POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Analisi Statica del codice e gestione del technical debt con Sonarqube

Relatori:

Prof. Giovanni MALNATI

Dott. Hamza RHAOUATI

Candidato:

Alessio BINCOLETTO

Anno Accademico 2022/2023
Torino

Abstract

Nel contesto dello sviluppo di prodotti software con funzionalità che richiedono aggiornamenti e manutenzione nel lungo termine, la gestione del debito tecnico svolge un ruolo fondamentale per garantire la sostenibilità di questo processo. Tuttavia, il debito tecnico presenta sfaccettature complesse che rendono difficile per i tecnici individuarne le cause e comunicare le difficoltà e i problemi correlati allo sviluppo del prodotto al management e al personale aziendale non direttamente coinvolto nello sviluppo. L'analisi statica del codice è una tecnica che consente di misurare e presentare in modo strutturato metriche utili per individuare i problemi all'interno di una codebase, comprenderne la natura e stimare l'effort necessario per risolverli o riportarli a un livello sostenibile. SonarQube è uno strumento per l'analisi statica e la presentazione dei dati estratti agli operatori umani. Si differenzia dagli altri tool per la sua capacità di integrare informazioni da report di terze parti, per il suo ampio database di regole di analisi per i linguaggi di programmazione e i framework più comuni, nonché per il suo orientamento alla gestione del debito tecnico durante lo sviluppo. L'obiettivo di questa tesi è l'integrazione di SonarQube con gli strumenti di CI/CD (Jenkins, GitlabCI) in un azienda che gestisce progetti di diverse dimensioni (da kLOC a MLOC) e con diverse tecnologie, esplorndo un approccio di introduzione incrementale del tool nel processo di sviluppo per favorirne l'uso da parte dei developer.

Ringraziamenti

Grazie a tutti coloro che mi hanno sostenuto in questi anni e nella realizzazione di questa tesi.

Table of Contents

List of Figures

Acronyms

CI/CD
: Continuous Integration Continuous Development/Continuous Deployment

CISA
: Cybersecurity and Infrastructure Security Agency

CLI
: Command Line Interface

CVE
: Common Vulnerabilities and Exposures

DevOps
: Development and Operations

DSL
: Domain Specific Language

GTD
: Generic Test Data

JVM
: Java Virtual Machine

KEV
: Known Exploited Vulnerabilities

LCOV
: Line COVerage

LOC

Lines Of Code

NIST

National Institute of Standards and Technology

NVD

National Vulnerability Database

OWASP

Open Worldwide Application Security Project

VCS

Version Control System

VM

Virtual Machine

XML

Extensible Markup Language

YAML

Yet Another Markup Language

Chapter 1

Introduzione

In un contesto di sviluppo di prodotti software le cui funzionalità vengono aggiornate e devono essere mantenute per anni o decenni, la gestione del techincal debt ricopre un ruolo fondamentale nell'assicurare la sostenibilità di questo processo. Uno dei problemi principali del debito tecnico è la sua natura complessa e multi-sfaccettata da cui consegue una forte difficoltà da parte di coloro che lo percepiscono (ovvero i tecnici) da un lato di individuarne le origini per agire in direzione di una soluzione, dall'altro di poter comunicare al management e al personale aziendale non direttamente coinvolto negli aspetti tecnici le difficoltà e i problemi relativi allo sviluppo del prodotto. L'analisi statica del codice è una tecnica che permette di misurare e presentare in modo ordinato delle metriche per fornire in modo efficace delle direttive sul dove si trovano i problemi all'interno di una codebase, quanti sono, qual' è la loro natura e quanto effort (in termini di ore di lavoro) è stimato come necessario per liberarsene o per riportarlo ad una soglia considerata sostenibile. L'analisi statica del codice è dunque un processo automatizzato che può essere effettuato con diversi strumenti per ottenere diversi tipi o profondità di analisi a seconda della complessità dello strumento. In un ottica di automatizzazione dei processi andata delineandosi negli ultimi decenni e che ha preso forma con la nascita di figure professionali ibride come i DevOps e i più recenti DevSecOps, sono diventati sempre più popolari e importanti strumenti per la CI/CD (Continuous Integration and Continuous Delivery/Continuous Deployment) come Jenkins, GitlabCI, GithubActions, CircleCI e tanti altri. E' dunque in questo contesto che si può collocare l'uso degli strumenti di analisi statica del codice e in particolare l'uso di SonarQube il quale è de-facto il più avanzato e diffuso strumento nell'ambito dell'analisi statica e della presentazione dei dati estratti ad operatori umani. SonarQube differisce dagli altri strumenti di analisi statica per la

sua ampia capacità di integrazione di informazioni provenienti da report prodotti con strumenti di terze parti, per il suo ampio database di regole di analisi per la maggior parte dei più comuni linguaggi di programmazione e frameworks e per il suo orientamento al fornire strumenti per tenere sotto controllo la crescita del debito tecnico durante lo sviluppo piuttosto che limitarsi al mostrare le metriche di analisi.

1.1 Obiettivo della tesi

L'integrazione di uno strumento come SonarQube nei processi di sviluppo di un azienda su multipli progetti già avviati da tempo può presentare due principali problemi, da un lato l'aspetto tecnico dell'integrazione dello strumento nei processi di CI/CD di progetti con stack tecnologici diversi tra loro, dall'altro l'aspetto umano: essendo sostanzialmente il ruolo di SonarQube quello di mettere in evidenza delle criticità nel codice prodotto dagli sviluppatori, è importante che venga loro presentato in modo che questi ultimi non lo percepiscano come una critica al loro lavoro ma piuttosto come uno strumento a loro vantaggio. Questa tesi si propone come obiettivo l'integrazione di SonarQube con gli strumenti di CI/CD (Jenkins, GitlabCI) di progetti diversi tra loro nelle dimensioni (da kLOC a MLOC) e nelle tecnologie usate, in un modo non invasivo che permetta un graduale acquisto di fiducia nello strumento da parte degli sviluppatori favorendone l'utilizzo.

Chapter 2

Tecnologie usate

Questo capitolo provvede una descrizione degli elementi principali e delle tecnologie chiave coinvolte nello sviluppo di questa tesi. Dapprima vengono introdotti i sistemi di versioning del codice offerti da Subversion e Git, dopodichè si passa al concetto di continuous integration continuous development/continuous deployment e il ruolo di Jenkins, in seguito verranno esposte le caratteristiche e le funzionalità offerte dai due prodotti per l'analisi statica utilizzati in questa tesi: SonarQube e SonarLint, infine si procederà con il citare la containerizzazione con Docker.

2.1 Versioning

Il versioning (o version control) è un concetto fondamentale nello sviluppo software e si riferisce alla gestione delle diverse versioni di un progetto nel corso del tempo. Gli strumenti di versioning consentono agli sviluppatori di tenere traccia delle modifiche apportate al codice sorgente, di collaborare in modo efficace con altri membri del team e di agire in modo isolato su eventuali problemi del software.

Alcuni termini:

- **clone/checkout**: è l'atto di copiare il contenuto di un repository remoto sul proprio client in modo da poterci lavorare
- **commit**: è l'atto di salvare uno stato del software nella storia dello stesso in seguito a modifiche, aggiunte o rimozioni di codice o risorse. Lo scopo è salvare

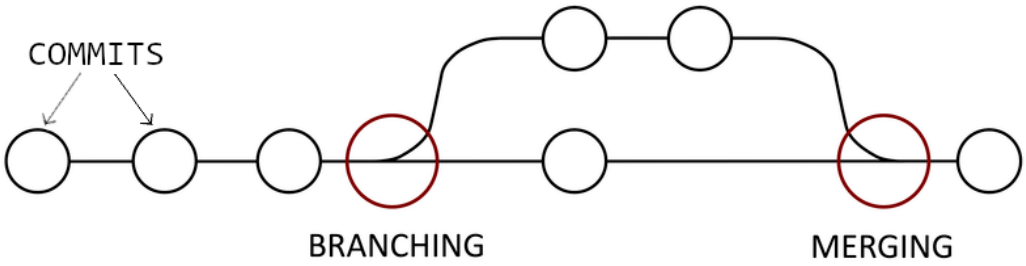

Figure 2.1: Processo di merge tra due branch [1]

questa versione per poterla usare come base per continuare lo sviluppo in futuro a partire da essa, descrivere in una timeline quali sono state le modifiche effettuate ai file e da chi sono state effettuate e soprattutto permettere la possibilità di riutilizzare specifiche versioni del progetto in futuro.

- **merge**: è l'atto di produzione di una nuova versione del progetto a partire dall'unione di due versioni differenti dello stesso, questa operazione è necessaria nel contesto del lavoro concorrente di più persone allo stesso progetto.

- **branch**: i branch o rami di un repository sono in realtà considerabili come repositories a se stanti aventi una commit history (quindi una successione storica delle versioni generata dai commit e dai merge) indipendente. Branch diversi di un progetto vengono creati in genere per sviluppare versioni differenti del prodotto o per salvare stati specifici come versioni di rilascio principali.

Esistono vari sistemi di versioning, ma i due maggiormente diffusi sono Git e Subversion:

- Git è un sistema di controllo di versione distribuito e permette agli sviluppatori di creare una copia del repository del progetto sul proprio computer locale contenente l'intera cronologia delle modifiche effettuate al codice sorgente, consentendo agli sviluppatori di lavorare in modo indipendente senza interferire con il lavoro degli altri portando avanti delle modifiche alla cronologia delle versioni del software (tramite i commit) la cui sincronizzazione con il repository remoto può essere delegata ad un secondo momento. Nel contesto aziendale i repository Git vengono gestiti solitamente gestiti tramite GitLab, una piattaforma che offre anche la possibilità di essere self-hosted, di integrarsi con altri strumenti comunemente usati in ambito aziendale come Slack.

- Apache Subversion invece è un sistema di versioning centralizzato, la differenza sostanziale è che non esiste una versione della storia del software locale ad ogni client, gli utenti possono però creare un branch di lavoro che rimane sempre comunque memorizzato nel server come "sotto-cartella" del repository principale e che potrà poi essere unito (merge) con quello principale. Ciò ha come conseguenza non poter gestire la history offline o comunque disconnessi dal server aziendale.

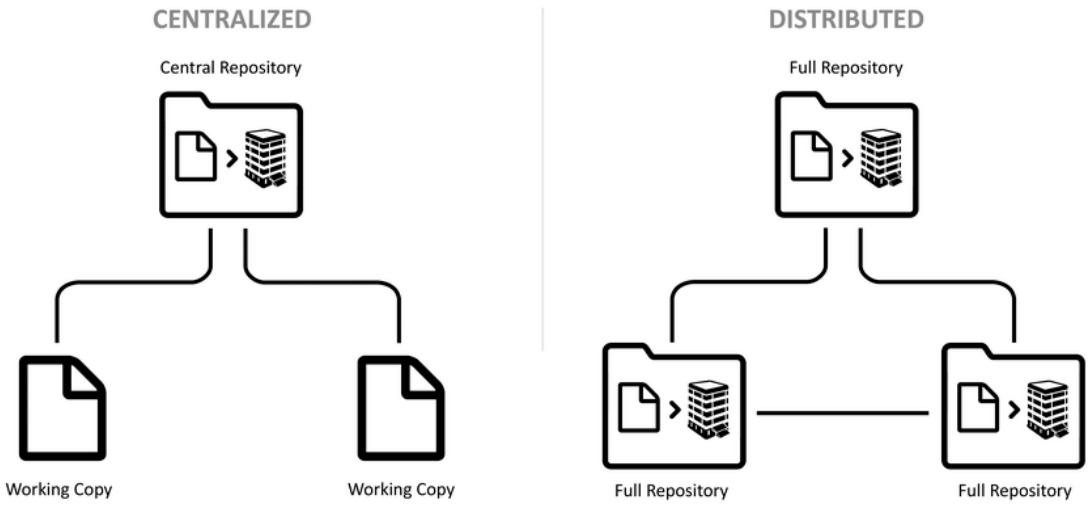

Figure 2.2: VCS centralizzato vs distribuito [1]

Il versioning dunque offre numerosi vantaggi. Innanzitutto, permette di tenere traccia delle modifiche specifiche apportate al codice sorgente. Ogni volta che viene effettuato un commit, viene creata una nuova versione che rappresenta uno snapshot del progetto in quel momento.

Inoltre agevola il lavoro collaborativo, più sviluppatori possono lavorare contemporaneamente sullo stesso progetto e sugli stessi file senza rischiare di sovrascriversi inconsapevolmente il lavoro a vicenda, permettendo l'automatizzazione dell'attività di merging quando le modifiche non coinvolgono le stesse aree di codice e fornendo invece meccanismi di scelta delle modifiche da mantenere o scartare quando necessario. Altro aspetto importante è la possibilità di identificare e risolvere bug o errori nel software. Se un errore viene introdotto in una nuova versione, è possibile risalire alle modifiche apportate precedentemente per individuare la causa e correggerlo.

Infine, il versioning consente anche di documentare il processo di sviluppo software generando una timeline del progetto che include informazioni sulle modifiche apportate, i problemi risolti e altre note di rilascio.

In conclusione, il versioning è fondamentale nello sviluppo software perché permette di tenere traccia delle modifiche, facilita il lavoro collaborativo, agevola la risoluzione di errori e fornisce una cronologia delle attività di sviluppo. È uno strumento essenziale per mantenere il controllo e la gestione efficace di un progetto software lungo il suo ciclo di vita.

