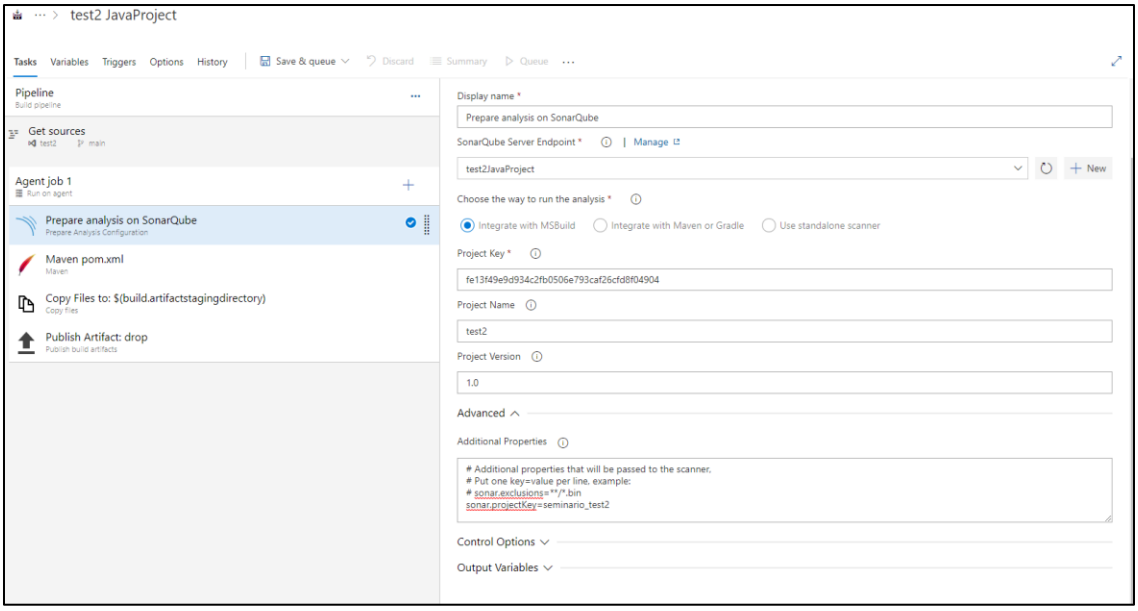
Fuente: elaboración propia, captura de pantalla. Azure *Service Connection.*

2.3.2.8. Creación de tareas SonarQube en *pipeline*

Agregar las tareas de SonarQube al *pipeline*, las cuales se describen en la figura 32, según el lenguaje utilizado.

Crear una tarea previa a las existentes, la cual prepara el análisis del código que se encuentra en el repositorio, ver figura 32.

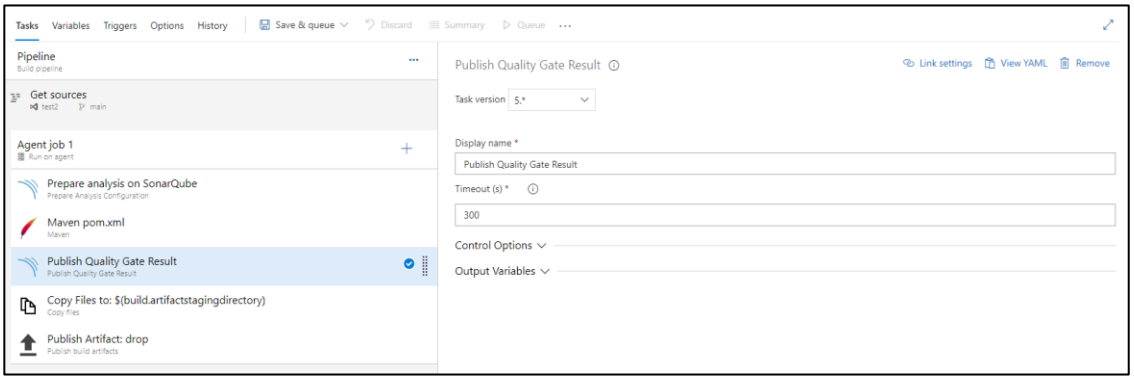
Figura 32. **Tarea *prepare analysis* SonarQube**



Fuente: elaboración propia, captura de pantalla. SonarQube *prepare analysis.*

Se procede agregar la tarea para la publicación de los resultados del análisis del código y con esto, al momento de la publicación del artefacto, visualizar las métricas de dicho análisis. Ver figura 33.

Figura 33. **Tarea en *pipeline publish quality gate result***

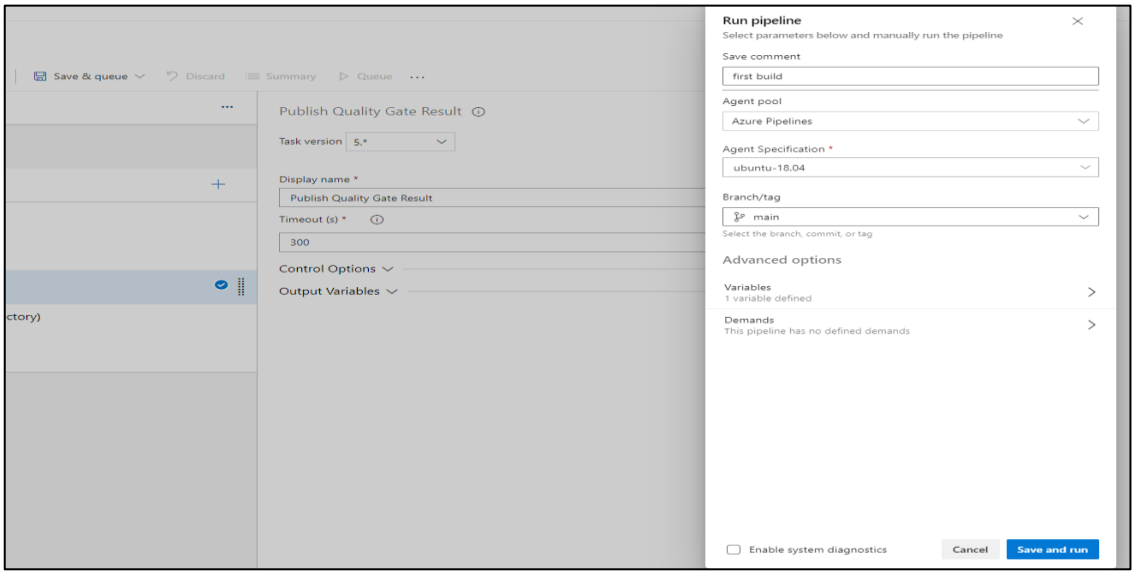


Fuente: elaboración propia, captura de pantalla. SonarQube *Publish Result.*

2.3.2.9. Ejecución de *pipeline*

Se procede a la ejecución del *pipeline*, como se muestra en la figura 34.

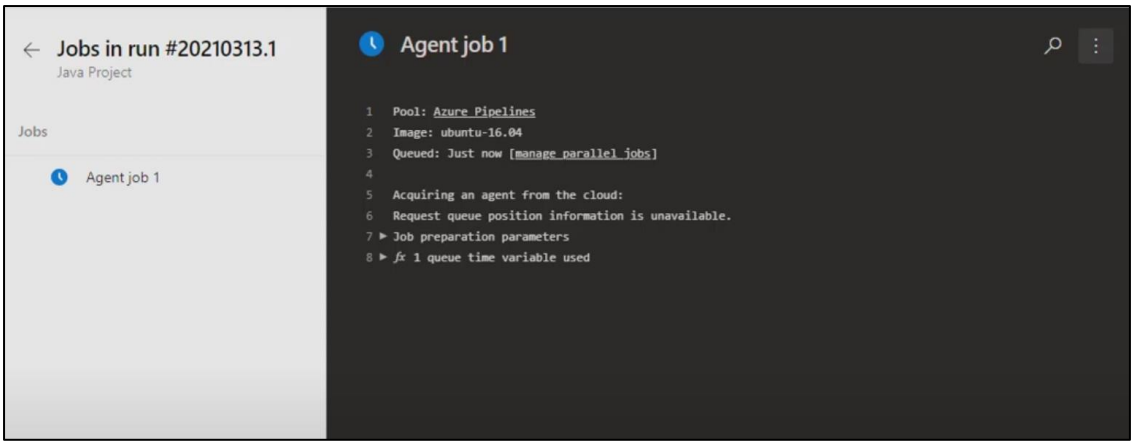
Figura 34. **Guardar configuración de *pipeline***



Fuente: elaboración propia, captura de pantalla. Ejecutar *pipeline.*

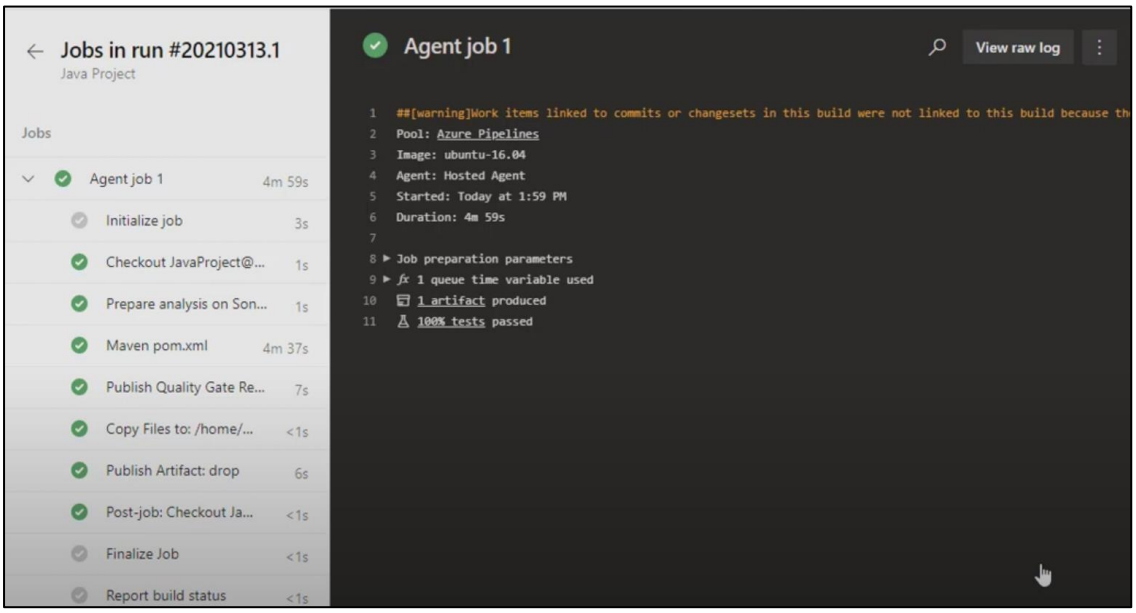
Se visualiza el *log* con las tareas del *pipeline* en ejecución, cumpliendo proceso de CI. En las figuras de la 35 a la 37, se visualiza el proceso de ejecución.

Figura 35. **Ejecución de *pipeline***



Fuente: elaboración propia, captura de pantalla. *Log pipeline.*

Figura 36. **Estado de ejecución de *pipeline***



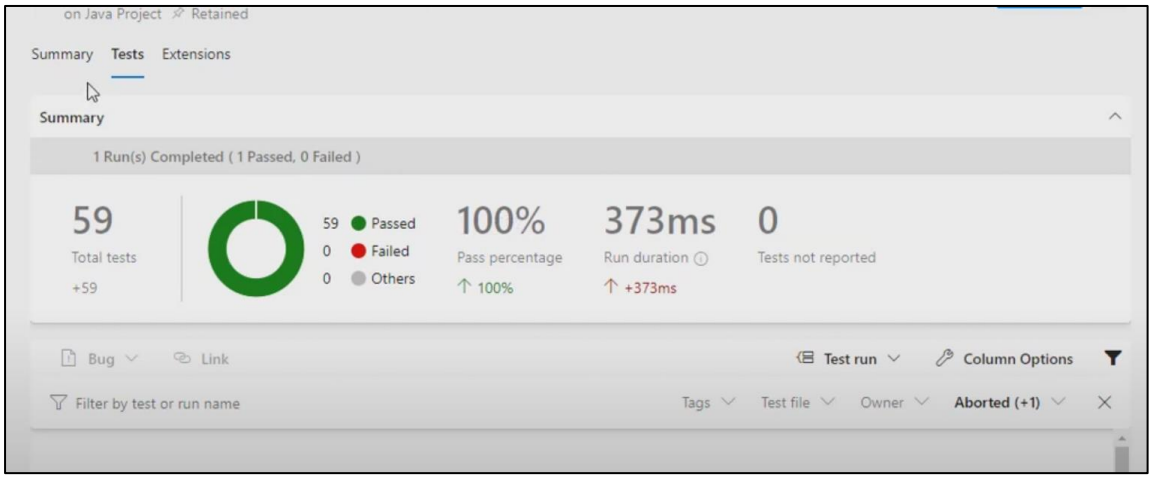
Fuente: elaboración propia, captura de pantalla. *Log pipeline.*

Figura 37. **Visualización de artefacto creado**

Fuente: elaboración propia, captura de pantalla, artefacto *pipeline.*

En la figura 38, se visualiza un resultado general de la ejecución dle análisis en Azure *pipeline*.