2.2 CI/CD

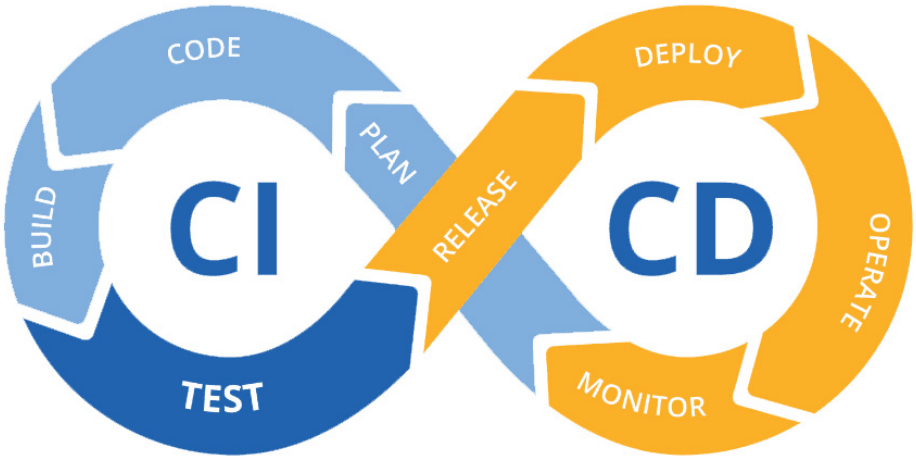

Figure 2.3: step di iterazione della produzione software [2]

Il processo di produzione software comprende molte attività oltre al semplice sviluppo di codice che richiedono effort e possono risultare complesse e inefficaci se svolte a mano e che in un ottica di sviluppo agile devono essere effettuate spesso a causa dei frequenti rilasci. Alcune di queste attività sono la verifica del funzionamento del codice ad ogni commit/push/merge effettuato sul progetto, il rispetto delle funzionalità richieste al codice descritte dai test, l'archiviazione degli

eseguibili/delle immagini degli applicativi prodotti, la verifica della loro sicurezza, la distribuzione (deploy) di questi ultimi agli utenti e il monitoraggio del technical debt. In queste attività sono coinvolti tipicamente più dipartimenti della stessa azienda, dal team di sviluppo (Dev) a quello operativo (Ops), per questa ragione negli ultimi decenni hanno preso sempre più piede strumenti che permettono l'automazione di queste attività tramite l'uso di pipeline CI/CD e le figure professionali associate a questi strumenti e a queste pratiche vengono identificate come DevOps.

Jenkins

Uno degli strumenti open source più noti per la creazione di pipeline CI/CD è il server di automazione Jenkins. Jenkins offre un ampissimo set di funzioni e plugin di terze parti e open source, inoltre è self-hosted il che permette un maggior controllo sui dati da parte delle aziende ed è compatibile con le più svariate tecnologie di repository hosting (sia subversion che GitLab ad esempio), è relativamente facile da installare e non richiede tecnologie specifiche essendo sviluppato in java e potendo dunque godere dei benefici della JVM: per tutte queste ragioni è un tool molto popolare tra le aziende.

Una considerazione però non ignorabile è che rispetto ad altri tool come GitlabCI, le Github Actions e altri competitor, insieme ad una molto più ampia libertà di azione da parte degli utenti presenta anche una curva di apprendimento più alta, dando possibilmente forma a situazioni in cui la semplicità delle operazioni da effettuare non giustifica la complessità dello strumento usato.

l'uso e il funzionamento base di Jenkins si può riassumere nei seguenti punti:

1. Si genera una nuova pipeline dall'interfaccia di Jenkins impostandolo in modo da "guardare" un repository in attesa di eventi che ne producano cambiamenti come commit, push o merge.

2. Nella root del repository si genera un file nominato Jenkinsfile: in questo file si descrive la pipeline che verrà effettuata dal server Jenkins.

    - la pipeline è divisa in multipli "stage" ovvero unità di evento, per ognuna delle quali può essere specificato un agent ovvero il processo e la macchina fisica che devono eseguire le istruzioni contenute in quello stage.
    - stage di esempio possono essere quello di Build, quello di Test e quello di Deploy

- le pipeline possono essere descritte tramite due diversi DSL (domain specific language), uno simile a Groovy chiamato Advanced Scripted Pipeline, con una sintassi imperativa e che permette l'iniezione di codice Groovy o java, oppure un linguaggio di scripting dichiarativo chiamato "Declarative Pipeline" di più recente rilascio, che non permette iniezione di codice obbligando quindi i developer a sfruttare le funzioni già esistenti a loro disposizione.
- singoli stage o intere pipeline possono essere eseguiti su container indicando un immagine docker e predisponendo adeguatamente i nodi

3. nel momento in cui il server Jenkins rileva un evento per cui era in attesa (ad esempio un commit) fa il checkout(o clone) del progetto in locale, dopodiché legge il Jenkinsfile nella root ed esegue in ordine le attività descritte nella pipeline, abortendone l'esecuzione al primo fallimento di uno stage.

4. la sequenza di definizione degli stage è fondamentale, infatti posizionare lo stage di Test in cui viene effettuato il run di tutti i test disponibili prima della fase di deploy, permette di non effettuare il deploy di software non compliant con la test suite grazie al blocco della pipeline in seguito al fallimento di uno o più test case.

Jenkins ha un architettura distribuita, ovvero presenta un master node che si occupa della gestione delle pipeline e dell'interfaccia utente e multipli nodi che possono essere utilizzati per il run delle istruzioni contenute negli stage. I nodi possono avere caratteristiche diverse, come sistema operativo, architettura hardware della macchina ospitante e disponibilità o meno di servizi software come ad esempio il docker daemon. Ciò permette da una parte una più articolata distribuzione del carico di lavoro, dall'altra la possibilità di definire ambienti di test specifici per sistemi operativi o versioni di software come i browser.

GitLab CI

GitlabCI è lo strumento di gestione pipelines integrato nella piattaforma di versioning GitLab. Esso è meno articolato di Jenkins essendo vincolato alla piattaforma, ma gode comunque di un largo set di funzionalità. Le pipeline sono descritte tramite file YAML e non permettono iniezione di codice. GitlabCI ha in comunque con Jenkins la gestione distribuita degli stage della pipeline e la possibilità di utilizzare immagini docker per l'esecuzione.

2.3 SonarQube

SonarQube è uno strumento di analisi statica del codice integrabile nelle CI pipeline (come Jenkins o GitlabCI) per identificare i problemi presenti in un progetto, e monitorarne l'introduzione/la risoluzione ad ogni push o merge in modo automatico, esso può inoltre processare i report prodotti da altri tool di analisi statica (ad esempio Checkstyle o FindBugs) o dinamica (ad esempio i tool per la computazione della coverage) per integrarne i risultati nelle metriche esposte attraverso la sua interfaccia.

2.3.1 Componenti

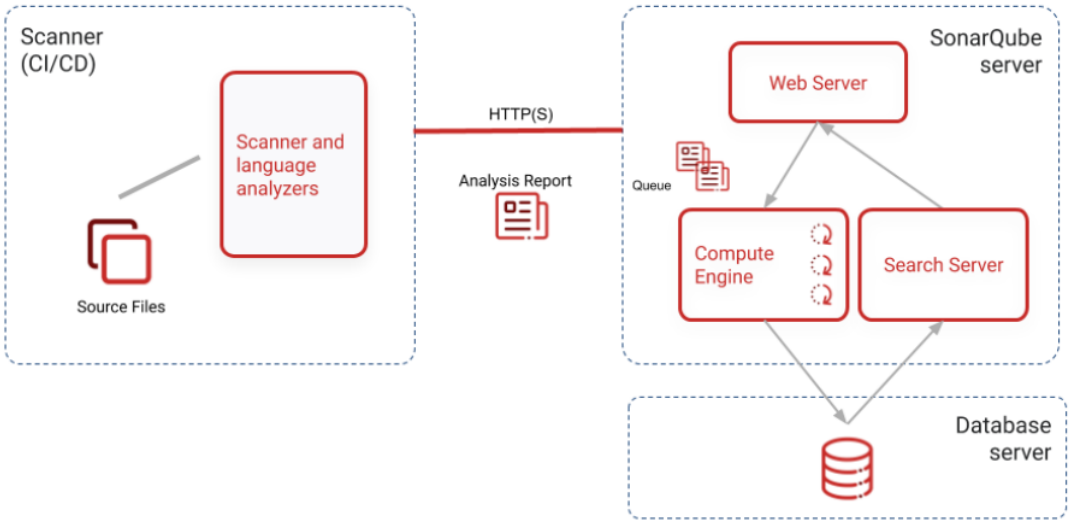

Figure 2.4: Componenti di SonarQube [3]

SonarQube si compone di tre parti principali:

1. Uno o più **scanner** che vengono eseguiti nel server di CI/CD o localmente al pc di uno sviluppatore.
2. Il **SonarQube Server** che esegue i seguenti processi:
   - Il web server che serve l'interfaccia utente di SonarQube
   - Un server di ricerca basato su ElasticSearch

- Il motore di computazione che ha il compito di processare i report delle analisi e di salvarli nel SonarQube database

3. Il **database** per lo storage di metriche e issues per la qualità e la sicurezza del codice generate durante la scansione del codice e i dati relativi alla configurazione dell'istanza di SonarQube.

Esiste inoltre la possibilità di installare plugin sul SonarQube server per ottenere ulteriori funzionalità. Allo stato attuale i plugin sono esclusivamente di terze parti dunque il funzionamento e il loro mantenimento non è garantito da SonarSource.

2.3.2 Scanners

Gli scanner possono essere di terze parti, che generano report in formati standard come XML, GTD (Generic Test Data), LCOV (Line Coverage) oppure di SonarQube, che si occupano di leggere il codice e analizzarlo applicandovi un set di regole che varia di linguaggio in linguaggio e che collezionano i report generati dagli scanner di terze parti. Il report finale generato dagli scanner di SonarQube contenente il codice sorgente e le informazioni generate dall'analisi viene poi spedito al SonarQube Server.

Gli scanner di Sonar possono essere:

- Lo scanner nella sua forma CLI: SonarScanner
- Plugin per ambienti di sviluppo come:
  - SonarScanner per Gradle
  - SonarScanner per .NET
  - SonarScanner per Maven
- Estensioni per strumenti di CI/CD come:
  - Estensione SonarQube per Jenkins
  - Estensione SonarQube per Azure DevOps

Gli scanner possono modificare il loro comportamento in esecuzione sulla base di parametri che possono essere comunicati in multipli modi diversi tra loro a seconda della tipologia di scanner, questi modi sono principalmente:

- Impostare i valori **da linea di comando**, ad esempio per impostare l'indirizzo del Server a cui spedire il report impostabile tramite la proprietà *sonar.host.url*, posso lanciare il comando di scan precedendo la proprietà con -D: sonar-scan -Dsonar.host.url=http://server.di.esempio.it

- Impostare i valori settandoli in un **file sonar-project.properties** situato nella root del progetto: sonar.host.url=http://server.di.esempio.it

- Utilizzare i metodi specifici di **setting degli ambienti di sviluppo** nel caso si stia utilizzando un plugin, dunque ad esempio nel caso del SonarScanner per Maven lo aggiungeremmo alle properties nel file pom.xml: <sonar.host.url>http://server.di.es

2.3.3 SonarQube Server

Si presenta agli utenti come un portale online raggiungibile ad un certo indirizzo web. La pagina principale contiene una lista di progetti, ogni progetto su SonarQube viene identificato da un valore ProjectKey che permette allo scanner di utilizzarlo come target quando viene inviato il report dell'analisi. Ogni progetto sul Sonar Server dunque corrisponde all'analisi di un progetto software, idealmente all'analisi di un repository, infatti lo stesso progetto sonar può collezionare l'analisi di più branch.

2.3.4 Sezioni Progetto Sonar

Overview.

Ogni progetto su sonar presenta come pagina principale un overview delle principali metriche estratte di interesse disponibili con due viste possibili: una detta "Overall Code" [2.5] e l'altra detta "New Code" [2.6]

Elementi di interesse:

- **Bug e Vulnerabilità**: sono da intendere come **possibili** bug e vulnerabilità, ovvero punti del codice che sono stati ricondotti dal motore di analisi di SonarQube a delle **regole di analisi** classificate come bug o vulnerabilità. Essi sono distinti in oltre in diversi livelli di gravità, dal meno grave al più grave sono: minor, major, critical e blocker.

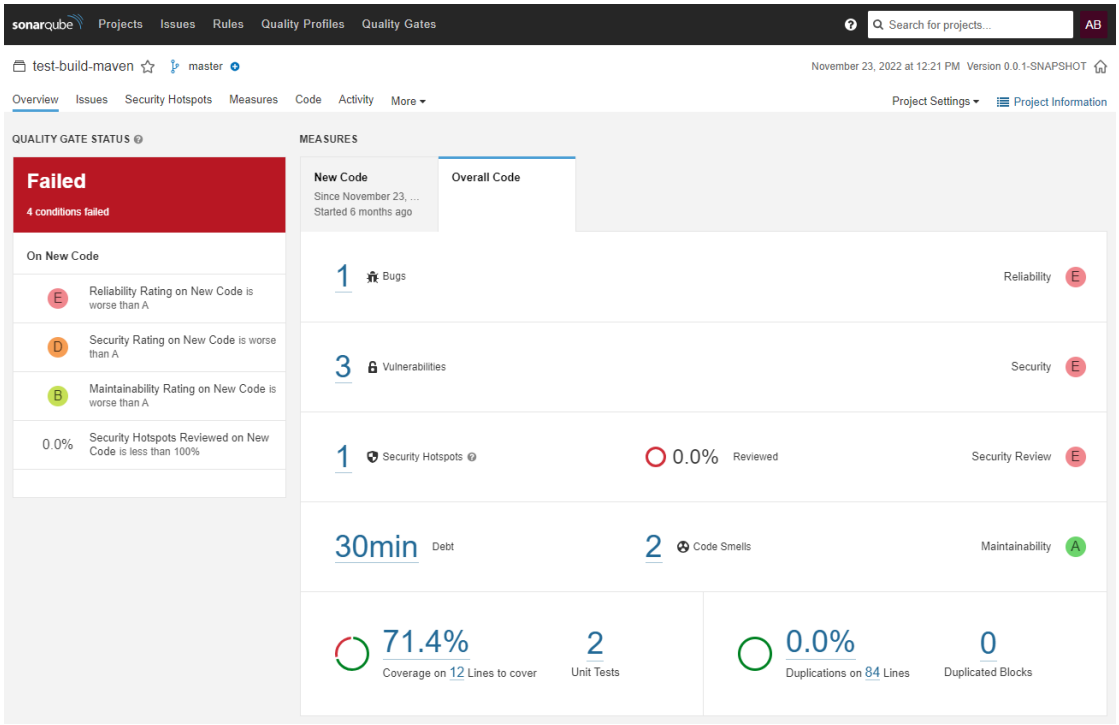

Figure 2.5: Sonar project overview: Overall Code

- **Security Hotspots**: Punti del codice identificati come potenziali problemi di sicurezza (query sql formate dinamicamente, campi «password» a cui vengono assegnate delle stringhe statiche, ecc...)
- **Code Smells**: Problemi di stile nel codice scritto, (codice commentato, funzioni con troppi argomenti, classi/funzioni senza javadoc associata, ecc...)
- **Coverage**: Linee di codice testabile (quindi non commenti, annotazioni, ecc...) che sono toccate nell'esecuzione di almeno un test automatizzato
- **Duplications**: Linee di codice identiche tra loro: con il crescere delle duplicazioni diminuisce la manutenibilità del codice
- **Debt**: Effort stimato in termini di tempo per effettuare degli interventi di manutenzione del codice utili a portare le metriche allo stato ideale (0 bug, 0 vulnerabillità, 100% review dei security hotspots, 0 code smells, 100% coverage, 0% duplications)

Le metriche inoltre riportano delle indicazioni qualitative espresse con lettere che vanno da A ad E per indicare progressivamente quanto è grave un certo dato.

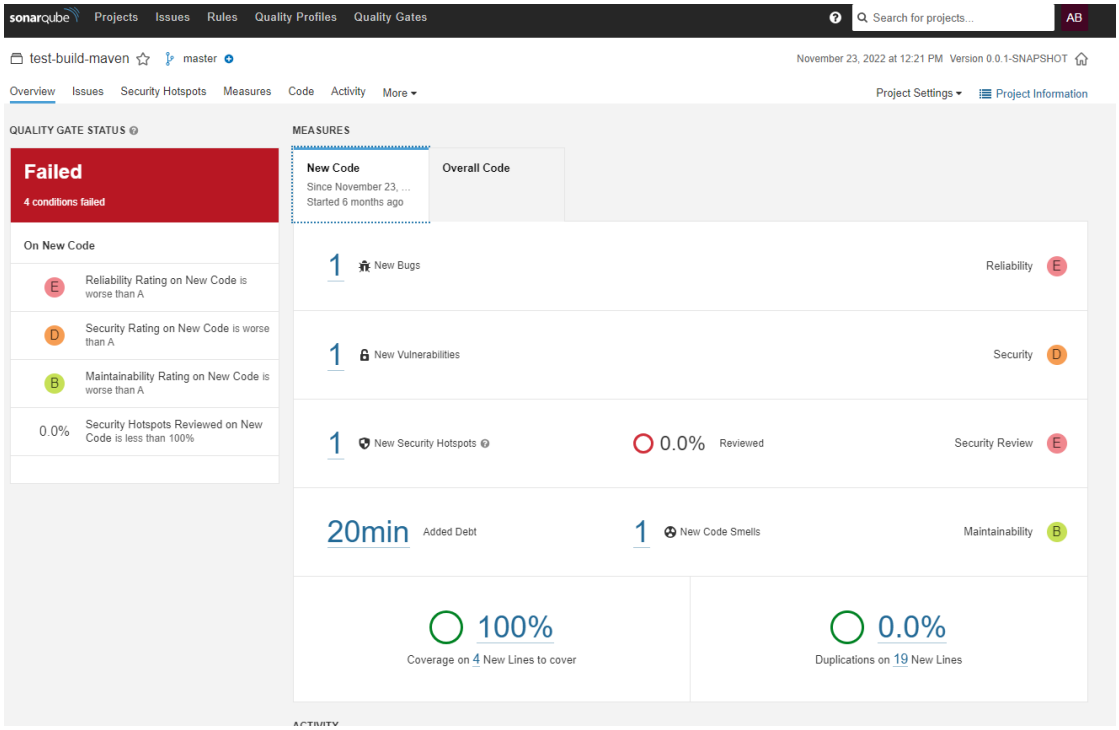

Figure 2.6: Sonar project overview: New Code

Per quanto riguarda i security hostspots, la metrica qualitativa fa riferimento alla percentuale di review degli stessi piuttosto che al numero di security hotspots presenti.

La differenza tra la vista "Overall code" e quella "New Code" sta nella porzione di codice su cui vengono computate le metriche. Mentre la Overall Code presenta delle metriche relative all'intero progetto, la sezione New Code fa riferimento al codice che viene aggiunto, discriminandolo dal resto del codice secondo diverse possibili definizioni di "New Code" [2.7].

Le possibili definizioni sono:

- **Previous Version**: Per tenere traccia del variare della qualità del codice usando come riferimento le modifiche apportate dall'ultima modifica fatta al numero di versione del progetto
- **Number of Days**: Il new code inteso come floating window temporale
- **Specific analysis**: La cronologia delle analisi effettuate viene salvata nel database, si può dunque prendere come riferimento una specifica analisi passata per monitorare l'andamento della qualità del codice a partire da un certo momento in poi.

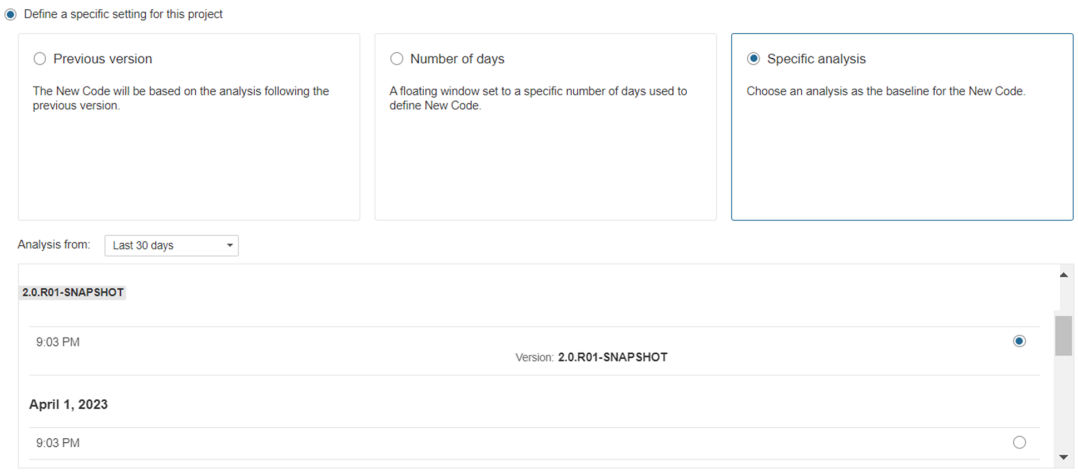

Figure 2.7: impostazioni per la definizione del New Code

Nella fascia sinistra della pagina di overview, sia in [2.5] che in [2.6] inoltre è presente una colonna che indica lo stato (Passed o Failed) del **Quality Gate** [2.18] seguito dalle regole che non vengono rispettate in caso lo stato fosse "Failed".

Issues

Ad ogni problema individuato sonar crea automaticamente una nuova issue [2.8] e [2.9], alla quale viene associato il codice a cui fa riferimento, una spiegazione del problema identificato, un indicazione qualitativa della gravità dello stesso, un tempo stimato di risoluzione, e spesso una spiegazione con esempi su possibili metodi risolutivi [2.10].

Le issues possono essere viste applicando filtri ad ogni metrica, favorendone l'uso nel processo di code review, inoltre le issues possono essere assegnate a membri del team in modo da distribuire e progettare gli interventi di bugfixing e refactor da effettuare dopo la review.

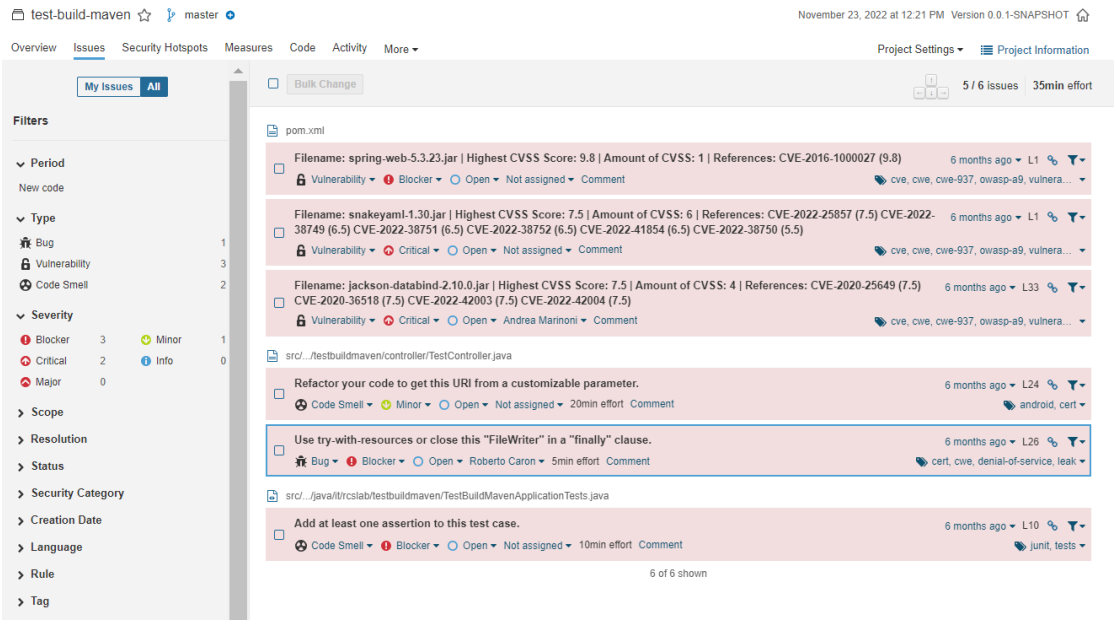

Figure 2.8: lista delle Issues di un progetto

Security Hotspots

Anche i security hotspots vengono organizzati in una lista la quale è poi divisa in tre sezioni di priorità di revisione: low, medium e high. Per ogni security hotspot è possibile visualizzare dove si trova il problema e per quale ragione il pattern riconosciuto potrebbe rappresentare un problema. Essendo catalogati come Security Hotspots pezzi di codice riconosciuti da dei pattern più complessi

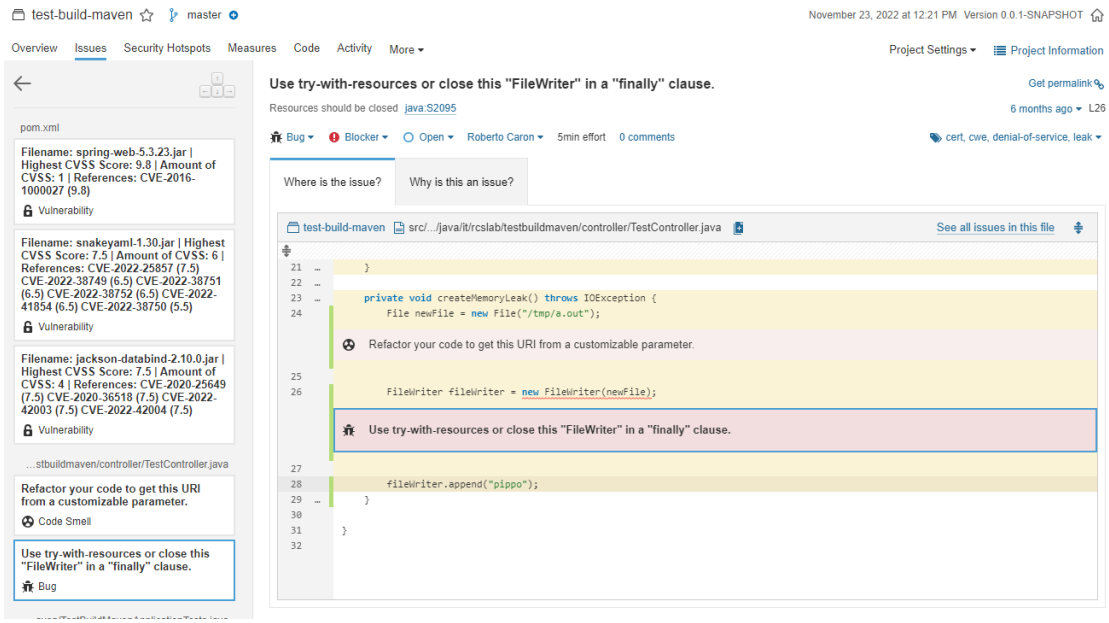

Figure 2.9: Galleria dei bug con dettagli

di quelli tipicamente riconducibili a Bug o Vulnerabilità, questi spesso vanno contestualizzati e riconosciuti come tali da un operatore umano, per questa ragione esiste la possibilità di catalogare i Security Hotspots sulla base del loro stato di revisione [2.11]

Measures.

È possibile esplorare il progetto tramite un approccio focalizzato sulle misure, ad esempio nell'immagine [2.12] si sta esplorando il progetto guardandolo con vista ad albero di direttori focalizzandosi sulla coverage. Questo approccio può essere utile per capire se ci sono parti del nostro progetto in cui si concentrano i problemi, aiutando a decidere se riscriverli, dismetterli o sostituirli.