Figura 38. **Resultado análisis en Azure DevOps**

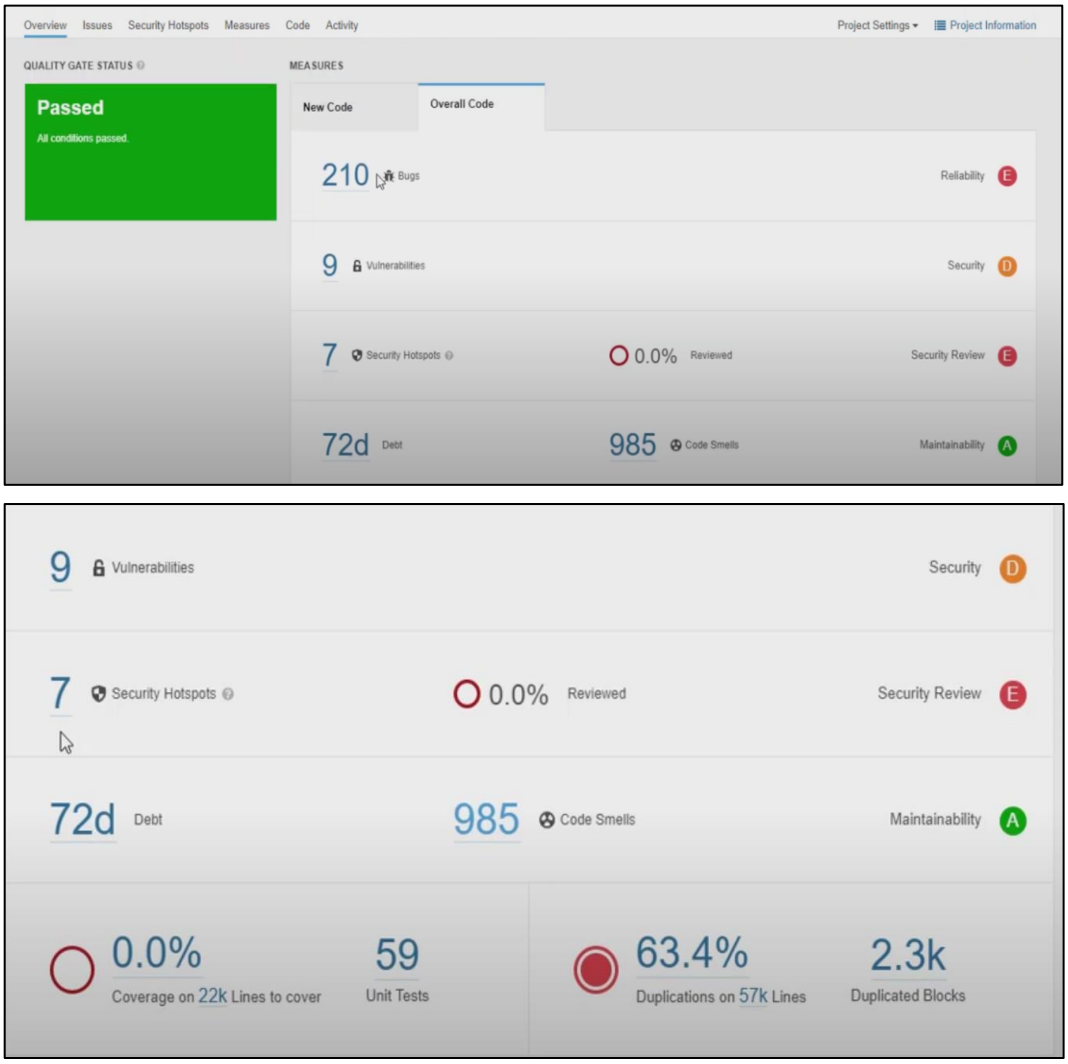


Fuente: elaboración propia, captura de pantalla, resultado *pipeline*.

2.3.2.10. Visualización de resultados de análisis de SonarQube

Posterior a la ejecución exitosa del *pipeline*, se visualizarán los resultados obtenidos del análisis del código en el servidor de SonarQube, ingresando al *dashboard* del proyecto, ver figura 39.

Figura 39. ***Dashboard* 1 resultados análisis SonarQube**



Fuente: elaboración propia, captura de pantalla. *Dashboard* SonarQube project.

2.4. Implementación en ambiente local

SonarQube puede implementarse en un ambiente local donde se estén codificando los proyectos o el repositorio local donde se esté versionando el código.

Esta implementación en un ambiente local es con el fin de realizar una inspección estática del código y así obtener un reporte de los errores que este pueda tener sin ser enviado a un ambiente de integración continua (CI).

Se procede a instalar el servidor de SonarQube en la que necesitamos instalar un motor de base de datos previo a la instalación del servidor de SonarQube utilizando el motor base de datos que este trae incrustado. Tiene ciertas limitaciones, pero es funcional en un ambiente local.

2.4.1. Implementacion en sistema operativo linux local

Ver requerimientos en la documentación oficial en la página oficial de SonarQube. Uno de los principales requerimientos para ejecutar SonarQube es tener instalado Java, Oracle JRE 11 o OpenJDK 11. Con esto ya se lleva a cabo la inspección de código en un ambiente local.

2.4.1.1. Descarga de SonarQube

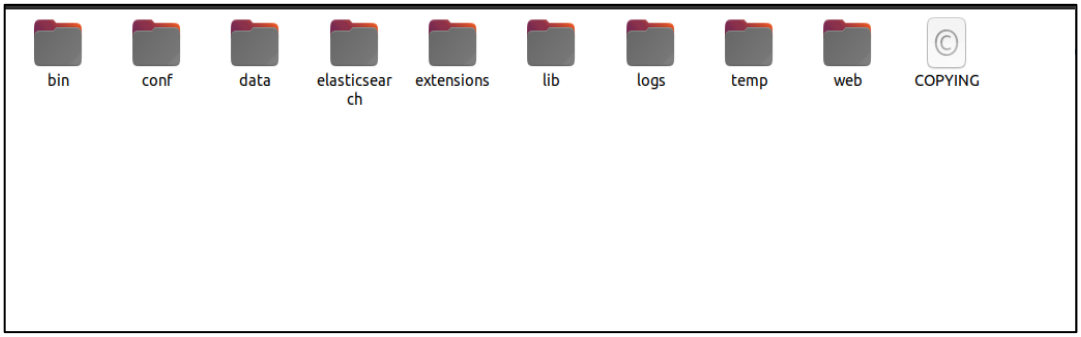
Para esta implementación, descargar SonarQube en extensión zip, desde la página oficial.

Se procede a instalar OpenJDK la versión 11. La instalación de este paquete se realiza con el siguiente comando:

```
$ sudo apt-get install openjdk-11-jdk -y
```

Después de haber descargado SonarQube en extensión zip, desde la página oficial la versión Community Edition, (https://www.sonarsource.com/products/sonarqube/downloads/), se procede a descomprimir dicho archivo, la estructura que estará será similar a la figura 40.

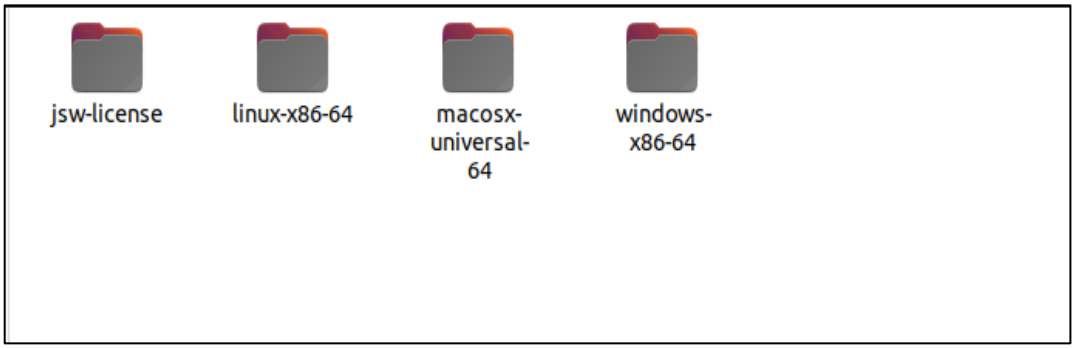
Figura 40. **Estructura de archivos SonarQube ZIP**



Fuente: elaboración propia, captura de pantalla. Estructura carpetas SonarQube.

Dentro de la carpeta bin, se encuentra el ejecutable específico para cada sistema operativo, ver figura 41.

Figura 41. **Ejecutables SonarQube**



Fuente: elaboración propia, captura de pantalla. Carpetas SonarQube.

En las configuraciones por defecto de SonarQube, se accederá al servidor por medio de la dirección local*host* y el puerto 9000 https://localhost:9000. Si se desea cambiar alguna configuración, realizarla en la ruta donde se encuentran los archivos de ejecución de SonarQube.

En la ruta donde se encuentra la carpeta *configure*, se encuentran los archivos de configuración y el principal de estos es el sonar.properties donde se encuentra la configuración para motor de bases de datos, el servidor web, autenticación y otros. Ver figura 42.

Figura 42. **Archivo de propiedades SonarQube**

Fuente: elaboración propia, captura de pantalla. *Dashboard* SonarQube *project.*

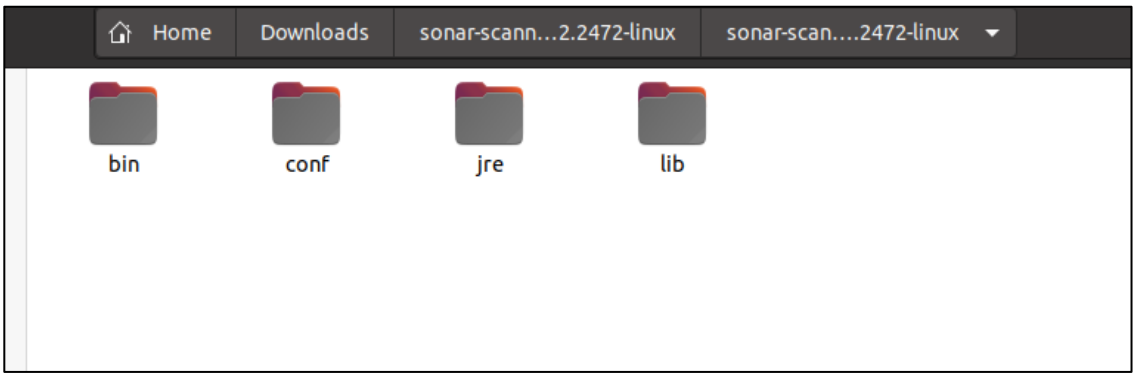
2.4.1.2. Configuración SonarQube

Para evaluar el código localmente, según el lenguaje de programación, será requerido complemento SonarScanner. Con este se evaluará el código en cualquiera de los lenguajes soportados por SonarQube.

- Proceder a descargar SonarScanner, esto se hace de igual manera en la página oficial de SonarQube. (SonarScanner, 2021)

- Descargar la versión CLI con base en el sistema operativo, en este caso, Linux.
- Descomprimir el archivo descargado, el cual tendrá la estructura que muestra en la figura 43.

Figura 43. **SonnarScanner CLI estructura de carpetas**



Fuente: Elaboración propia, captura de pantalla. SonnarScanner en Linux

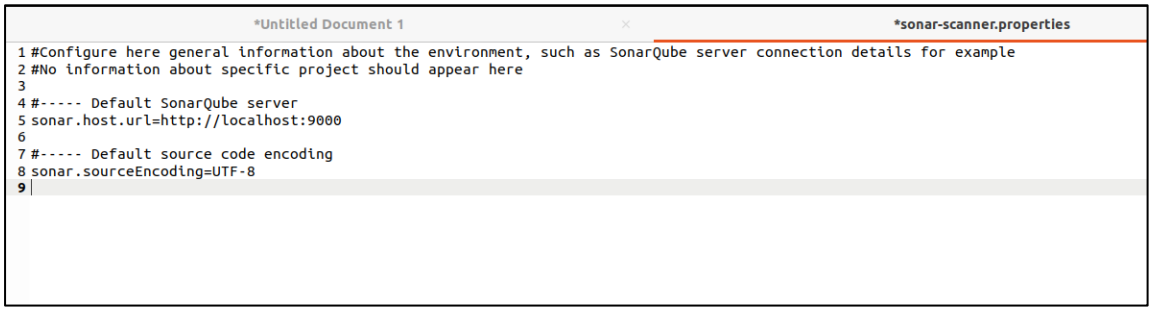
Agregar la carpeta *bin/bash* al PATH o a las variables de entorno del sistema para la ejecución de *Sonar-Scanner* utilizando *script*. Y consultar variables de entorno, ver figura 44.

Figura 44. **Configurar variable de entorno**

Fuente: elaboración propia, captura de pantalla. Variables de entorno en Linux.

Dentro de la carpeta *configure,* proceder a modificar el archivo *sonar-scanner.properties*, dentro de este archivo definir la ubicación del servidor local de SonarQube https://localhost:9000, ver figura 45.

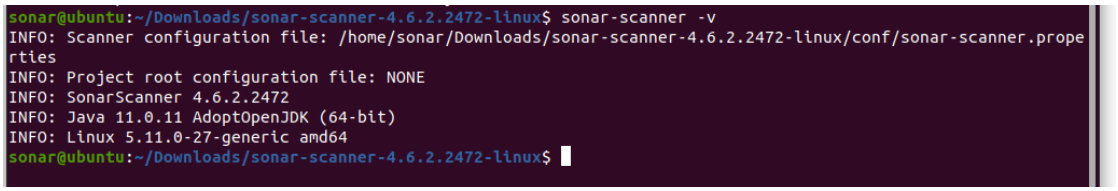
Figura 45. **Archivo configuración SonarQube**



Fuente: elaboración propia, captura de pantalla. Archivo configuración SonarQube.