Activity

Permette di visualizzare l'attività che influenza le metriche del progetto nel tempo e nel susseguirsi delle versioni del nostro progetto. [2.13]

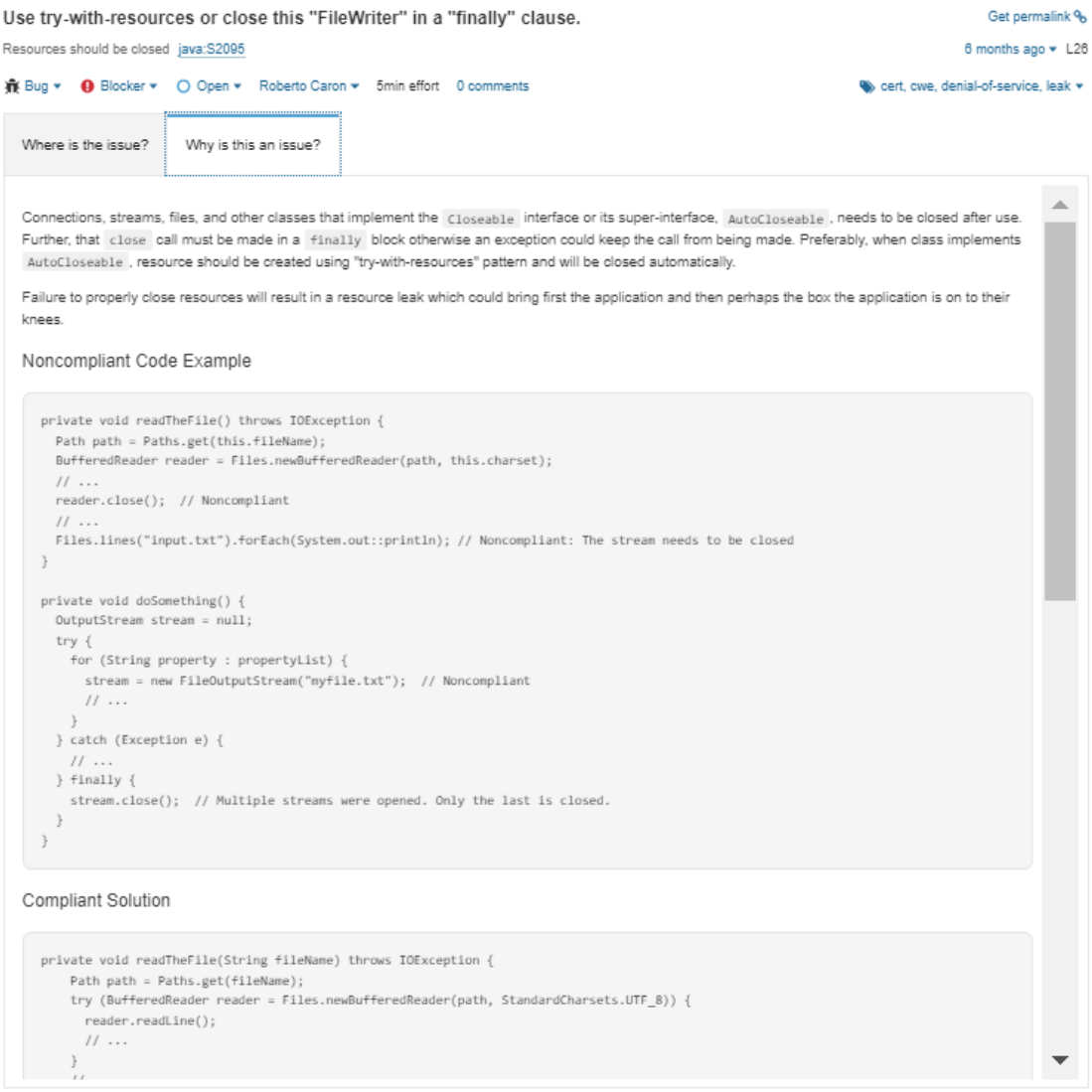

Figure 2.10: Spiegazione di una issue

Branch Analysis

É una feature disponibile solo con le licenze a pagamento del software, permette di raccogliere in un unico progetto sonar l'analisi di tutti i branch analizzati di un progetto/repository, permettendo il monitoraggio della qualità del codice in tutte le sue versioni. [2.14]

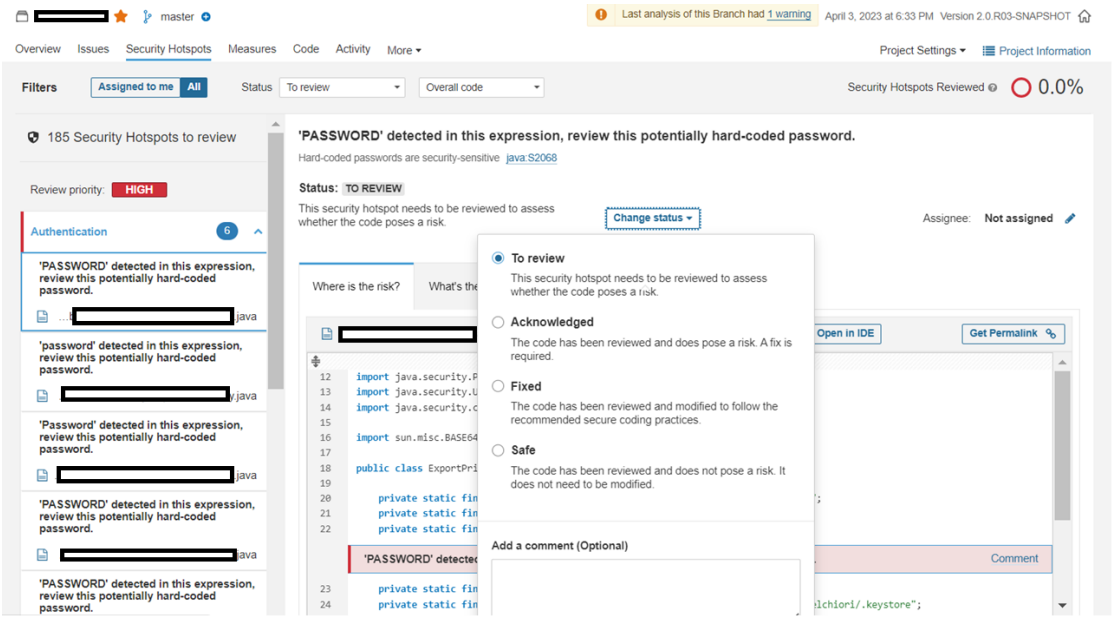

Figure 2.11: cambiare lo stato di un security hotspot

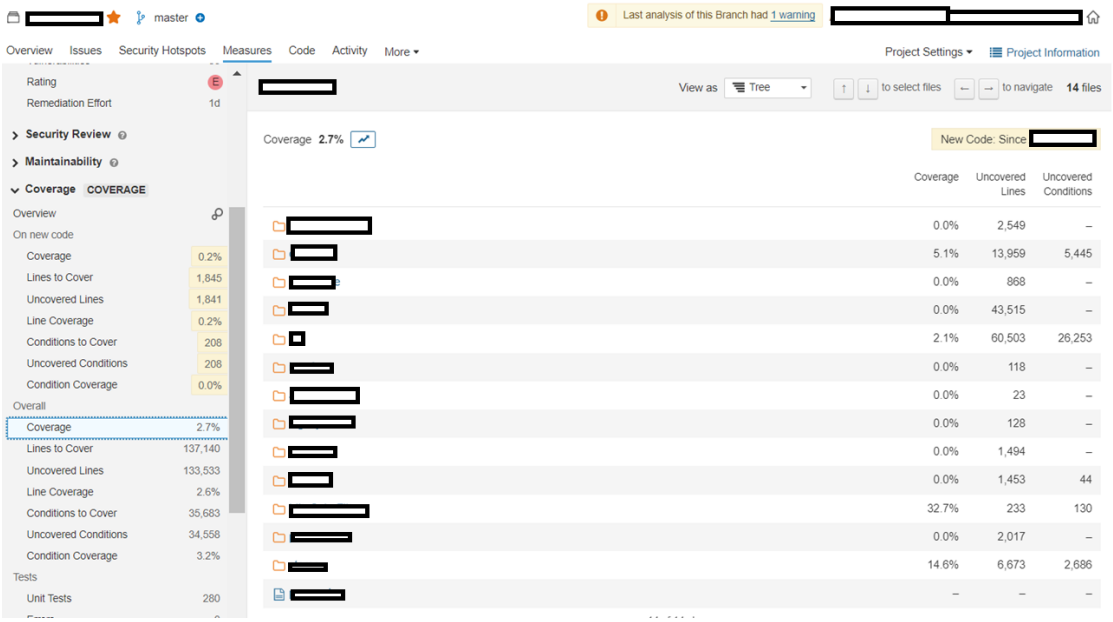

Figure 2.12: visualizzazione della distribuzione della coverage tra i moduli

Taint Analysis

Analisi orientata alla sicurezza che segue i dati che vengono immessi nel codice dall'esterno (da utenti, da richieste http, ecc..) e ne segue il percorso fino ai loro

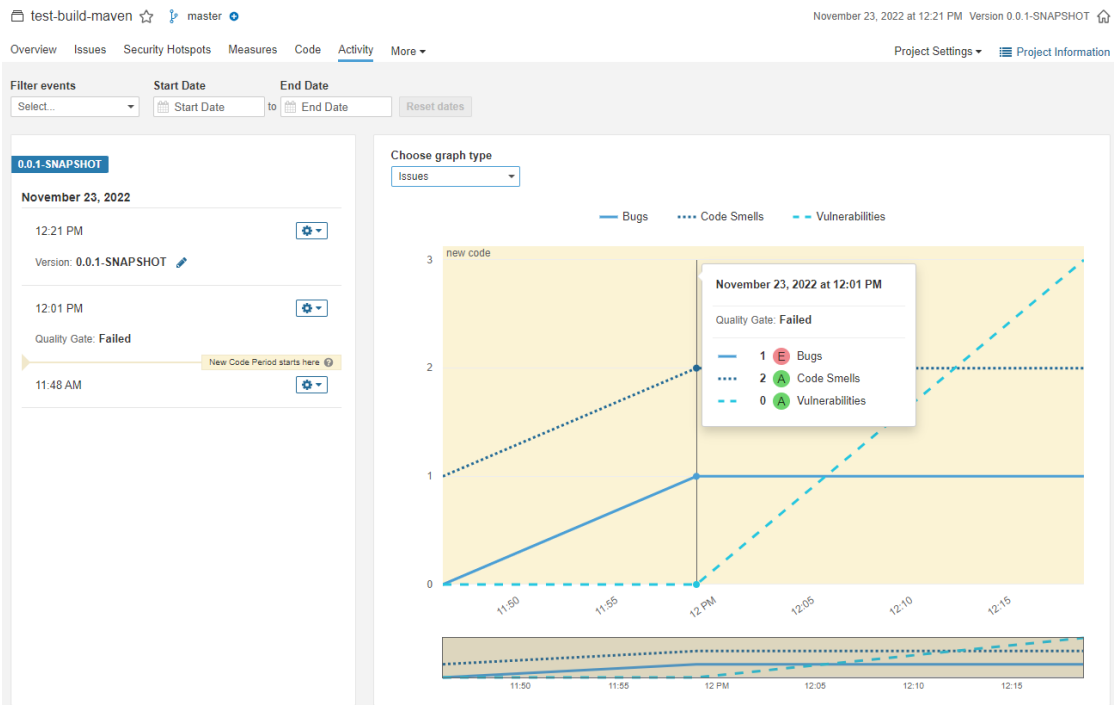

Figure 2.13: visualizzare la variazione delle metriche nel tempo

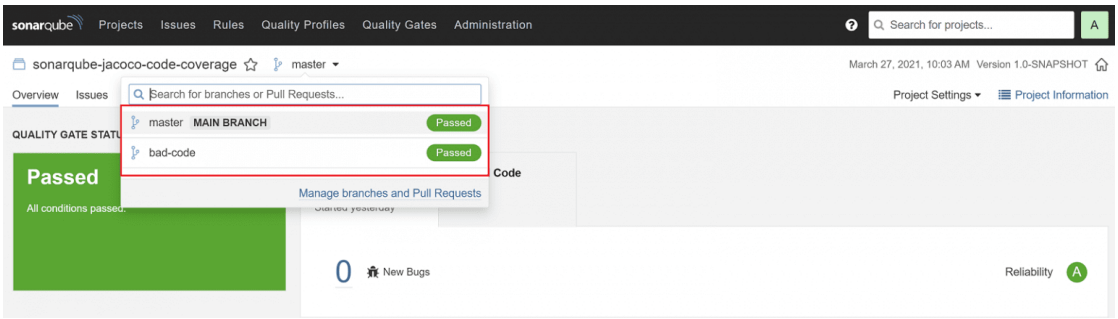

Figure 2.14: Branch analysis
[4]

punti di utilizzo per assicurarsi che vengano sanificati e siano sicuri 2.15

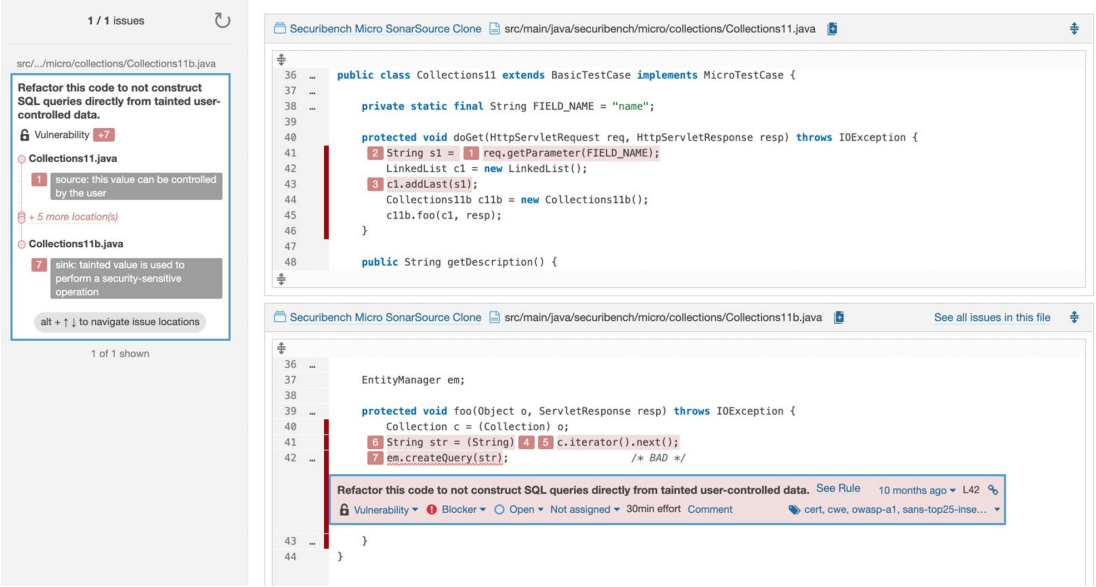

Figure 2.15: Esempio di taint analysis [5]

2.3.5 Rules

Il sonar engine elabora il report prodotto e inviatogli dal sonar scanner cercando pattern che facciano un match con delle "**Rules**". Le Rules sono descrizioni di pattern di codice, ogni regola descrive un bug, un code smell, una vulnerabilità o un security hotspot. Ogni regola poi è associata all informazioni tipicamente di interesse per quel genere di issue, come una spiegazione sul perchè rappresenti un problema, come questo potrebbe essere risolto e un livello di rischio rappresentato qualitativamente su una scala da Minor a Blocker. Ogni regola può fare riferimento ad uno specifico linguaggio di programmazione e può essere etichettata per categorizzarla [2.16]

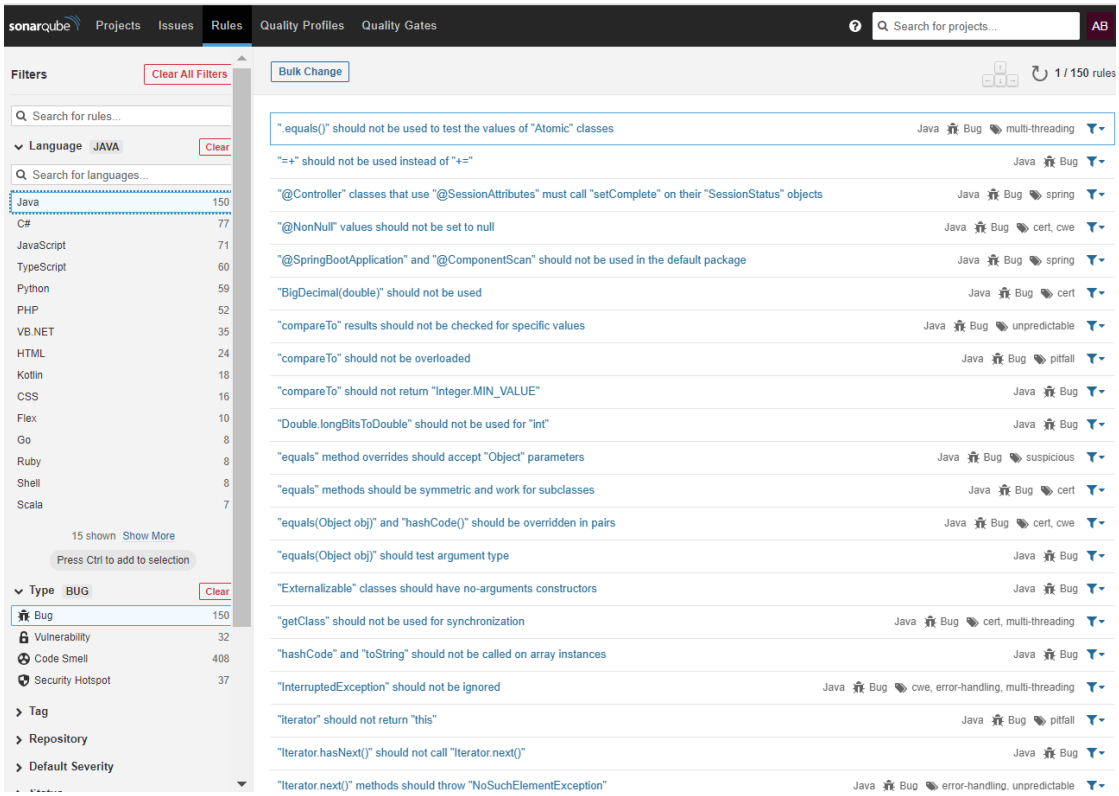

Figure 2.16: Rules Section: rules filtrate per linguaggio = "Java" e tipologia = "Bug"

2.3.6 Quality Profiles

I Quality Profiles sono gli insiemi di rules specifiche per certi linguaggi che possono essere applicati ad un analisi. Per ogni linguaggio è definito un quality profile standard comprensivo di tutte le rules disponibili per quel linguaggio ma nuovi profili possono essere definiti dagli utenti. [2.17]

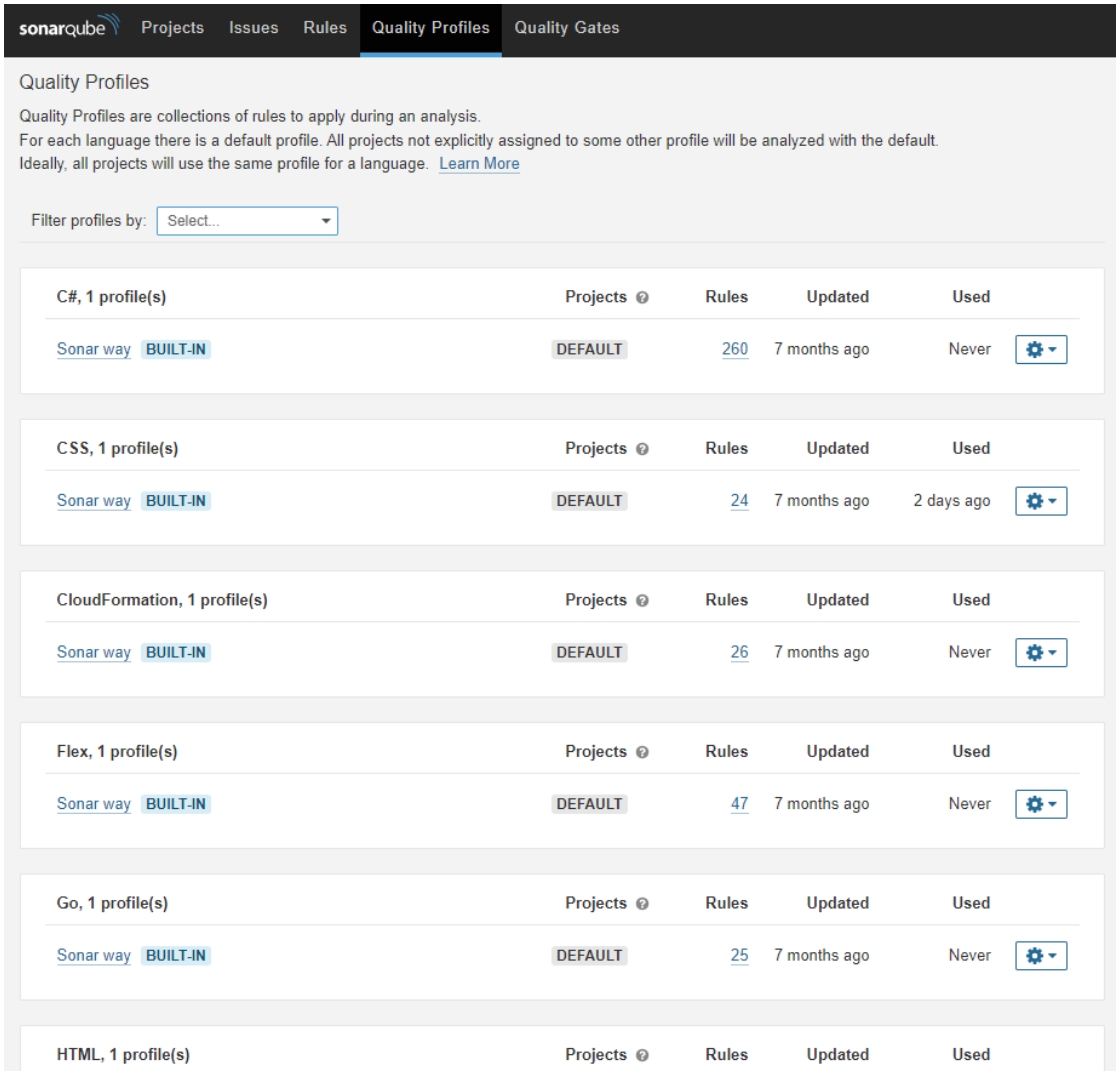

Figure 2.17: Quality Profiles section

2.3.7 Quality Gate.

Il Quality Gate è un set di specifiche facenti riferimento alle misure di analisi del progetto che il team vuole considerare come soglia limite di qualità. Queste specifiche possono essere definite separatamente sia sulle misure relative al new code sia su quelle relative all'overall code. È inoltre possibile definire multipli quality gate e applicare gli stessi a multipli progetti sonar. [2.18]

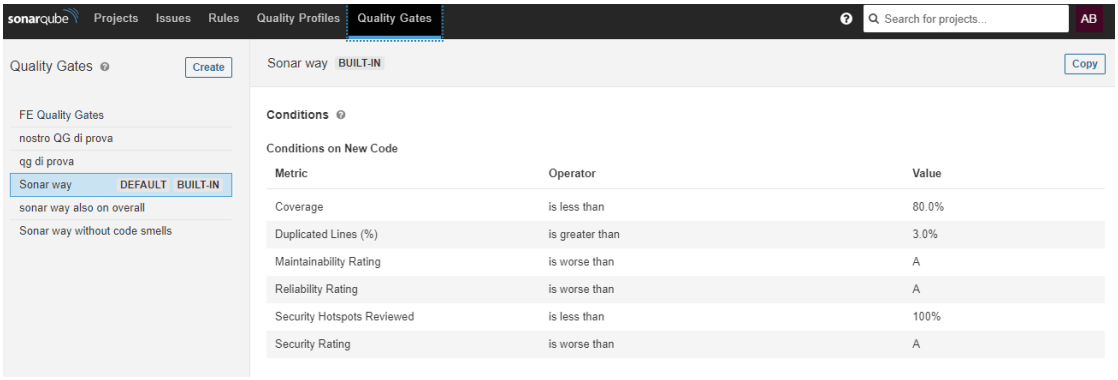

Figure 2.18: Impostazioni dei Quality Gates e regole default del QG "Sonar way"

2.3.8 Analyzers Aggiuntivi

Per ottenere ulteriori informazioni dalle analisi dei progetti, ho usato due tool di analisi di terze parti: Checkstyle e OWASP Dependency-Check.

OWASP Dependency-Check

OWASP è un acronimo che sta per Open Worldwide Application Security Project [6] è un progetto open-source che ha l'obiettivo di realizzare linee guida, strumenti e metodologie per migliorare la sicurezza delle applicazioni web e non. Dependency-check è un tool mantenuto da questa organizzazione che ha come obiettivo il l'analisi delle innumerevoli dipendenze di un progetto per cercarne menzioni nei principali database riguardanti le vulnerabilità:

- CVE (Common Vulnerabilities and Exposures) mantenuti e pubblicati dal MITRE [7]
- KEV (Known Vulnerability Issues) mantenuti dal CISA [8]
- NVD (National Vulnerability Database) mantenuto dal NIST [9]

l'impiego del tool insieme a SonarQube è reso possibile da due componenti: l'analizzatore locale al progetto, il quale produce i documenti che vengono in seguito inclusi nel report prodotto dallo scan di sonar, e il plugin per SonarQube, installato sul server.

CheckStyle

l'impiego di checkstyle permette di ottenere un analisi più approfondita riguardo alle scelte stilistiche di codice (metrica relativa ai code smells) per quanto riguarda il codice java, ampiamente usato nei progetti presi in analisi nel corso di questa tesi [10].

2.4 Docker

La virtualizzazione nasce come tecnica per eseguire multiple macchine virtuali in singoli server fisici, garantendo isolamento reciproco tra di esse e permettendo un più efficiente utilizzo delle risorse fisiche della macchina. Il problema delle macchine virtuali è il loro grande overhead: ogni VM infatti è una macchina avente in esecuzione simulata ogni suo componente, dall'hardware virtualizzato al sistema operativo. Per ovviare a questo problema, è nata una diversa forma di virtualizzazione detta containerization. I Container sono simili alle macchine virtuali, ma godono di minore isolamento in quanto condividono con la macchina host il sistema operativo. Come le VM però, i container hanno il proprio filesystem, CPU, memoria, spazio dei processi, ecc... le due caratteristiche più interessanti dei container per quanto riguarda il loro utilizzo in ambito CI/CD sono:

- la loro portabilità: poiché sono disaccoppiati dall'infrastruttura della macchina host, possono girare su qualsiasi sistema operativo.

- il fatto che pesino così poco permette loro di essere integrati nelle pipeline come ambiente di run poiché possono essere inizializzati, eseguiti e fermati con un basso effort computazionale e in pochissimo tempo.

Docker è una piattaforma open-source che permette la creazione, la distribuzione e l'esecuzione di applicazioni in contenitori leggeri e isolati. La creazione dei container passa attraverso la scrittura di Dockerfile, file testuali dove vengono descritte le caratteristiche come sistema operativo, programmi, e comandi che devono essere lanciati in un certo container. I Dockerfile una volta interpretati ed eseguiti dal Docker daemon danno origine alle immagini docker, file considerabili dei template per la creazione dei container, condivisibili e archiviabili in servizi cloud, dai quali possono essere scaricati per avviare i container che descrivono.

2.5 SonarLint

SonarLint come SonarQube è un prodotto della SonarSource ed è un tool di analisi statica che rileva le stesse metriche di SonarQube. A differenza di SonarQube però, SonarLint è pensato come plugin per IDE, esiste infatti sia per IntelliJ che per Visual Studio Code. La potenza di analisi di SonarLint è limitata rispetto a quella di SonarQube, inoltre non può integrare i report di altri analizzatori come i risultati delle analisi di test coverage o OWASP Dependency Check. SonarLint però permette di effettuare un collegamento tra l'ambiente di sviluppo e SonarQube, visualizzando molte delle informazioni presenti nell'analisi sul server relativa ad un certo progetto mentre gli sviluppatori ci lavorano, direttamente nel loro ambiente di lavoro. Inoltre fornisce un analisi preliminare del codice mentre questo viene scritto, ponendosi come uno step di analisi a monte dell'infrastruttura di CI/CD, massimizzando il valore del suo contributo in termini di prevenzione del debito tecnico 3.2

Chapter 3

Situazione iniziale

Questo capitolo si propone di dare un overview della situazione di partenza sul contesto in cui si è svolta l'attività nel merito dello stato iniziale dei principali progetti e delle ragioni che hanno determinato alcune scelte implementative.

Contesto aziendale L'azienda target di questa tesi opera in un contesto in cui il processamento dei dati e la sicurezza sono fondamentali, pertanto vi è un forte impiego di tecnologie self-hosted e un forte limite per quanto riguarda lo scambio dei dati tra intranet ed internet, per cui anche librerie e immagini Docker vengono hostate in server proprietari tramite Nexus. L'organizzazione del lavoro prende alcuni elementi dalla metodologia agile: Lo sviluppo è diviso in sprint e vi sono numerosi team di poche persone che eseguono delle task di sviluppo comprese in quello sprint. L'organizzazione però non è agile: gli sprint hanno durata altamente variabile (dalle settimane ai numerosi mesi) e non sono chiusi rispetto all'introduzione di nuove task durante il loro svolgimento, inoltre le task vengono assegnate dai team leader e non è ad esse associata un esplicita stima di effort. Non sono inoltre comprese nelle task le attività di code review e refactoring che sono lasciate all'iniziativa degli individui. Spesso inoltre viene a mancare la fase di testing in (presunto) favore della velocità di produzione delle features.