Verificar que SonnarScanner esté configurado correctamente, como se muestra en la figura 46.

Figura 46. **Consultar configuración y versión SonarScanner**



Fuente: elaboración propia, captura de pantalla. Variables de entorno en Linux.

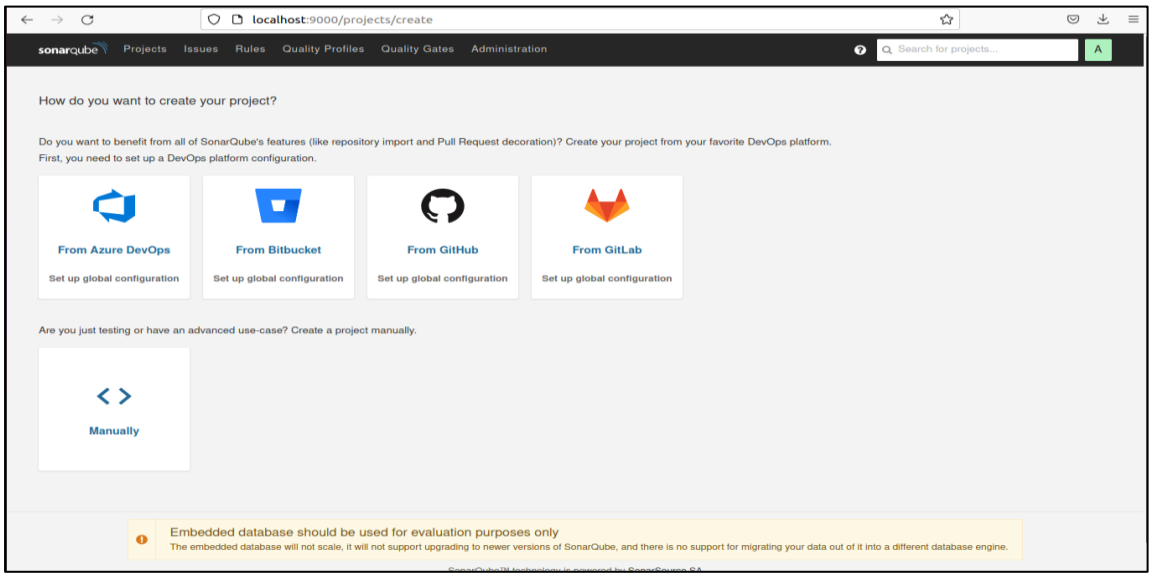
2.4.1.3. Ejecución local de SonarQube

Se procede a ejecutar el archivo del sistema operativo respectivo, en este caso bin/linux-x86-64 para la ejecución de SonarQube con el comando $ sh sonar.sh console. En el navegador se ingresa a la ubicación configurada ingresando las credenciales

Figura 47. **Ejecución SonarQube localmente**

Fuente: elaboración propia, captura de pantalla. Ejecución SonarQube.

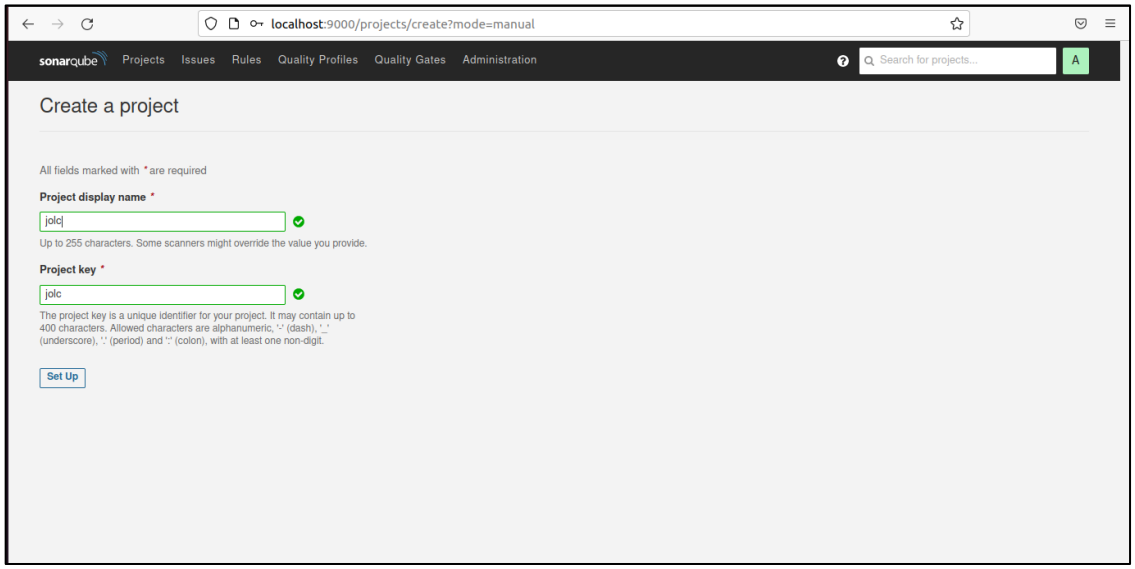
Figura 48. ***Dashboard* SonarQube**



Fuente: elaboración propia, captura de pantalla. Ejecución SonarQube.

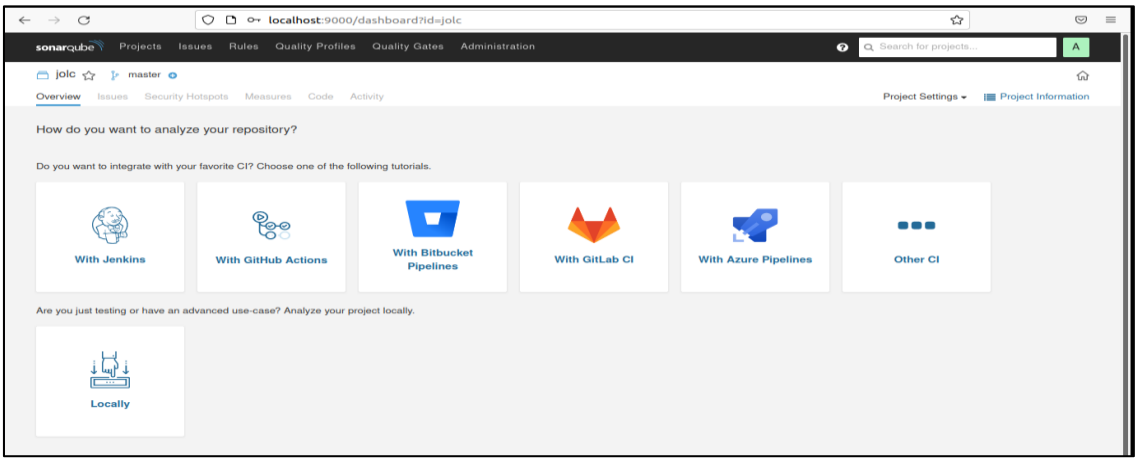
Se procede a crear un nuevo proyecto manualmente y seleccionar el proyecto o repositorio localmente, iniciar la configuración y generar el *token*, ver figuras de la 49 a la 51.

Figura 49. **Crear proyecto SonarQube localmente**



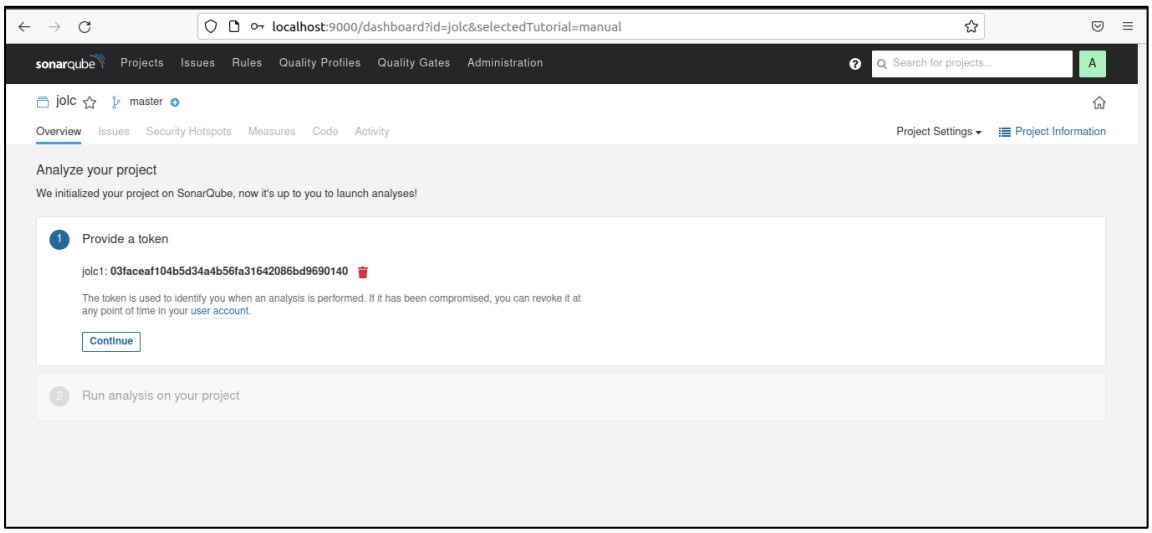
Fuente: elaboración propia, captura de pantalla. SonarQube Project.

Figura 50. **Seleccionar repositorio SonarQube localmente**



Fuente: elaboración propia, captura de pantalla. SonarQube Project.

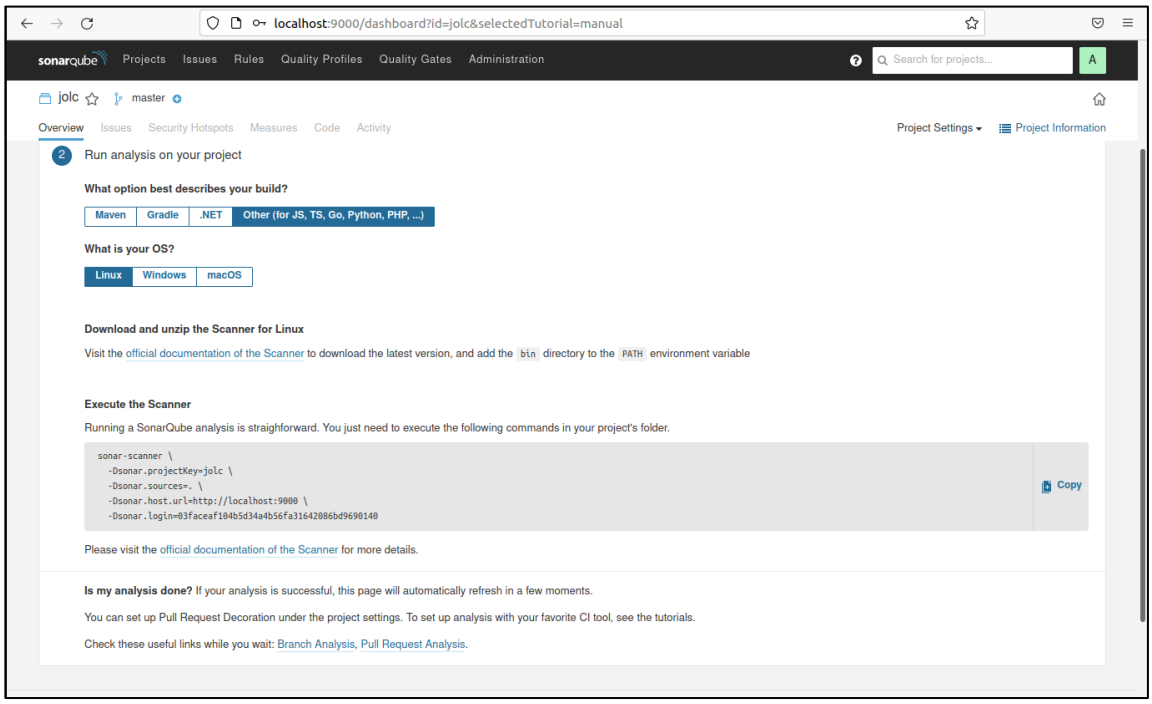
Figura 51. ***Token* SonarQube local**



Fuente: elaboración propia, captura de pantalla. SonarQube project.

Se procede a elegir el lenguaje de programación y el sistema operativo Linux, SonarQube muestra los comandos a ejecutar. Ver imagen 52.