Gesione progetti I progetti sono gestiti in parte tramite subversion e in parte tramite Gitlab, VCS introdotto in azienda da non molto tempo. Come strumento di CI/CD inoltre viene usato Jenkins, il server Jenkins presenta centinaia di pipeline aventi come source repository sia SVN che GitLab, l' esecuzione delle pipeline è

importante per i processi aziendali e dunque la soluzione ideale non crea disservizi e non mette a rischio la stabilità di Jenkins. Inoltre alcuni progetti stanno migrando all'uso di GitlabCI o sono direttamente passati ad utilizzare quel tool, ciò rende auspicabile trovare una soluzione che renda semplice la migrazione da un tool all'altro in caso di necessità. Infine sia su Jenkins che su GitlabCI sono presenti stage che fanno utilizzo di immagini Docker come ambiente di esecuzione, quindi la tecnologia è già familiare ai developers.

3.1 Introduzione di sonar

Per favorire l'introduzione di SonarQube nel workflow aziendale si è deciso di puntare prima alla realizzazione di una soluzione per un singolo progetto multimodulo che avesse una complessità sufficiente ad esplorare le varie funzionalità del tool. Lo scopo era da una parte farsi un idea preventiva dell'utilità del tool e dall'altra avere dei risultati da poter presentare ai developers in modo da dare il via ad un periodo di introduzione con un primo team.

3.1.1 Progetto di prova

Il progetto sul quale si è deciso di modellare la soluzione è un software internamente sviluppato dall'azienda avente le seguenti caratteristiche:

- circa 1 MLOC
- più di 30 moduli diversi
- linguaggi multipli, prevalentemente Java
- gestito con maven
- hostato su subversion
- ho lavorato sulla versione precedente rispetto all'active branch in modo da poter eseguire modifiche alla pipeline e al codice senza interferire con il lavoro dei developers

3.2 Prima soluzione proposta

La prima soluzione proposta era centrata attorno all'utilizzo del plugin di SonarQube per Jenkins [3.1].

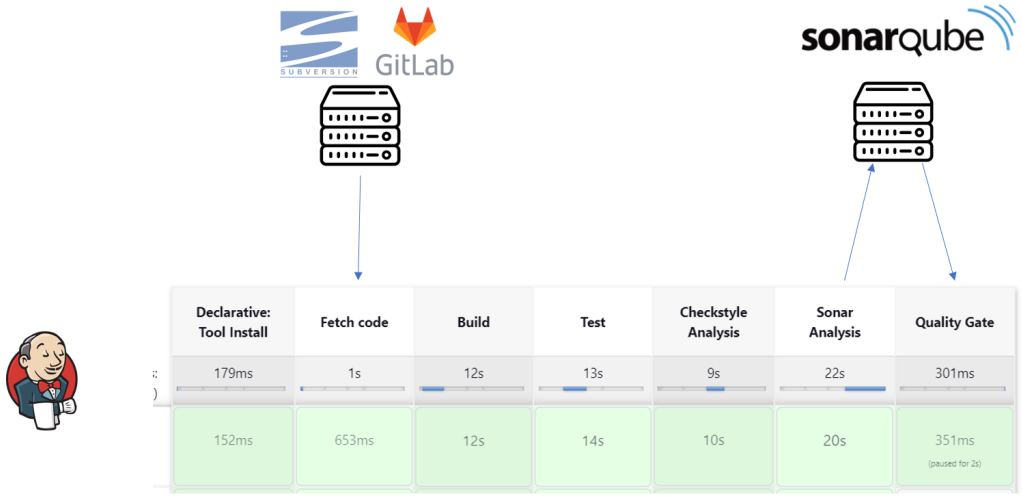

Figure 3.1: Pipeline con SonarQube per Jenkins

```
pipeline {
    agent none
    stages {
        stage('build') {
            agent {
                [...]
            }
            steps {
                [...]
            }
        }
        stage('Test'){
            steps {
                sh 'mvn test'
            }
        }
        stage('Checkstyle Analysis'){
```

```
18              steps {
19                  sh 'mvn checkstyle:checkstyle'
20              }
21          }
22          stage('Sonar Analysis') {
23              environment{
24                  scannerHome = [REDACTED]
25              }
26              steps {
27                 withSonarQubeEnv('sonar){
28                 sh '''$(scannerHome)/bin/sonar-scanner \
29                 -Dsonar.projectKey=[REDACTED] \
30                 -Dsonar.projectName=[REDACTED] \
31                 -Dsonar.projectVersion=[REDACTED] \
32                 -Dsonar.sources=[REDACTED] \
33                 -Dsonar.java.binaries=[REDACTED] \
34                 -Dsonar.junit.reportsPath=[REDACTED] \
35                 -Dsonar.jacoco.reportsPath=[REDACTED] \
36                 -Dsonar.java.checkstyle=[REDACTED]'''
37                 }
38              }
39          }
40          stage('Quality Gate') {
41              steps {
42                  timeout(time: 1, unit: 'HOURS'){
43                      waitForQualityGate abortPipeline:
    true
44                  }
45              }
46          }
47      }
48
49  }
```

Pro: Permette il tuning dei parametri di analisi, testandoli, senza agire su file del progetto passandoli tramite linea di comando

Contro:

- **Complessità progetti.** La soluzione così proposta è funzionale a progetti di dimensioni medio-piccole (ordine di grandezza massimo: decine di kLOC)

e con un processo di build semplice (singolo modulo, singolo linguaggio, limitata necessità di software specifici nell'ambiente di build). Alcuni progetti dell'azienda però hanno dimensioni da 100 kLOC a 1 MLOC, inoltre sono multi-modulari, scritti in più linguaggi e hanno la necessità di avere ambienti di build specifici.

- **Plugin Jenkins.** Nonostante il plugin Sonar per Jenkins sia mantenuto da SonarSource stessa, rendendolo dunque affidabile, l'accumulo di più plugin nel server Jenkins notoriamente può dare luogo a rallentamenti e instabilità del server. Essendo il servizio una risorsa chiave per l'azienda, è desiderabile evitare l'uso di plugin non strettamente necessari.
- **Portabilità.** La soluzione è locale a Jenkins, rende comunque necessario individuarne una diversa per GitlabCI

3.3 Prototipo Finale:

Per risolvere i problemi della prima soluzione proposta, ho deciso di emulare l'esecuzione dell'analisi in locale in un container Docker.

Pro:

- Poiché i progetti che lo necessitano già utilizzano pipeline eseguite in un container adatti alla loro build, si può creare un immagine a partire da essa aggiungendo esclusivamente ciò che serve all'esecuzione dell'analisi.
- Emulando in container un analisi locale, si elimina la necessità di usare il plugin SonarQube per Jenkins.
- Poiché sia Jenkins che GitlabCI mettono a disposizione la possibilità di eseguire stage in container, la soluzione è valida per entrambe le piattaforme.

Contro:

- Non è possibile distinguere il fail dell'analisi di sonar dovuto al fallimento del quality gate dal fallimento per altre ragioni (come errori di build, di compilazione o di parsing del codice) senza leggere il log dello stage di Jenkins (o GitlabCI) fallito.
- Per cambiare lo stage di analisi bisogna cambiare un immagine Docker

Essendo il progetto di testing gestito con maven, per effettuare la scansione in locale ho installato il plugin sonar-maven-plugin nel pom.xml che gestisce la build dell'intero progetto. Dopo aver individuato i parametri di analisi voluti eseguendo analisi in locale, li ho organizzati in due script:

- Il primo script serve ad utilizzare gli scanner di terze parti per il test reporting e per la coverage in un sottomodulo scritto in typescript, è stato necessario modificare il package.json del modulo per abilitare dei comandi
- Il secondo script si occupa di:
  - eseguire il dependency-check-maven plugin per ottenere il report sulle vulnerabilità
  - eseguire Checkstyle per ottenere il report sui code smells
  - eseguire jacoco per ottenere il report dei test java e la coverage
  - eseguire lo scan con sonar-maven-plugin e raccogliere i risultati degli step precedenti

Script1.sh

```
#!/usr/bin/bash -x

export NVM_DIR="$HOME/.nvm" \
&& [ -s "$NVM_DIR/nvm.sh" ] \
&& \. "$NVM_DIR/nvm.sh" \
&& nvm use system \
&& npm install nyc && npm install mocha-sonarqube-reporter \
&& npm install --force && npm run report-and-coverage
```

comandi aggiunti al package.json

```
{
    ...,
    "scripts": {
        "start:dev": "webpack-dev-server --mode development --
env.development --open --hot",
        "build": "npm run test && webpack --mode production --
env.production",
```

```
6             "test": " mocha --require ignore-styles -r ts-node/
    register -r jsdom-global/register --require ./test/helper.ts '
    test/**/*.ts' 'test/**/*.tsx'",
7             "test:report": "mocha --require ignore-styles -r ts-
    node/register -r jsdom-global/register --require ./test/helper.
    ts 'test/**/*.ts' 'test/**/*.tsx' --reporter mocha-sonarqube-
    reporter ./ --reporter-options output=test-report.xml --
    reporter-options useFullFilePath=true",
8             "coverage": "nyc npm run test",
9             "coverage:lcov": "nyc --reporter=lcov npm run test",
10            "report-and-coverage": "npm run test:report && npm run
     coverage:lcov"
11        },
12  ...
13    }
```

Script2.sh

```
1  #!/usr/bin/env bash
2
3  set -e
4
5  if [ -z "${SONAR_PROJECT_KEY}" ]; then
6    echo "Missing env variable SONAR_PROJECT_KEY. Please create a
      Sonar project and put its project key in the SONAR_PROJECT_KEY
      env variable."
7    exit 1
8  fi
9
10 if [ -z "${SONAR_PROJECT_TOKEN}" ]; then
11   echo "Missing env variable SONAR_PROJECT_TOKEN. Please generate
      a token for the sonar project and assign it to the
      SONAR_PROJECT_TOKEN env variable."
12   exit 1
13 fi
14
15 if [ -z "${AUTO_UPDATE}" ]; then
16   echo "Missing env variable AUTO_UPDATE. Please create it and set
       it either to true or false. documentation at: https://
      jeremylong.github.io/DependencyCheck/dependency-check-maven/
      check-mojo.html#autoUpdate"
17   exit 1
18 fi
19
20 if [ !"${AUTO_UPDATE}" = true -a !"${AUTO_UPDATE}" = false ]; then
21   echo "Set env variable AUTO_UPDATE either to true or false."
```

```
22    echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       autoUpdate"
23    exit 1
24  fi
25
26  if [ -z "${CVE_URL}" ]; then
27    echo "Missing env variable CVE_URL. Please provide a Base Data
       Mirror URL for CVE 1.2."
28    echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       cveUrlBase"
29    exit 1
30  fi
31
32  if [ -z "${RETIREJS_URL}" ]; then
33    echo "Missing env variable RETIREJS_URL. Please provide The
       Retire JS repository URL."
34    echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       retireJsUrl"
35    exit 1
36  fi
37
38  if [ -z "${KEV_URL}" ]; then
39    echo "Missing env variable KEV_URL. Please provide the mirror
       URL to the CISA Known Exploited Vulnerabilities JSON datafeed..
       "
40    echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       knownExploitedUrl"
41    exit 1
42  fi
43
44  set +e
45
46  mvn -Dformats=HTML,JSON -DossindexAnalyzerEnabled=false \
47   -DcveUrlBase="${CVE_URL}" \
48    -DretireJsUrl="${RETIREJS_URL}" \
49    -DknownExploitedUrl="${KEV_URL}" \
50    -DautoUpdate="${AUTO_UPDATE}" \
51  clean jacoco:prepare-agent \
52  package jacoco:report org.owasp:dependency-check-maven:7.3.2:
       aggregate --batch-mode;
53
54  set -e
55
56  mvn checkstyle:checkstyle; \
```

```
export NVM_DIR="$HOME/.nvm" && [ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" && nvm use 16.19.1;
JAVA_HOME=/usr/lib/jvm/java-11-openjdk/ mvn \
 -Dsonar.projectKey="${SONAR_PROJECT_KEY}" \
 -Dsonar.host.url=[SONAR_SERVER_URL] \
 -Dsonar.login="${SONAR_PROJECT_TOKEN}" \
 -Dsonar.dependencyCheck.summarize=true \
 -Dsonar.dependencyCheck.jsonReportPath=mito/target/dependency-check-report.json \
 -Dsonar.dependencyCheck.htmlReportPath=mito/target/dependency-check-report.html \
 -Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml \
 -Dsonar.scm.disabled=true \
 -Dsonar.testExecutionReportPaths=mitojs/mitojs/reactComponents/test-report.xml \
 -Dsonar.javascript.lcov.reportPaths=mitojs/mitojs/reactComponents/coverage/lcov.info \
 org.sonarsource.scanner.maven:sonar-maven-plugin:3.9.1.2184:sonar --batch-mode
```

Ho generato dunque a partire dall'immagine utilizzata per la build, l'immagine usata per l'analisi, alla quale ho aggiunto open-jdk-11 e node 16, rispettivamente la versione della JDK minima e la versione di node consigliata per il supporto del sonar-scanner:

```
FROM nome-immagine-di-build:versione-immagine

#to execute the sonar scan process java 11 or higher is required ( https://docs.sonarqube.org/8.9/analyzing-source-code/moving-analysis-to-java-11/ )
RUN sudo yum install java-11-openjdk-devel -y
COPY settings.xml /home/build/.m2/
RUN chown build:build /home/build/.m2/settings.xml

#a separate script to first build and generate reports for the javascript/typescript module (mitojs)
ADD scripts/generate-mitojs-reports.sh "/opt/generate-mitojs-reports.sh"
RUN chown build:build /opt/generate-mitojs-reports.sh
RUN chmod +x /opt/generate-mitojs-reports.sh

#then build and generate reports for the whole maven project (mitoparent)
ADD scripts/sonar-scan-mitocuberpm.sh "/opt/sonar-scan-mitocuberpm.sh"
```

```
15      RUN chown build:build /opt/sonar-scan-mitocuberpm.sh
16      RUN chmod +x /opt/sonar-scan-mitocuberpm.sh
17
18      #switching to user build to install the different node
    versions required
19      USER build
20      #to execute the scan process of the javascript/typescript
    module (mitojs) node version 14 or higher is required ( https
    ://docs.sonarqube.org/9.7/analyzing-source-code/languages/
    javascript-typescript-css/ )
21      RUN curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0
    .39.3/install.sh | bash -x\
22        && export NVM_DIR="$HOME/.nvm" \
23        && [ -s "$NVM_DIR/nvm.sh" ] \
24        && \. "$NVM_DIR/nvm.sh" \
25        && nvm install 16.19.1 \
26        && nvm install 12.22.1 \
27        && nvm use system
28
29
30      ENV NVM_DIR="$HOME/.nvm"
31      ENV CVE_URL=https://[path/to/nexus]nist_cve/nvdcve-1.1-%d.json
    .gz
32      ENV     RETIREJS_URL=https://[path/to/nexus]retire_js/
    jsrepository.json
33      ENV KEV_URL=https://[path/to/nexus]cisa-known-vuln-expl/
    known_exploited_vulnerabilities.json
34      ENV AUTO_UPDATE=true
```

ed infine ho ristrutturato la pipeline secondo le best practice indicate sul sito di Jenkins per quanto riguarda l'uso di pipeline multi-container [11] ottenendo:

```
1  pipeline {
2      agent none
3      stages {
4          stage('build') {
5              agent {
6                  docker {
7                      label 'docker'
8                      image "[nome-immagine-di-build:versione
   -immagine]"
9                      registryUrl "[registry-aziendale]"
10                     registryCredentialsId '[credenziali-
   registry]'
```

```
11                  args "--user build [...]"
12               }
13           }
14           steps {
15               [...]
16           }
17       }
18       [...]
19       stage('Sonar scan') {
20           agent {
21               docker {
22                 label 'docker'
23                   image '[immagine-di-scan:versione-
    immagine]'
24                 registryUrl "[registry-aziendale]"
25                 registryCredentialsId '[credenziali-
    registry]'
26                 args "--user build [...]"
27                }
28           }
29           environment{
30               SVNCRED = [...]
31               NEXUSCRED = [...]
32               SONAR_PROJECT_KEY=[...]
33               SONAR_PROJECT_TOKEN=[...]
34           }
35           steps {
36               dir('path/to/js-module'){
37                   sh "cat /opt/script1.sh"
38                   sh "/opt/script1.sh"
39               }
40               dir('path/to/composer-module') {
41                   sh "cat /opt/script2.sh"
42                   sh "/opt/script2.sh"
43               }
44           }
45       }
46    }
47
48 }
```

3.4 Introduzione del tool ai developers

Allo scopo di mettere i developers nelle condizioni di utilizzare SonarQube, bisogna creare nel team un vocabolario comune di concetti che favoriscano un mindset positivo verso di esso. Per questa ragione ho fatto una lezione ai developers allo scopo di mettere tutti sullo stesso livello di conoscenza sia di alcuni concetti di software engineering come il technical debt, le sue cause principali e l'importanza che ha l'anticipare il più possibile l'intercettazione dei defects in relazione al ciclo di vita del codice che viene scritto [3.2] sia nel merito delle funzionalità di SonarQube e SonarLint che avrebbero utilizzato, mostrando come esempio l'analisi ottenuta dal primo progetto. Un aspetto importante dell'introduzione del tool è essere precisi nella definizione dei significati delle metriche esposte e nello specificare la naturale fallibilità dello strumento, questo serve ad evitare che i developer vedano l'uso del tool come una critica al loro lavoro e perdano fiducia in esso appena si dovesse verificare la situazione in cui il tool ha segnalato un defect che non doveva essere considerato tale.

Table 5-2. Preliminary Estimates of Relative Cost Factors of Correcting Errors as a Function of Where Errors Are Introduced and Found (Example Only)

	Where Errors are Found				
Where Errors are Introduced	**Requirements Gathering and Analysis/ Architectural Design**	**Coding/ Unit Test**	**Integration and Component/ RAISE System Test**	**Early Customer Feedback/Beta Test Programs**	**Post-product Release**
Requirements Gathering and Analysis/ Architectural Design	1.0	5.0	10.0	15.0	30.0
Coding/Unit Test		1.0	10.0	20.0	30.0
Integration and Component/ RAISE System Test			1.0	10.0	20.0

Figure 3.2: crescita relativa del TD dovuto ad un defect in relazione alla fase di vita in cui avviene la sua scoperta[12]

Alla fine della presentazione, ho presentato un questionario ai 9 developers allo scopo di:

- sondare il loro sentiment riguardo il tool e l'iniziativa di introdurlo nel loro workflow

- misurare la loro esperienza/percezione sulla località e gravità dei problemi in modo da:
  - considerarla nella scelta del prossimo progetto da analizzare
  - poterla confrontare con la knowledge emersa dalle analisi dei progetti a posteriori

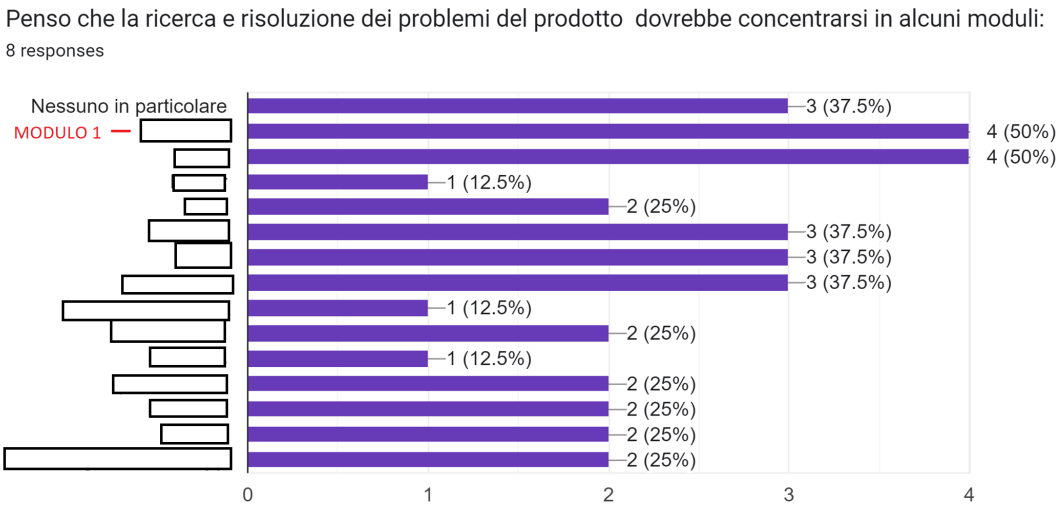

Figure 3.3: Risultati domanda 1

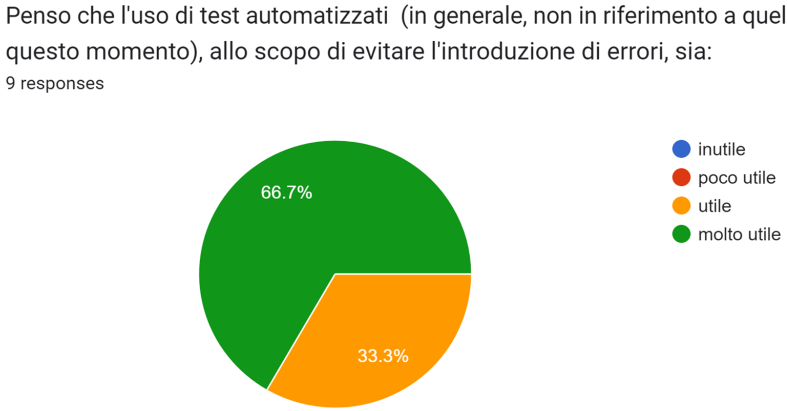

Figure 3.4: Risultati domanda 2

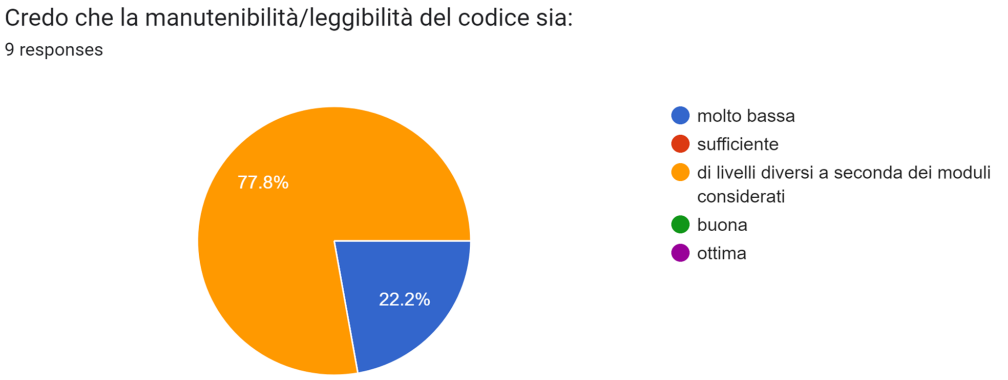

Figure 3.5: Risultati domanda 3

Figure 3.6: Risultati domanda 4

Conclusioni del questionario:

- per quanto riguarda la prima domanda [3.3] non ci sono stati picchi nell'individuazione di uno specifico progetto tra quelli proposti quindi è stato scelto [MODULO 1] per la sperimentazione con il team essendo uno dei due più indicati e in fase di sviluppo.
- nonostante il limitato (relativamente alla grandezza della codebase) quantitativo di test automatizzati presente, tutti i developers hanno espresso fiducia nel loro impiego [3.4]
- Dalla terza domanda [3.5] emerge l'insoddisfazione dei developers nel merito

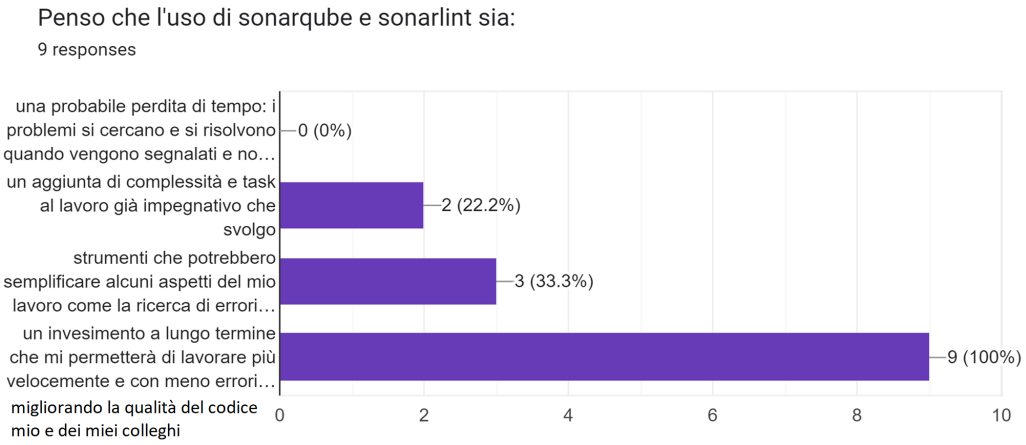

Figure 3.7: Risultati domanda 5

della leggibilità del codice. Questo dato è stato determinante nella scelta di mantenere anche per i futuri progetti sonar l'impiego di Checkstyle in modo da forzare uno stile standard nel codice prodotto.

- I risultati della domanda 4 [3.6] fanno venire a galla il problema principale del processo di produzione. Il tempo degli sprint viene quasi totalmente allocato allo sviluppo e al bugfixing dei bug riportati, non esiste un processo di code review strutturato e delle task apposite per esso.
- I risultati della domanda 5 [3.7] infine mostrano un preliminare senso di fiducia nei confronti dell'impiego del tool.

Chapter 4

Integrazione in progetti attivi

4.1 Approccio scelto

Considerato il fatto che tutti i progetti presi in considerazione erano avviati da tempo e avevano alle spalle dalle decine di migliaia alle centinaia di migliaia di linee di codice, il technical debt accumulato ed evidenziato nella maggior parte dei casi non è affrontabile in tempi di breve-medio termine. Per questa ragione si è scelto di utilizzare l'approccio "clean as you code" che consiste nel concentrarsi sulla qualità del codice prodotto come "New Code". In questo modo, non solo si evita di introdurre nuovo technical debt sotto forma di bug e vulnerabilità, ma si migliora progressivamente anche il resto della codebase poiché con il crescere della complessità dei progetti aumenta la necessità di modificare il codice già prodotto in passato quando si effettuano nuove aggiunte [13] e di conseguenza questo codice rientra a fare parte del "new code" che deve passare il quality gate.

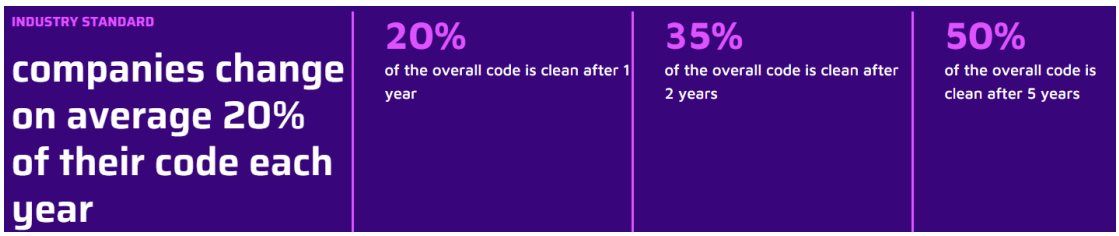

Figure 4.1: clean as you code: impatto globale nel tempo[13]

Inoltre per rendere incrementale l'adozione di questo approccio si è scelto di integrare lo step di sonar alla fine delle pipeline in modo da non renderlo bloccante ma di dare comunque un feedback sulla qualità del codice, per poi passarlo in un secondo momento alla prima posizione della pipeline in modo da rendere mandatorio il passaggio del Quality Gate. Lo stesso principio è stato adottato per le metriche del QG: partendo da delle concessioni rispetto al Sonar-way (soglie più permissive) si è deciso di incrementare la qualità richiesta di sprint in sprint fino a incontrare lo standard

4.2 Primo progetto: "Modulo 1"

Il progetto che chiameremo "Modulo 1" come indicato nell'immagine [3.3] e i suoi developers sono stati i primi scelti come banco di prova per l'uso di SonarQube sulla base delle risposte al questionario.

Il progetto era così strutturato a grandi linee:

- hostato su GitLab
- pipeline diverse per diversi branch, su Jenkins
- diviso in tre parti [4.2]:
  - il codice effettivamente parte di questo progetto, mantenuto ed espanso ad ogni sprint dai developers, prevalentemente scritto in TypeScript
  - due librerie di cui vengono usate limitate funzionalità, che non subiscono quasi mai modifiche secondo l'esperienza dei developers e che sono in via di abbandono, erano vecchi progetti dell'azienda.
  - dimensione nell'ordine delle decine di migliaia di righe di codice.