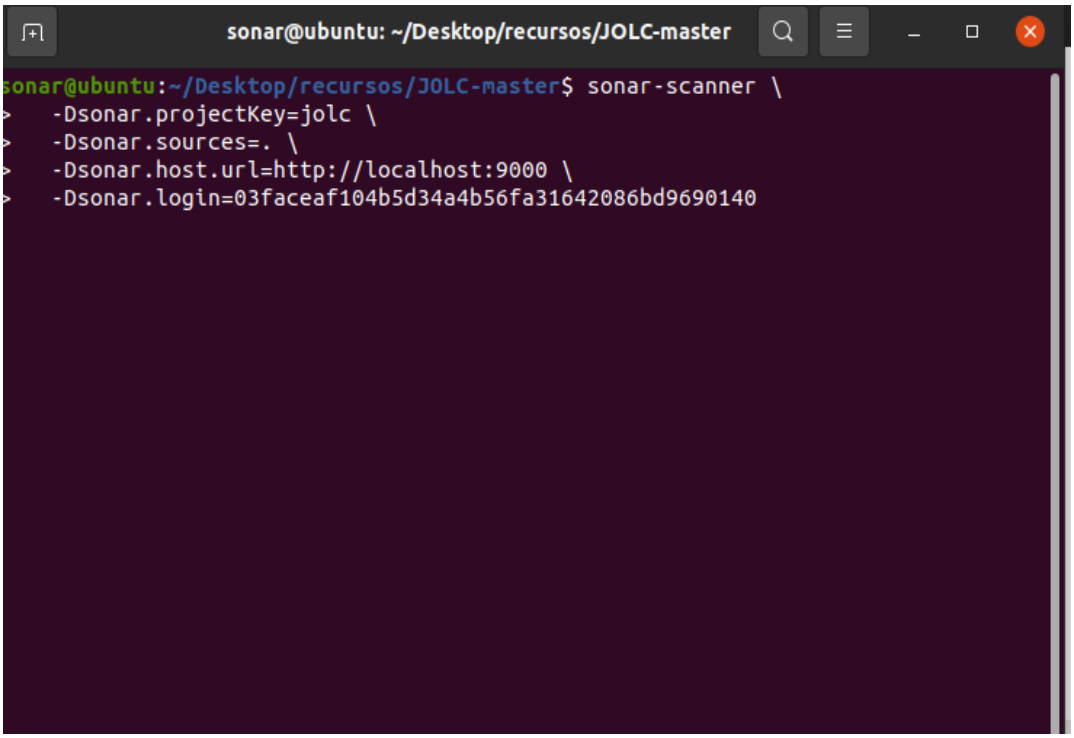
Figura 52. **Configuración de análisis SonarQube local**



Fuente: elaboración propia, captura de pantalla. SonarQube Project.

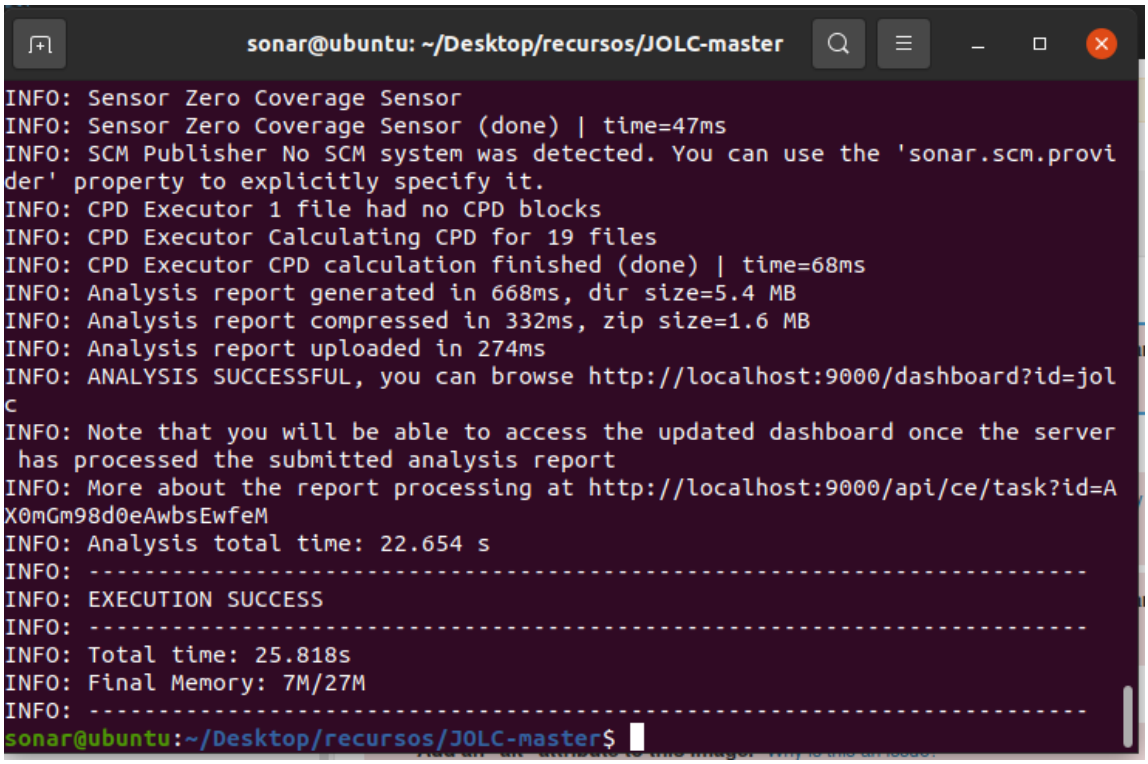
En la ruta donde se encuentra el proyecto a inspeccionar, se ejecuta el comando que está proporcionando SonarQube y esperar, ver figuras 53 y 54.

Figura 53. ***Script* ejecución análisis SonarQube local**



Fuente: elaboración propia, captura de pantalla. SonarQube, análisis de código.

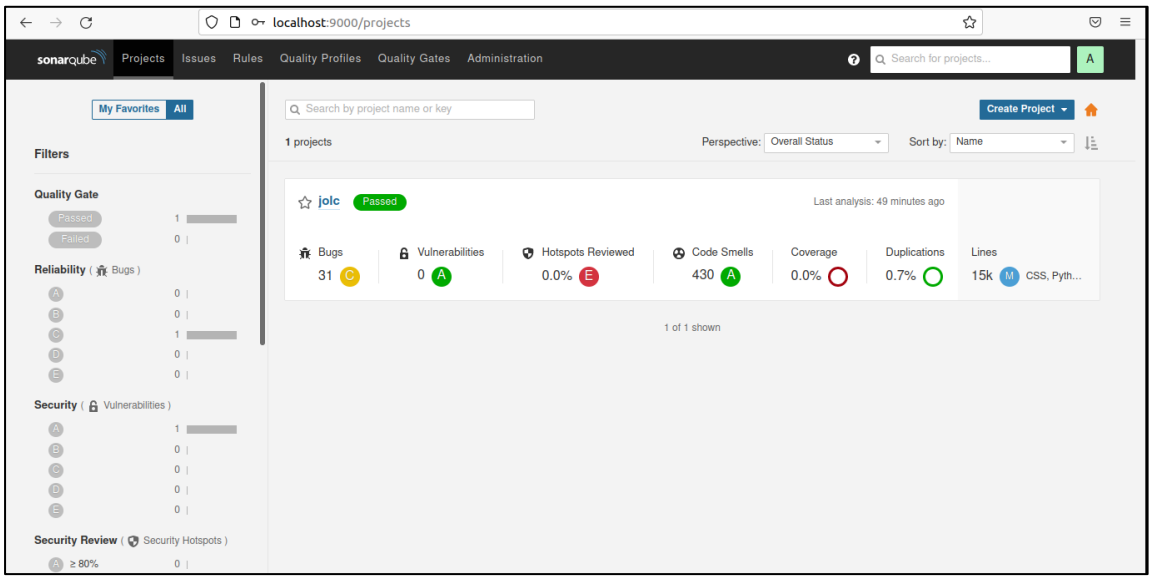
Figura 54. **Finaliza análisis SonarQube local**



Fuente: elaboración propia, captura de pantalla. *SonarQube* análisis de código.

Se procede a ingresar al *dashboard* de SonarQube y visualizar el reporte de resultados, ver figura 55.

Figura 55. ***Dashboard* reporte SonarQube local**



Fuente: elaboración propia, captura de pantalla. SonarQube, resultados.

2.4.2. Análisis de código en un entorno de desarrollo integrado

Todo desarrollador de *software* debe tener una herramienta esencial que le permita detectar errores, obtener una retroalimentación y sugerencia para la solución de he dichos errores, y al mismo tiempo aprender nuevas y mejores prácticas de desarrollo convirtiéndose en un desarrollador más eficiente, permitiéndole al desarrollador tomar las mejores decisiones de codificación en su propio entorno de desarrollo integrado.

Dicha herramienta, que proporciona dicha retroalimentación y es parte también de uno de los productos proporcionados para análisis de código por SonarSource es *SonarLint*.

SonarLint, al igual que el servidor de SonarQube, soporta varios lenguajes de programación y es compatible con una gran cantidad de entornos de desarrollo integrado, por lo que se visualiza en la plataforma las características de dicha herramienta (SonarSource, 2021).

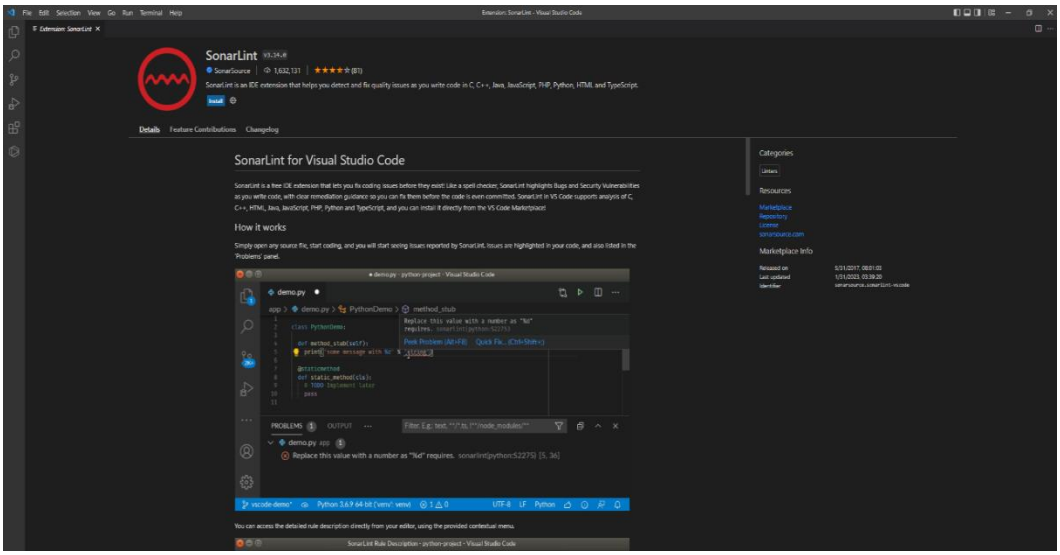
SonarLint contiene una cantidad muy amplia de reglas configuradas para los diferentes tipos de lenguajes de programación, tomando en cuenta atributos de código que generan valor durante el desarrollo los cuales son confiabilidad, mantenibilidad, legibilidad, seguridad, calidad y otros más. Brindando un reporte en tiempo real del análisis realizado al código mientras se trabaja al mismo tiempo (SonarSource, 2008).

2.4.2.1. Integración de *SonarLint* al entorno de desarrollo integrado IDE

SonarLint puede ser integrado a diferentes tipos de entornos de desarrollo integrado como lo menciona la documentación (SonarLint.org, 2021), entre los cuales se encuentran los más utilizados, actualmente, por los desarrolladores y con el tiempo esta compatibilidad ha ido en aumento, así como los lenguajes de programación soportados.

Para esta implementación se utilizará el entorno de desarrollo integrado VS Code, ya que este soporta diferentes tipos de lenguajes de programación. Y se procede a la descarga de la extensión para este IDE.

Figura 56. ***SonarLint* extensión para VSCode**

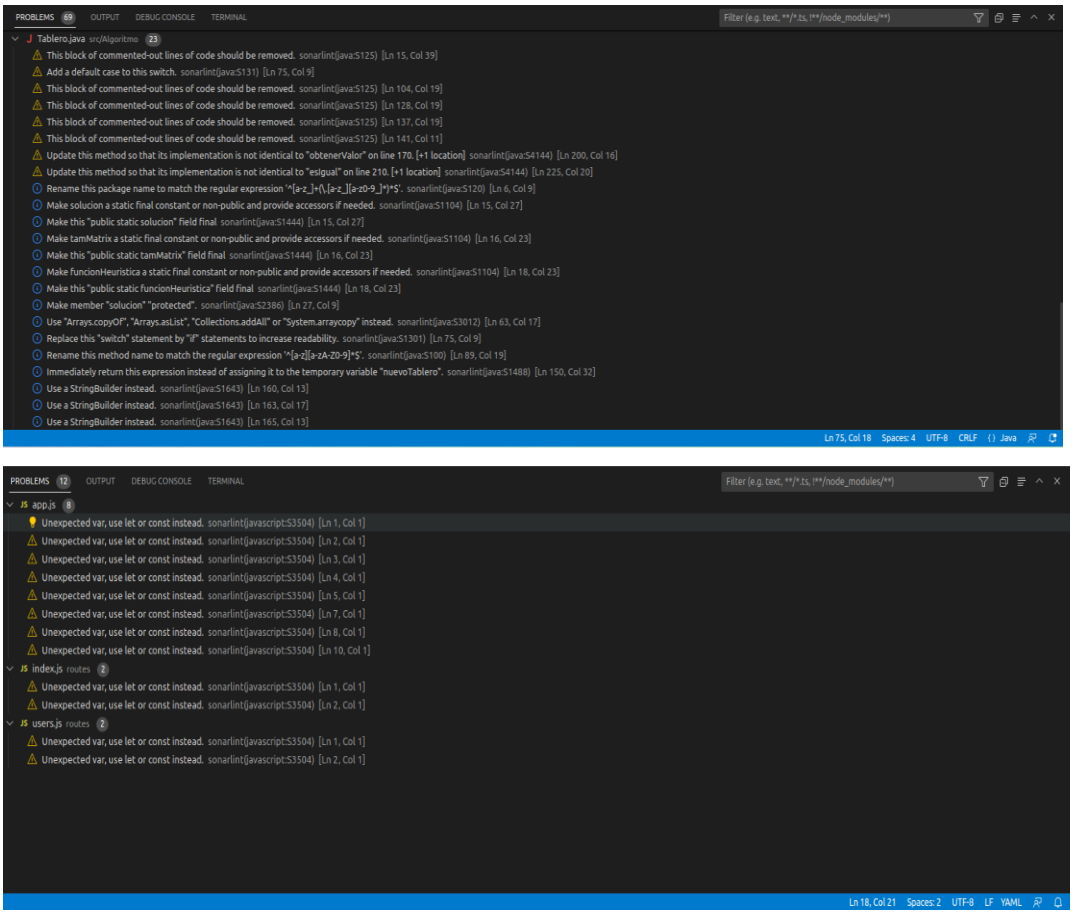


Fuente: elaboración propia, captura de pantalla, *SonarLint* extensión VSCODE.

Para ejecutar esta herramienta hay que tener instalado en el ambiente de desarrollo Java Runtime (JRE) 11.

Proceder a crear el proyecto que se desarrollará o abrir la carpeta raíz donde se encuentre el proyecto existente que se requiera desarrollar y analizar; *SonarLint* al detectar lenguaje utilizado procederá analizar el código en tiempo real mientras trabaja. El reporte con errores o sugerencias de mejora de código se visualizará en la pestaña de problemas. Ver figura 57.

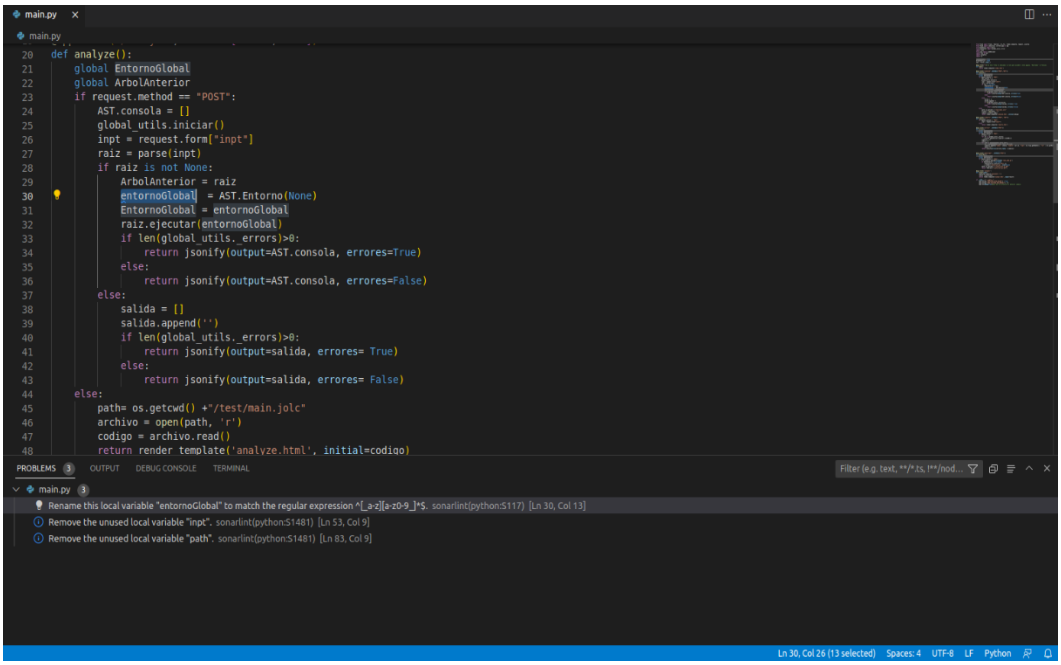
Figura 57. **Reporte 1 de *SonarLint* en IDE**



Fuente: elaboración propia, captura de pantalla. Reporte *SonarLint.*

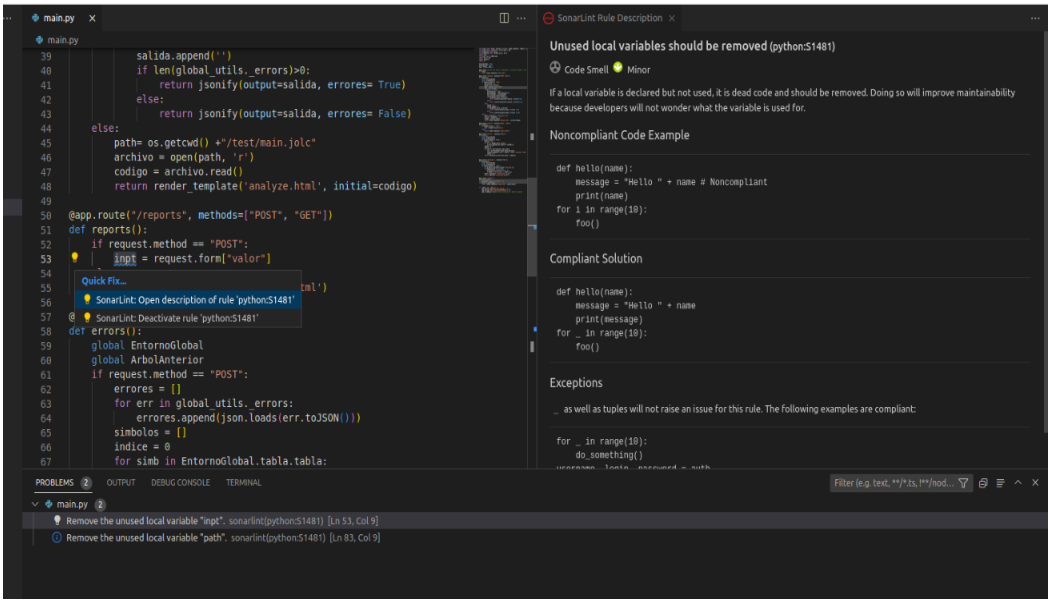
Dichos errores, no solo se mostrarán en la parte inferior, sino también un ícono de sugerencia en la línea específica donde puede haber un error o mejora al código fuente y se podrá ampliar dicha información, ver figuras 58-59.

Figura 58. **Sugerencias y error en editor de IDE, *SonarLint***

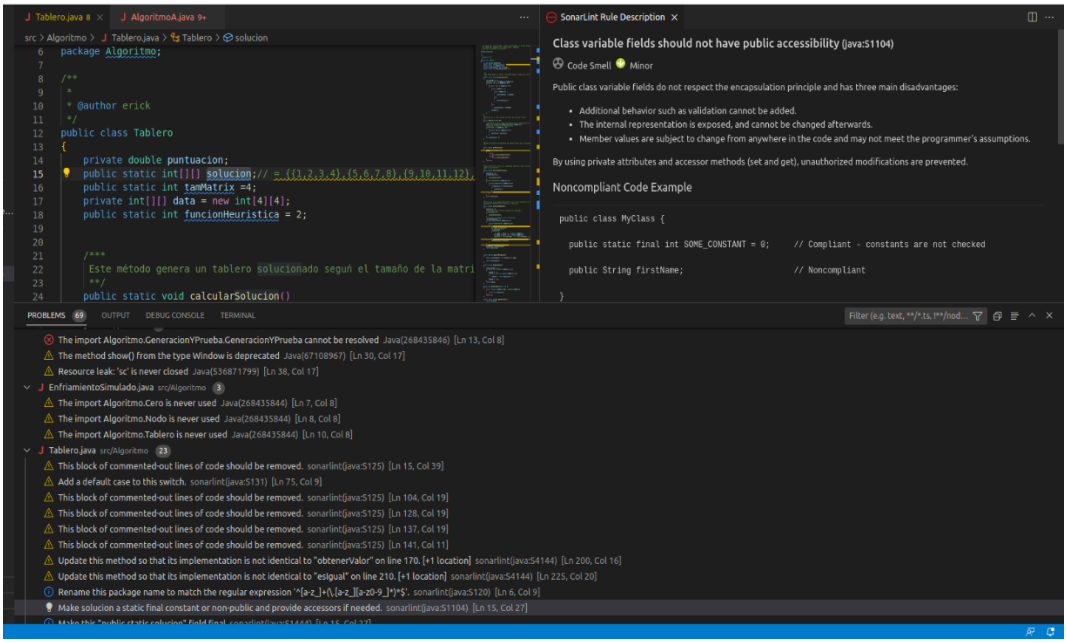


Fuente: elaboración propia, captura de pantalla, reporte *SonarLint*.

Figura 59. **Ampliar información de regla *SonarLint***



Continuación de la figura 59.



Fuente: elaboración propia, captura de pantalla. Reporte *SonarLint.*

2.5. Configuración de características y herramientas de SonarQube

A continuación, se mostrará la configuración de las diversas características que contiene SonarQube para un mejor análisis y mejoramiento de nuestro código durante la fase de desarrollo.

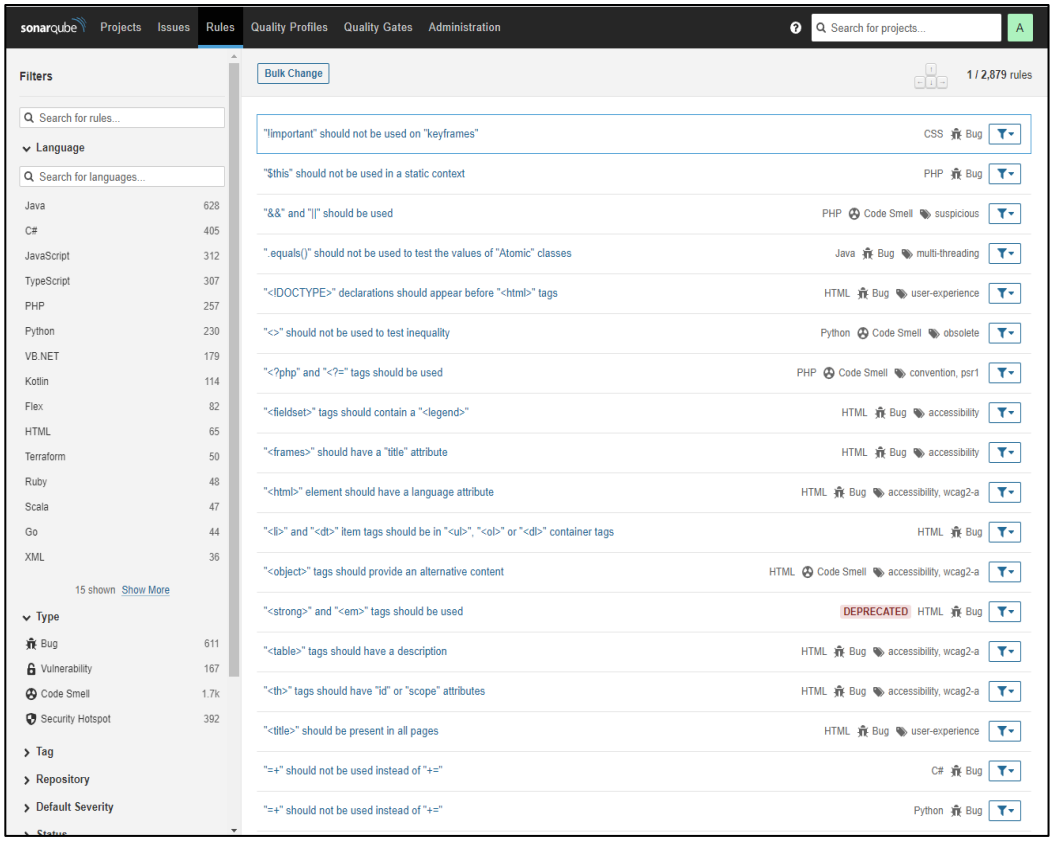
2.5.1. Reglas en SonarQube

Las reglas en SonarQube son un conjunto de criterios predefinidos o personalizados que se utilizan para evaluar la calidad de tu código e identificar posibles problemas relacionados con los errores, la seguridad y el mantenimiento del código.

Estas reglas están diseñadas para ayudar a identificar problemas de código comunes, como olores de código, errores y vulnerabilidades de seguridad, ver figura 60.

El uso de reglas en SonarQube puede ayudar a identificar problemas potenciales en su código al principio del proceso de desarrollo, antes de que se vuelvan más difíciles y costosos de solucionar.

Figura 60. **Reglas SonarQube**

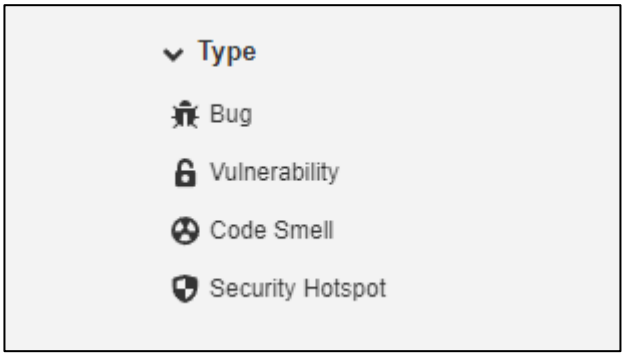


Fuente: elaboración propia, captura de pantalla. Reglas SonarQube Server.

Se filtrarán las reglas, según el lenguaje de programación al cual pertenecen, así como visualizar dichas reglas y personalizarlas a conveniencia.

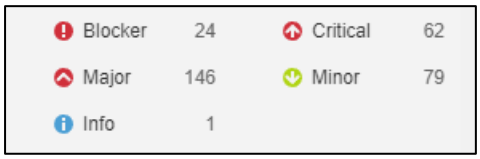
Las reglas también se pueden clasificar de diferentes maneras. A continuación, las más importantes en las figuras 61 y 62.

Figura 61. **Tipos de reglas SonarQube**



Fuente: elaboración propia, captura de pantalla. Reglas SonarQube Server.

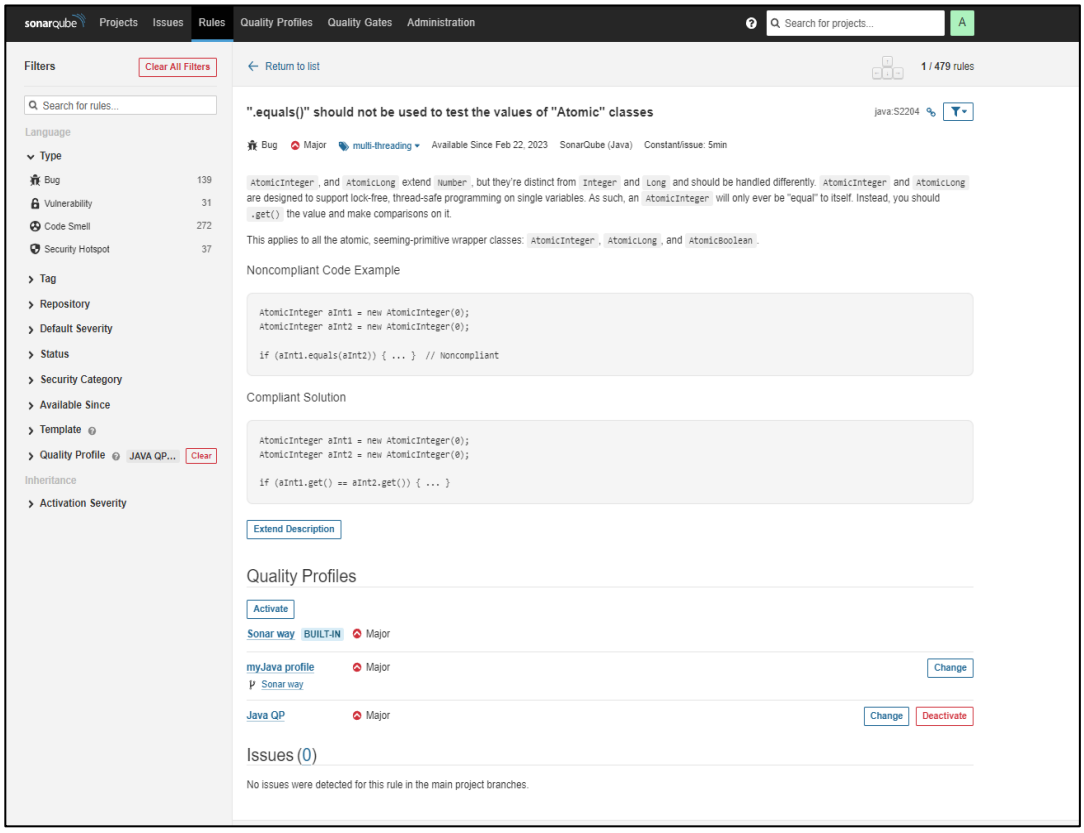
Figura 62. **Tipo de impacto reglas en SonarQube**



Fuente: elaboración propia, captura de pantalla. Reglas SonarQube Server

También se aplican etiquetas para su identificación y clasificación. A continuación, una regla predefinida por SonarQube para el lenguaje de programación JavaScript y las modificaciones que se realizan, ver figura 63.

Figura 63. **Visualización de una regla en SonarQube**



Fuente: elaboración propia, captura de pantalla. Reglas SonarQube Server.

2.5.2. Quality profile en SonarQube

Los *quality profiles* en SonarQube son un conjunto de reglas que definen los estándares de calidad para el código del proyecto y son un componente importante de las capacidades de análisis de código de SonarQube.

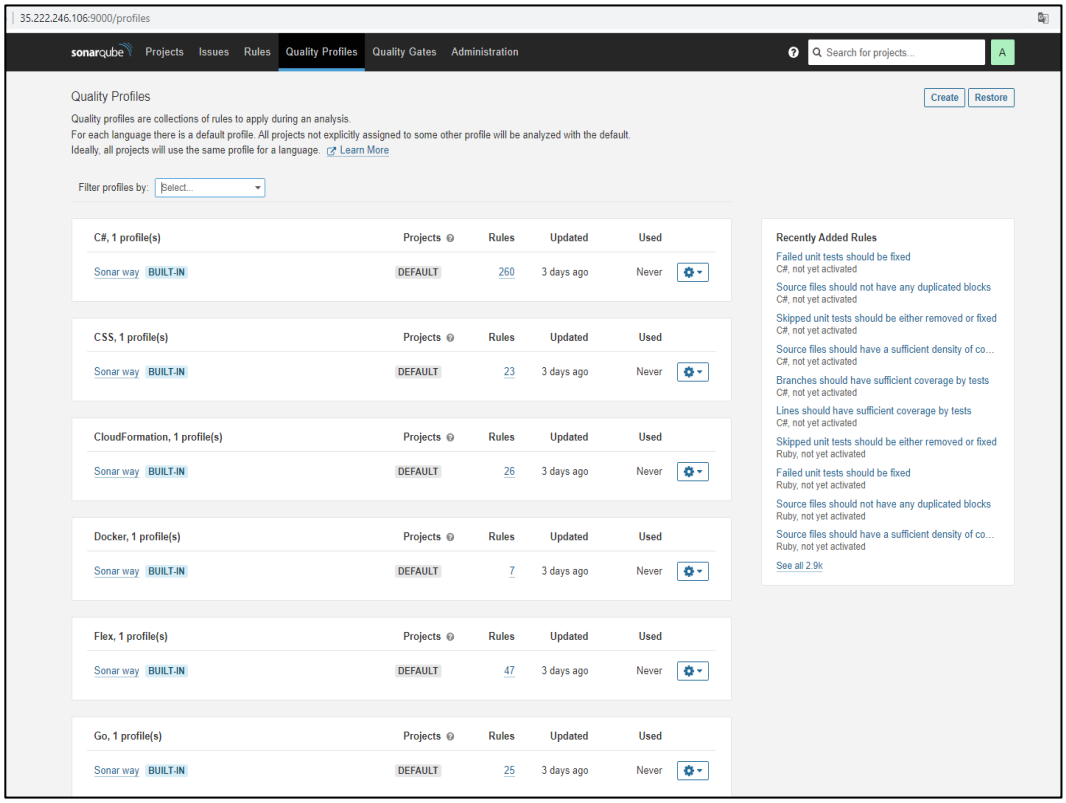
Los *quality profile* son personalizables, lo que le permite agregar o eliminar reglas, según sea necesario para satisfacer las necesidades específicas del proyecto.

2.5.2.1. Creación y configuración de un *quality profile* en SonarQube

De forma predeterminada, SonarQube proporciona *quality profiles* integrados, pero también puede crear los propios y personalizados para satisfacer las necesidades específicas del proyecto. Los *quality profiles,* también se pueden compartir entre proyectos o equipos, lo que facilita la estandarización de la calidad del código y las prácticas de seguridad en toda la organización.

La creación de un *quality profile* no es muy recomendable hacerlo porque SonarQube, ya proporciona de forma predeterminada reglas establecidas que están configuradas en los *quality profiles* para los lenguajes de programación que soporta. Y SonarQube actualiza de manera muy recurrente dichos *quality profiles* y las reglas que estos contiene, basadas en mejoras prácticas de programación. En la figura 64 se muestran los *quality gates* proporcionados por SonarQube.

Figura 64. **Visualización de *quality profile* en SonarQube**

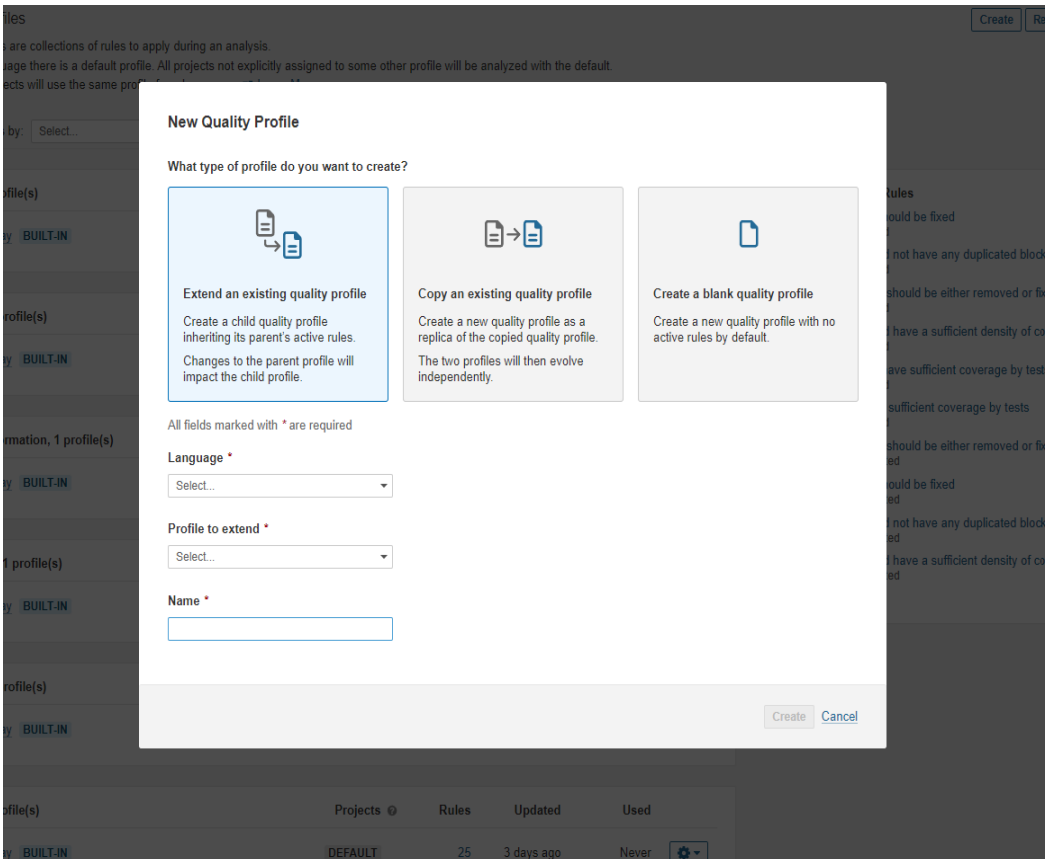


Fuente: elaboración propia, captura de pantalla. *Quality profile* SonarQube Server.

2.5.2.1.1. Creación de un *quality profile*

Existen tres formas de crear un *quality profile,* lo recomendable es crear un *quality profile* basado en un perfil que SonarQube, porque proporciona por defecto o de un ya existente según el lenguaje de programación, ya que, si se elige crear un *quality profile* desde cero, este no tendrá ninguna regla relacionada, por lo que sería tarea del usuario asociar dichas reglas al perfil a utilizar, ver figura 65.

Figura 65. **Creación de *quality profile* SonarQube**



Fuente: elaboración propia, captura de pantalla. *Quality profile* SonarQube Server.

Al visualizar la imagen anterior existen tres opciones para crear un *quality profile.*

La primera opción es crear un perfil heredando las reglas asociadas a este, el perfil que hereda dichas reglas asociadas se modificarán a conveniencia, pero si el perfil padre tiene cambios, estos se replicarán en el perfil que hereda dichas configuraciones.

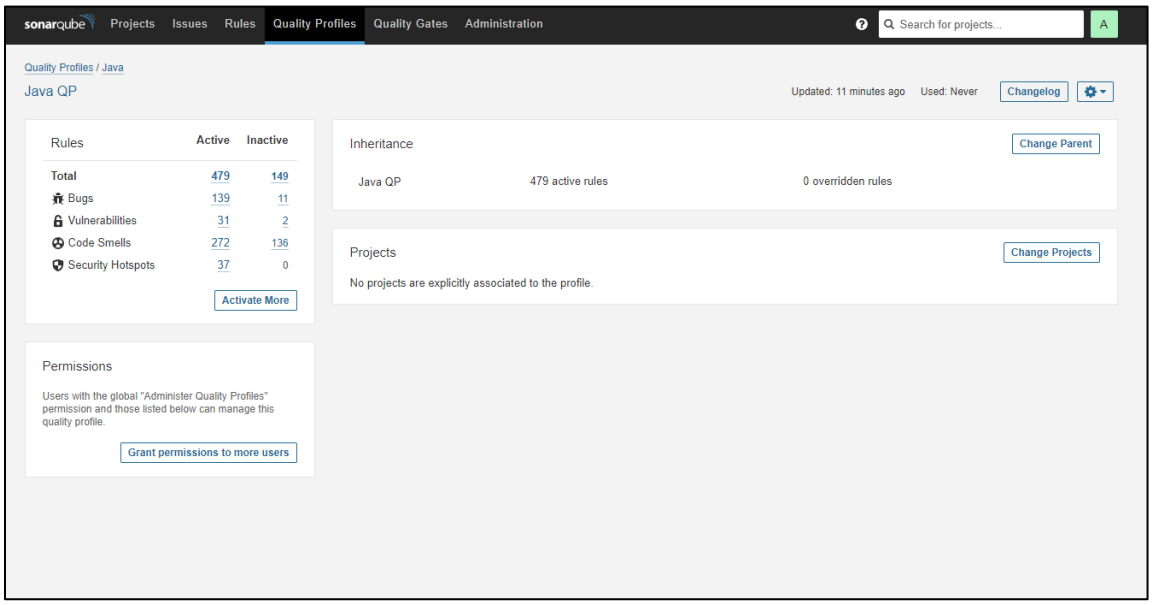
La segunda opción es crear una copia idéntica de un perfil que exista en nuestro servidor de SonarQube, por lo que al crear un perfil por medio de esta opción dichos perfiles serán independientes, y los cambios aplicados a cualquiera de los dos no afectarán al otro u otros.

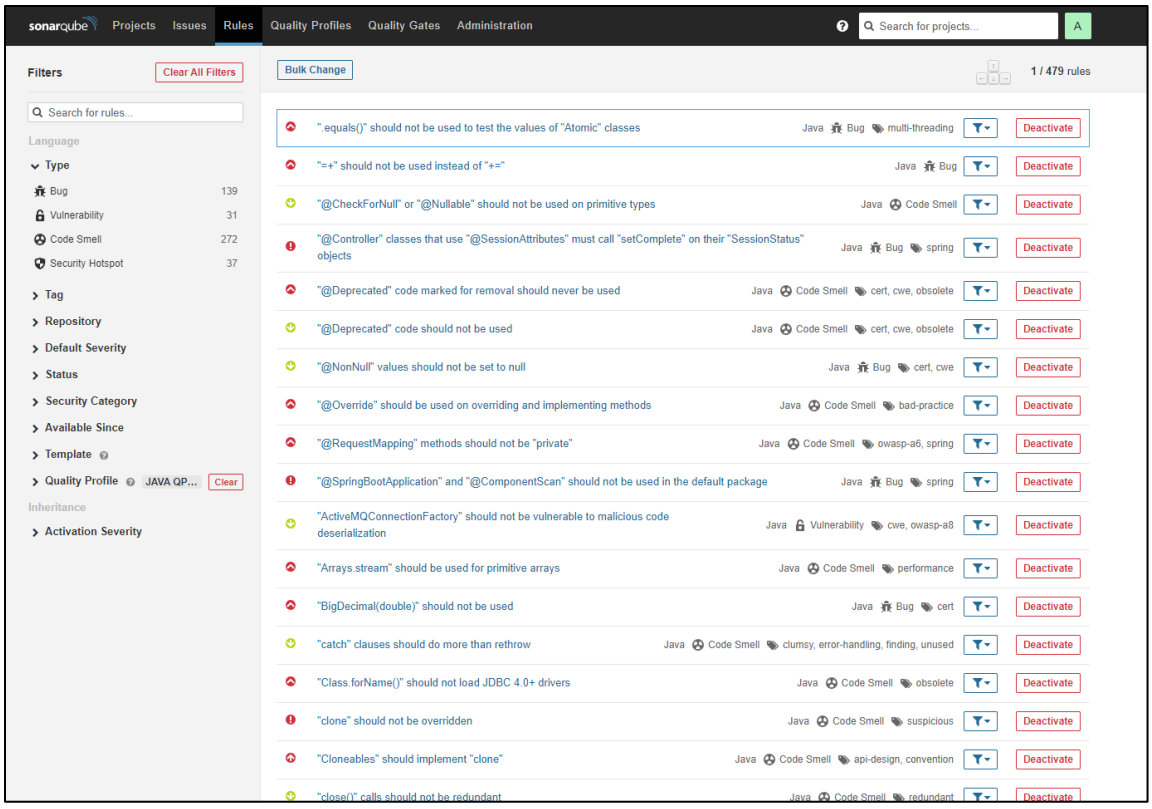
La tercera opción, que no es recomendable, crear un *quality profile* desde cero.

Los *quality profiles* que brinda SonarQube por defecto están creados en base a análisis realizados, por lo que SonarSource que recopila dicha información para ir mejorando las reglas e ir estableciendo dichos perfiles, así como actualizarlos.

A continuación, se muestra un *quality profile* con reglas asociadas en la cual pueden modificarse dicho perfil asociando o eliminando reglas, si el perfil es heredado de uno existente se cambia el perfil padre, así como también cambiar los proyectos a los cuales este aplicado dicho perfil, ver figura 66.

Figura 66. **Creación, configuración *quality profile* SonarQUbe**





Fuente: elaboración propia, captura de pantalla. *Quality profile* SonarQube Server.

2.5.3. *Quality gates* en SonarQube

Los *quality gates* de SonarQube son una característica importante que ayuda a garantizar la calidad del código en los proyectos de *software*. Estas son condiciones específicas que se definen en SonarQube y se aplican a un proyecto para garantizar que se cumplan los requisitos de calidad antes de permitir que el código se mueva a la siguiente fase del ciclo de vida del software. Los *quality gates* se pueden aplicar a diferentes tipos de proyectos y se pueden personalizar, según las necesidades específicas de cada proyecto.

Por lo que una buena definición de los *quality gates* de SonarQube puede garantizan la calidad del código en los proyectos de *software*. Al implementar *quality gates*, los equipos de desarrollo pueden garantizar que el código cumpla con los estándares de calidad antes de que se implemente en producción. Esto ayuda a reducir el número de errores en producción y aumenta la eficiencia del equipo de desarrollo al garantizar que se corrijan los errores antes de que sean demasiado costosos de arreglar.

2.5.3.1. Creación y configuración de *quality gates* en SonarQube

La configuración de *quality gates* en SonarQube se realiza a través de la interfaz de administración de SonarQube, donde se pueden crear, modificar y asignar *quality gates* a proyectos específicos. También se pueden personalizar las métricas y umbrales disponibles en SonarQube para adaptarlos a las necesidades y estándares de calidad de cada proyecto.

2.5.3.1.1. Buenas prácticas para el uso correcto de *quality gates*

A continuación, se describen algunas buenas prácticas que deben considerarse en la creación o configuración de los *quality gates en* SonarQube:

- Definir los *quality gates* adecuados: establecer las reglas que se aplicarán y los umbrales que se utilizarán para determinar si una nueva versión del *software* cumple o no con los requisitos de calidad.

- Configurar los *quality gates*: es necesario definir las reglas y métricas que se utilizarán para evaluar la calidad del código. Realiz*ar* pruebas en diferentes entornos para verificar que los *quality gates* están funcionando correctamente.

- Integrar los *quality gates* en el ciclo de vida del desarrollo.

- Establecer criterios claros para determinar si un proyecto ha sido exitoso o no en términos de calidad del *software*. Los criterios de éxito deben ser específicos, medibles, alcanzables, relevantes y oportunos.

- Realizar seguimiento y análisis de los resultados: dar seguimiento a los resultados de los *quality gates* en SonarQube, para identificar problemas de calidad y realizar mejoras en el código. Además, analizar los datos recopilados para determinar las tendencias en la calidad del *software* y realizar ajustes en los *quality gates,* según sea necesario.

2.5.3.1.2. Factores a tomar en cuenta para el uso de correcto de *quality gates*

- Objetivos de calidad del proyecto: es importante tener claros los objetivos de calidad del proyecto, para definir qué métricas y umbrales se deben establecer en el *quality gate.*
- Tipo de proyecto: puede influir en los umbrales que se establecen en el *quality gate*. Por ejemplo, los umbrales para un proyecto web pueden ser diferentes a los de un proyecto de *software* embebido.
- Prioridades de calidad: definir las que se consideran más importantes para el proyecto. Por ejemplo, puede ser más importante la seguridad del código que la complejidad.
- Madurez del proyecto: el *quality gate* debe adaptarse a la madurez del proyecto, teniendo en cuenta que los umbrales pueden ser más restrictivos en las primeras etapas del proyecto y menos restrictivos en etapas más avanzadas.
- Requisitos del cliente: para definir los umbrales de calidad adecuados.
- Equipo de desarrollo: debe ser considerado en la definición de los umbrales, teniendo en cuenta su experiencia y capacidad para alcanzar los objetivos establecidos.
- Métricas de SonarQube: ofrece una amplia variedad de métricas, por lo que es importante evaluar cuáles son las métricas más relevantes para el proyecto y cómo se utilizarán para medir la calidad del código.
- Umbral de calidad: definir umbrales de calidad que permitan al equipo de desarrollo detectar problemas y trabajar en la mejora continua del código, pero que no sean tan restrictivos que afecten negativamente el proceso de desarrollo.

Al tomar en cuenta estos factores, se puede configurar un *quality gate* en SonarQube que sea efectivo y adaptado a las necesidades del proyecto y del equipo de desarrollo.

2.5.3.1.3. Definición de métricas recomendada durante la configuración de *quality gates*

En este contexto, las métricas son indicadores numéricos que miden diferentes aspectos del código fuente, como la complejidad, la duplicación, la cantidad de errores, la cobertura de pruebas, entre otros. Estas métricas se utilizan para establecer los umbrales en los *quality gates*, es decir, los valores mínimos o máximos que debe cumplir cada métrica para que el código sea considerado de calidad.

- Cobertura de código: es la métrica que mide la cantidad de código que es cubierto por pruebas unitarias. Es importante porque permite asegurar que el código está siendo probado de manera adecuada y que se están detectando posibles errores. El umbral recomendado para esta métrica suele ser del 80 %.

- Duplicación de código: esta métrica mide la cantidad de código repetido en el proyecto. La duplicación de código puede aumentar la complejidad y la dificultad de mantenimiento del proyecto, por lo que es importante mantenerla bajo control. El umbral recomendado para esta métrica suele ser del 3 %.

- Complejidad ciclomática: esta mide la complejidad del código en términos de la cantidad de caminos posibles que pueden recorrerse a través de este. Cuanto mayor es la complejidad ciclomática, mayor es la dificultad de entender y mantener el código. El umbral recomendado para esta métrica suele ser del 10 %.

- Cantidad de errores: esta mide la cantidad de errores en el código, como, por ejemplo, variables no inicializadas o errores de sintaxis. Es importante mantener la cantidad de errores bajo control para evitar posibles problemas en el funcionamiento del proyecto. El umbral recomendado para esta métrica suele ser del 0 %.

- Tasa de comentarios: esta mide la cantidad de comentarios en el código. Los comentarios son importantes para facilitar la comprensión del código y su mantenimiento en el futuro. El umbral recomendado para esta métrica suele ser del 20 %.

- Vulnerabilidades: es la categoría más crítica de las métricas de seguridad y deberían tener un umbral de cero. Es decir, no se debe permitir ninguna vulnerabilidad en el código fuente. Si se encuentra alguna vulnerabilidad, debe corregirse inmediatamente.

2.5.3.1.4. Configuración de *quality gates* en SonarQube

SonarQube, ya proporciona un *quality gate* configurado por defecto, este según la documentación oficial de SonarQube, trae implementada la metodología Clean as You Code, es un enfoque de la calidad del código que

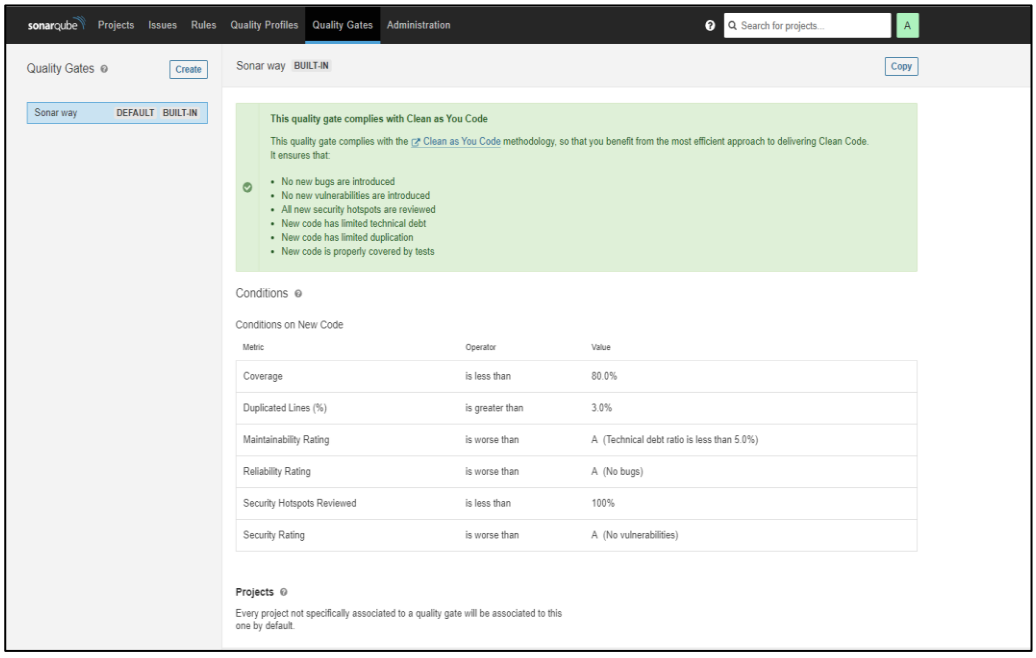
elimina muchos de los desafíos que conllevan las metodologías tradicionales. Y así como desarrollador, se enfoca en mantener altos estándares y asumir la responsabilidad específicamente en el nuevo código en el que está trabajando. Cumpliendo con este enfoque se obtienen los siguientes beneficios en el nuevo código creado:

- No se introducen errores
- No se introducen vulnerabilidades
- Se revisan todos los puntos de acceso de seguridad nuevos
- El código tiene una deuda técnica limitada
- El código tiene una duplicidad limitada
- El código está debidamente cubierto por las pruebas unitarias

A continuación, se describen los pasos necesarios para la creación y configuración de *quality gates* en SonarQube*:*

- Acceder a la sección de *quality gates*: en el menú principal de SonarQube, acceder a la sección de *quality gates* que se encuentra en el menú lateral, ver figura 67.

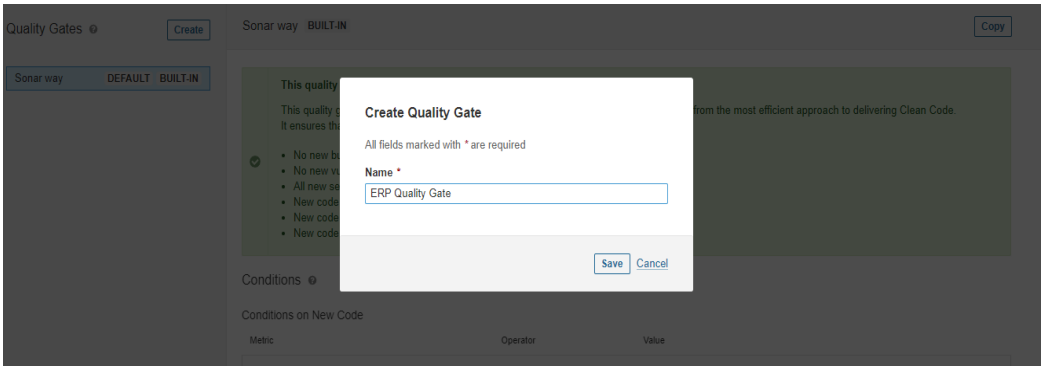
Figura 67. ***Dashboard quality gates* en SonarQube**



Fuente: elaboración propia, captura de pantalla. *Quality gates SonarQube Server.*

- Crear un nuevo *quality gate*: para crear un nuevo *quality gate*, hacer clic en el botón *Create* y asignarle un nombre, ver figura 68.

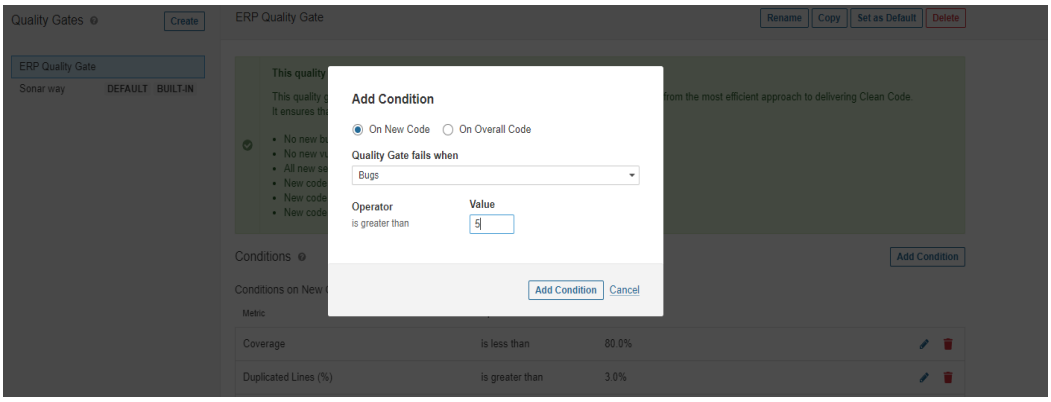
Figura 68. **Creación *quality gate* en *SonarQube***



Fuente: elaboración propia, captura de pantalla. *Quality gates SonarQube Server.*

- Configurar las condiciones del *quality gate*: dentro del *quality gate,* se pueden configurar una serie de condiciones para que el código sea evaluado en función de las métricas de *SonarQube*. Estas condiciones pueden incluir valores para métricas como la cobertura de pruebas unitarias, el número de bugs o la complejidad ciclomática, entre otras. Ver figura 69.

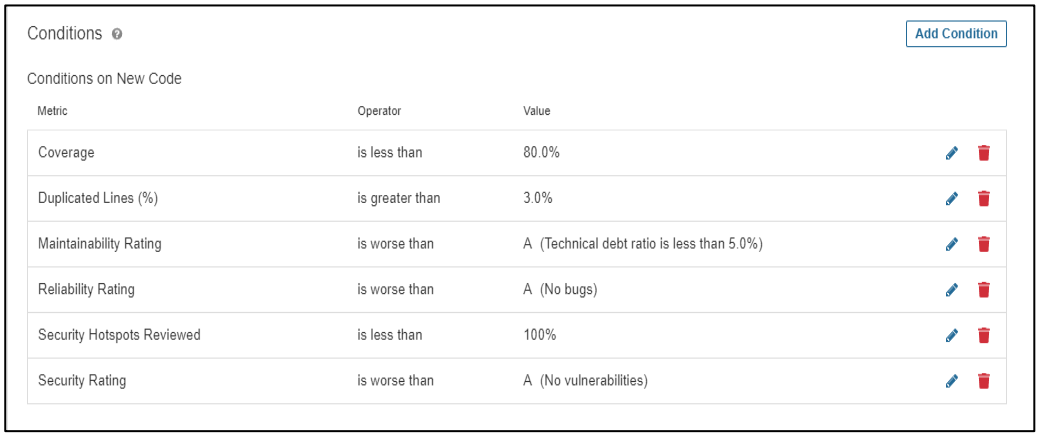
Figura 69. **Condiciones en *quality gates* SonarQube**



Fuente: elaboración propia, captura de pantalla. Condiciones *quality gates* SonarQube Server.

- Establecer los umbrales para cada métrica: una vez definidas las condiciones, es necesario establecer los umbrales para cada métrica. Estos umbrales indican el valor máximo o mínimo permitido para cada métrica. Por ejemplo, se podría establecer un umbral del 80% para la cobertura de pruebas unitarias, ver figura 70.

Figura 70. **Definición de valores a métricas en SonarQube**



Fuente: elaboración propia, captura de pantalla. Métricas *quality gates* SonarQube Server.

- Asignar el *quality gate* a un proyecto: finalmente, es necesario asignar el *quality gate* a un proyecto específico para que las condiciones establecidas sean evaluadas en el código de ese proyecto.

Figura 71. **Asociar *quality gates* a proyecto en SonarQube**

Fuente: elaboración propia, captura de pantalla. *Quality gates* SonarQube Server.

Es importante destacar que la configuración de los *quality gates* puede variar en función del proyecto y de los objetivos específicos de calidad que se quieran alcanzar. Además, SonarQube permite la creación de múltiples *quality gate*s, lo que permite establecer diferentes condiciones para distintos proyectos o fases de desarrollo.

2.5.4. Algunos patrones de diseño que mejor se adaptan a SonarQube

Los patrones de diseño de *software* que mejor se adaptan dependerán del lenguaje de programación y el tipo de proyecto en cuestión. Sin embargo, algunos patrones de diseño generales que pueden ayudar a mejorar la calidad del código y a facilitar la aplicación de SonarQube, estos son los siguientes

- Patrón de diseño de arquitectura limpia

- Patrón de diseño de inyección de dependencias
- Patrón de diseño de control de versiones
- Patrón de diseño de pruebas unitarias

En general, los patrones de diseño que mejor se adaptan al uso de SonarQube son aquellos que promueven el modularidad, la separación de responsabilidades, la facilidad de mantenimiento y la calidad de las pruebas.

2.5.5. ¿Qué evitar durante la utilización de SonarQube?

Algunas cosas que no se deben hacer o aplicar cuando se utiliza SonarQube incluyen lo siguiente:

- No ignorar los resultados de SonarQube: ignorar los resultados de SonarQube puede llevar a una mala calidad del código y problemas técnicos a largo plazo.

- No centrarse únicamente en los resultados de SonarQube: aunque SonarQube es una herramienta importante para garantizar la calidad del código, no debe ser el único criterio para evaluar el *software*.

- No utilizar SonarQube como herramienta de análisis de seguridad: aunque SonarQube puede proporcionar algunos indicadores de seguridad, no es una herramienta de análisis de seguridad completa y no debe utilizarse como tal.

- No utilizar SonarQube como una herramienta de seguimiento de errores: SonarQube no es una herramienta de seguimiento de errores, por lo que

no se deben utilizar sus resultados para identificar problemas específicos en el código.

- No confiar ciegamente en los resultados de SonarQube: los resultados de SonarQube deben ser interpretados y revisados cuidadosamente, ya que puede haber falsos positivos o falsos negativos.

- No olvidar la importancia de las pruebas: SonarQube puede identificar problemas en el código, pero no puede garantizar que el *software* funcione correctamente. Es importante realizar pruebas rigurosas para garantizar que el *software* cumpla con los requisitos y expectativas del usuario.

En resumen, SonarQube es una herramienta poderosa para mejorar la calidad del código en fase de desarrollo implementando análisis estático de código, pero no debe ser el único criterio para evaluar el *software* y se deben tomar en cuenta otros factores y herramientas complementarias para garantizar la calidad y seguridad del *software*.

CONCLUSIONES

1. La evaluación de la calidad del código en SonarQube es una herramienta muy útil para el mejoramiento continuo del software. Su implementación en un ambiente de integración continua o local permite identificar problemas de calidad en el código desde el principio del desarrollo, lo que reduce los costos y los tiempos de corrección.

2. La configuración adecuada de reglas y métricas en SonarQube, aplicada de manera consistente en diferentes proyectos, mejora significativamente la calidad del código y establecer un estándar de calidad en una organización o equipo. Al utilizar SonarQube para la inspección del código, se pueden identificar y abordar de manera proactiva problemas y deficiencias en el código fuente. Esto promueve una cultura de excelencia en el desarrollo de *software*, mejora la eficiencia y la mantenibilidad, y contribuye a la seguridad del *software* en general.

3. La retroalimentación constante sobre la calidad del código puede ayudar a los equipos de desarrollo a mejorar sus prácticas y tomar decisiones informadas sobre la implementación del *software*.

4. La mejora continua de la calidad del código debe ser un proceso constante y sistemático. Se deben establecer indicadores de calidad y realizar mediciones periódicas durante cada solicitud de cambios al proyecto para evaluar el progreso y detectar posibles problemas.

5. Utilizar SonarLint en un entorno de desarrollo integrado (IDE) para realizar análisis estático de código en tiempo real proporciona una guía instantánea a los desarrolladores, permitiéndoles identificar y corregir problemas de calidad del código de manera eficiente durante el proceso de desarrollo.

6. La configuración de pipelines en Azure DevOps que integre SonarQube y ejecute análisis de código de forma automática a través de solicitudes de cambios hacia código fuente, garantiza una revisión continua de la calidad del código y proporciona retroalimentación inmediata a los desarrolladores, fomentando la mejora continua del *software*.

RECOMENDACIONES

1. Analizar regularmente las métricas de SonarQube y tomar acciones para mejorar el código.

2. Utilizar SonarQube de manera regular para evaluar la calidad del código y detectar oportunidades de mejora.

3. Realizar pruebas unitarias en todo el código para detectar errores y mejorar su calidad de código.

4. Seguir los patrones y estándares de lenguajes de programación para garantizar la coherencia y la facilidad de mantenimiento del código.

5. Educar y capacitar a los desarrolladores sobre la importancia de la calidad del código y cómo mejorarla mediante el uso de herramientas como SonarQube.

6. Realizar una planificación detallada y asignar roles y responsabilidades a los miembros del equipo. Documentar de manera exhaustiva el proceso de integración, incluyendo los pasos y configuraciones necesarios, para asegurar una integración exitosa de SonarQube con Azure DevOps.

REFERENCIAS

1. Alcolea, C. D. (2019). *OpenWebinars*. Obtenido de Code Smells y deuda técnica: Recuperado de https://openwebinars.net/blog/code-smells-y-deuda-tecnica/

2. Bosch, J. (2000). *Design and Use of Software Architectures.* Harlow, UK: Addison-Wesley.

3. Carvajal Riola, J. (2008). *Metodologías ágiles: herramientas y modelo de desarrollo para aplicaciones.* Barcelona: UPC.

4. EKON. (2020). *Entornos de desarrollo: todo lo que sucede en el desarrollo de software*. Recuperado de Ekon: https://www.ekon.es/entornos-desarrollo-software/

5. Galin, D. (2003). *Software Quality Assurance From theory to implementation.* Israel: Pearson Addison Wesley.

6. Garlan, D., & Shaw, M. (1993). *An introduction to software architecture. Advances in Software Engineering and Knowledge Engineering.*

7. Glass, Robert. L. (2002). *Facts and Fallacies of Software Engineering.* Pearson Education, Inc.

8. IEEE The Institute of Electrical and Electronics. (1991). *IEEE Std 610.12-1990– IEEE Standard Glossary of Software.* New York.

9. Microsoft Azure. (2021). *¿Qué es DevOps?* Recuperado de: https://azure.microsoft.com/es-es/overview/what-is-devops/#devops-overview

10. Pons, C. Roxana, G. y Pérez, G. (2010). *Desarrollo de software dirigido por modelos. Teorías, metodologías y herramientas.* McGraw-Hill Education.

11. Sommerville, Ian. (2005). *Ingeniería del software (Vol. VII).* Madrid: Pearson Educación, S. A.

12. Sommerville, Ian. (2011). *Ingeniería de software* (Vol. IX). México: Pearson Educación.

13. SonarScanner. (2021). *Sonar-Scanner*. Recuperado de: https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/.

14. SonarQube inria. (2020). *SonarQube*. Recuperado de: SonarQube Inria: https://sonarqube.inria.fr/sonarqube/documentation/.

15. SonarQube S. A. (2021). Recuperado de SonarQube: https://docs.sonarqube.org/8.9/

16. SonarSource. (2008). *Sonar Rules*. Recuperado de https://rules.sonarsource.com/.

17. SonarSource. (2021). *SonarLint.org*. Recuperado de Sonarlint.org Site: https://www.sonarlint.org/.