La prima versione realizzata della pipeline dunque, che realizzava l'analisi dell'intera codebase del repository, si discostava dall'obiettivo di mantenere il focus sul codice nuovo andando a mischiare le metriche dell'overall code con i problemi rilevati nelle vecchie librerie in via di sostituzione. Per questo motivo ho riprogettato la pipeline in modo che esclusivamente sul main branch provvedesse ad effettuare l'analisi separata per le tre parti del repository corrispondenti a 3 progetti su sonar. Nei branch di sviluppo invece viene effettuata l'analisi escludendo le due librerie. Inoltre una problematica da affrontare è l'assenza della feature di branch analysis

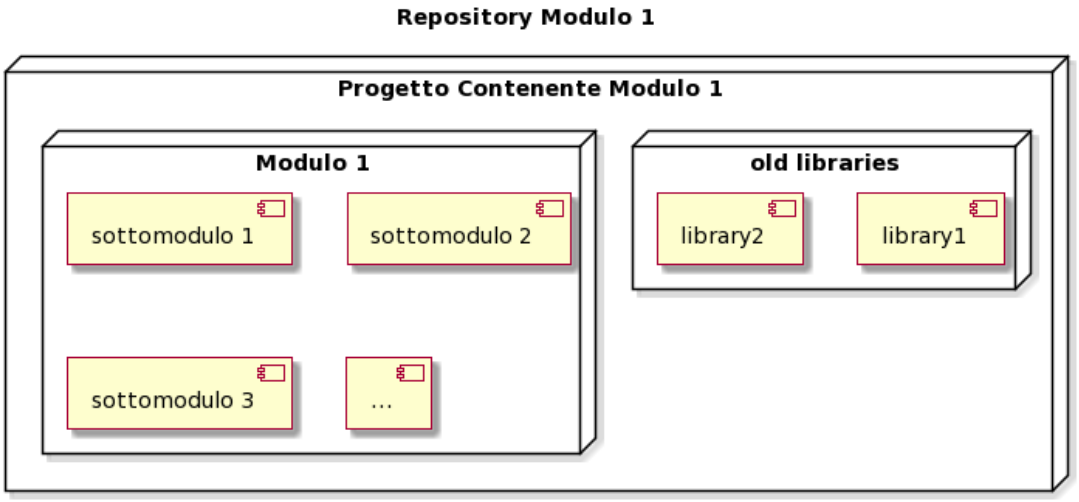

Figure 4.2: struttura repository modulo 1

poiché è una feature inclusa a partire dalla prima licenza a pagamento e questi test sono stati effettuati con la Community Edition (licenza gratuita).

Tool utilizzati:

- **Reporters:**
  - jest per il coverage report, invocato con 'react-scripts test –coverage . –watchAll=false –passWithNoTests'
  - dependency-check-cli [14] to produce the vulnerabilities report
- **Scanner:** sonar-scanner-npm [15] che sostanzialmente imbusta in un pacchetto npm la sonar-scanner cli

4.2.1 Test in locale

Ho testato l'analisi eseguendola nel container Docker dell'immagine di build in locale per

- definire quali tool sarebbero stati necessari nell'immagine di analisi
- rifinire i parametri di analisi

Sono arrivato alla conclusione che necessitavo dei CLI tools di dependency-check e del sonar-scanner, inoltre ho raggruppato in 3 script i comandi e i parametri di analisi per le tre sezioni del progetto:

Script di analisi per "Modulo 1":

```
#!/usr/bin/bash -x
#npm install -D jest-sonar-reporter sonarqube-scanner \
npm install --loglevel verbose --no-audit;
npm run test -- --coverage . --watchAll=false;
dependency-check.sh --project \"activity-list\" --scan \"package-
    lock.json\" -f JSON -f HTML \
 --cveUrlModified https://[path/to/nexus]nist_cve/nvdcve-1.1-2002.
    json.gz \
 --cveUrlBase "${CVE_URL}" --retireJsUrl "${RETIREJS_URL}" --
    kevURL "${KEV_URL}" \
 --disableOssIndex --disableNodeAudit;
sonar-scanner \
  -Dsonar.host.url=[SONAR_SERVER_URL] \
  -Dsonar.login="${SONAR_MODULO_1_TOKEN}" \
  -Dsonar.projectKey="${SONAR_MODULO_1_KEY}"\
  -Dsonar.projectName="${SONAR_MODULO_1_NAME}" \
  -Dsonar.sources=src/ \
  -Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info \
  -Dsonar.sourceEncoding=UTF-8 \
  -Dsonar.tests=./src/test \
  -Dsonar.test.inclusions=**/*.test.ts,**/*.test.tsx \
  -Dsonar.exclusions=./src/test/**/__test__/** \
  -Dsonar.dependencyCheck.htmlReportPath=./dependency-check-report
    .html \
  -Dsonar.dependencyCheck.jsonReportPath=./dependency-check-report
    .json \
  -Dsonar.qualitygate.wait=false
```

Script di analisi per "Libreria 1":

```
#!/usr/bin/bash -x

sonar-scanner \
  -Dsonar.host.url=[SONAR_SERVER_URL] \
  -Dsonar.login="${SONAR_Libreria_1_TOKEN}" \
  -Dsonar.projectKey="${SONAR_Libreria_1_KEY}"\
  -Dsonar.projectName="Libreria_1-filters" \
  -Dsonar.sources=. \
```

```
  -Dsonar.sourceEncoding=UTF-8 \
```

Script di analisi per "Libreria 2":

```
#!/usr/bin/bash -x

sonar-scanner \
  -Dsonar.host.url=[SONAR_SERVER_URL] \
  -Dsonar.login="${SONAR_Libreria_2_TOKEN}" \
  -Dsonar.projectKey="${SONAR_Libreria_2_KEY}"\
  -Dsonar.projectName="Libreria_2" \
  -Dsonar.sources=. \
  -Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info \
  -Dsonar.sourceEncoding=UTF-8 \
  -Dsonar.tests=./test \
  -Dsonar.test.inclusions=./test/**/*,**/*.test.ts,**/*.test.tsx \
  -Dsonar.exclusions=./Docker/**/*,./doc/**/* \
```

4.2.2 Immagine docker

Una volta definiti tool e parametri necessari dunque ho proceduto a creare un immagine docker a partire da quella usata per la build del progetto:

```
    FROM nome-immagine-di-build:versione-immagine

    # adding the scripts to scan the projects
    RUN mkdir /opt/scripts
    ADD scripts/sonar-scan-modulo_1.sh "/opt/scripts/sonar-scan-
    modulo_1.sh"
    ADD scripts/sonar-scan-modulo_1-Libreria_1.sh "/opt/scripts/
    sonar-scan-modulo_1-Libreria_1.sh"
    ADD scripts/sonar-scan-modulo_1-Libreria_2.sh "/opt/scripts/
    sonar-scan-modulo_1-Libreria_2.sh"
    RUN chown build -R /opt/scripts

    #install dependency check cli 8.2.1 (https://jeremylong.github
    .io/DependencyCheck/dependency-check-cli/index.html)
    RUN mkdir /opt/tools
    WORKDIR /opt/tools
    COPY tools/dependency-check-8.2.1-release.zip .
    RUN unzip /opt/tools/dependency-check-8.2.1-release.zip
```

```
15     RUN ln -s /opt/tools/dependency-check/bin/dependency-check.sh
       /usr/bin/dependency-check.sh
16     RUN chown build -R /opt/tools
17     WORKDIR /home/build
18
19     # npm module that basically wraps the sonar-scanner cli,
       documentation here: https://github.com/SonarSource/sonar-
       scanner-npm
20     RUN npm install -g sonarqube-scanner
21     #this sets up the sonar scanner cli for the build user
22     USER build
23     RUN sonar-scanner -h
24
25     ENV CVE_URL=https://[path/to/nexus]nist_cve/nvdcve-1.1-%d.json
       .gz
26     ENV RETIREJS_URL=https://[path/to/nexus]retire_js/jsrepository
       .json
27     ENV KEV_URL=https://[path/to/nexus]cisa-known-vuln-expl/
       known_exploited_vulnerabilities.json
28     ENV AUTO_UPDATE=true
```

4.2.3 Pipelines

Infine ho ristrutturato la pipeline secondo le best practice indicate sul sito di jenkins per quanto riguarda l'uso di pipeline multi-container [11] ottenendo:

Per i branch di development, ognuno con il suo progetto su sonar con project-key e project name univoca:

```
1  pipeline {
2      agent none
3      stages {
4          stage('build') {
5              agent {
6                  docker {
7                      label 'docker'
8                      image "[nome-immagine-di-build:versione
       -immagine]"
9                      registryUrl "[registry-aziendale]"
10                     registryCredentialsId '[credenziali-
       registry]'
```

```
                    args "--user build [...]"
                }
            }
            steps {
                [...]
            }
        }
        [...]
        stage('Sonar scan') {
            agent {
                docker {
                  label 'docker'
                    image '[immagine-di-scan:versione-
immagine]'
                  registryUrl "[registry-aziendale]"
                  registryCredentialsId '[credenziali-
registry]'
                  args "--user build [...]"
                 }
            }
            environment{
                SVNCRED = [...]
                NEXUSCRED = [...]
                SONAR_PROJECT_KEY=[...]
                SONAR_PROJECT_TOKEN=[...]
            }
            steps {
                dir('path/to/js-module'){
                    sh "cat /opt/script1.sh"
                    sh "/opt/script1.sh"
                }
                dir('path/to/composer-module') {
                    sh "cat /opt/script2.sh"
                    sh "/opt/script2.sh"
                }
            }
        }
        stage('Sonar scan') {
            agent {
                docker {
                    label 'docker'
```

```
                    image '[immagine-di-scan:versione-immagine]'
                  registryUrl "[registry-aziendale]"
                  registryCredentialsId '[credenziali-registry]'
                  args "--user build [...]"
              }
          }
          environment{
              SVNCRED = [...]
              NEXUSCRED = [...]
              SONAR_MODULO_1_NAME=[...]
              SONAR_MODULO_1_KEY=[...]
              SONAR_MODULO_1_TOKEN=[...]
          }
          steps {
              sh 'whoami'
              sh 'env'
              sh "cat /opt/scripts/sonar-scan-[MODULO 1].sh"
              sh "/opt/scripts/sonar-scan-[MODULO 1].sh"
          }
      }
   }

}
```

Per il branch master, con l'analisi separata per i tre progetti:

```
pipeline {
    agent none
    stages {
        stage('build') {
            agent {
                docker {
                 label 'docker'
                 image "[nome-immagine-di-build:versione-immagine]"
```

```
                registryUrl "[registry-aziendale]"
                registryCredentialsId '[credenziali-registry]'
                args "--user build [...]"
            }
        }
        steps {
            [...]
        }
    }
    [...]
    stage('Sonar scan') {
        agent {
            docker {
              label 'docker'
                image '[immagine-di-scan:versione-immagine]'
              registryUrl "[registry-aziendale]"
              registryCredentialsId '[credenziali-registry]'
              args "--user build [...]"
             }
        }
        environment{
            SVNCRED = [...]
            NEXUSCRED = [...]
            SONAR_PROJECT_KEY=[...]
            SONAR_PROJECT_TOKEN=[...]
        }
        steps {
            dir('path/to/js-module'){
                sh "cat /opt/script1.sh"
                sh "/opt/script1.sh"
            }
            dir('path/to/composer-module') {
                sh "cat /opt/script2.sh"
                sh "/opt/script2.sh"
            }
        }
    }
    stage('Sonar scan') {
```

```
47              agent {
48                  docker {
49                      label 'docker'
50                      image '[immagine-di-scan:versione-
    immagine]'
51                    registryUrl "[registry-aziendale]"
52                    registryCredentialsId '[credenziali-
    registry]'
53                    args "--user build [...]"
54                  }
55              }
56              environment{
57                  SVNCRED = [...]
58                  NEXUSCRED = [...]
59                  SONAR_MODULO_1_NAME=[...]
60                  SONAR_MODULO_1_KEY=[...]
61                  SONAR_MODULO_1_TOKEN=[...]
62                  SONAR_LIBRERIA_2_KEY=[...]
63                  SONAR_LIBRERIA_2_TOKEN=[...]
64                  SONAR_LIBRERIA_1_KEY=[...]
65                  SONAR_LIBRERIA_1_TOKEN=[...]
66              }[...]
67              steps {
68                  sh 'whoami'
69                  sh 'env'
70                  sh "cat /opt/scripts/sonar-scan-[MODULO
    1].sh"
71                  sh "/opt/scripts/sonar-scan-[MODULO 1].
    sh"
72                  dir('public/libreria_1/') {
73                      sh 'cat /opt/scripts/sonar-scan-
    modulo_1-libreria_1.sh'
74                      sh '/opt/scripts/sonar-scan-modulo_1
    -libreria_1.sh'
75                  }
76                  dir('public/LIBRERIA_2/') {
77                      sh 'cat /opt/scripts/sonar-scan-
    modulo_1-libreria_2.sh'
78                      sh '/opt/scripts/sonar-scan-modulo_1
    -libreria_2.sh'
79                  }
```

```
80                  }
81              }
82          }
83
84  }
```

4.2.4 Quality Gate

Il Quality Gate dunque è stato impostato su condizioni esclusivamente legate alle metriche del New Code ed è stato «rilassato» solo per quanto riguarda la metrica relativa alla soglia di coverage, portandola a 0, per dare tempo al team di impostare l'attività di testing e prendere confidenza con Jest.

4.3 Secondo progetto: "Modulo 2"

Il secondo progetto in cui è stato integrato sonar aveva caratteristiche simili al Modulo 1 per quanto riguarda le tecnologie utilizzate ma presentava un caso più semplice in quanto non comprendeva sotto-moduli da trattare a parte, chiameremo questo progetto "Modulo 2" Inoltre, questo progetto a differenza del precedente non presentava numerose pipelines per rispondere ad esigenze quali la produzione di versioni diverse o di organizzazione nello sviluppo di features.

"Modulo 2" era così strutturato a grandi linee:

- hostato su GitLab
- singola pipeline per il master branch, su Jenkins
- prevalentemente scritto in TypeScript
- un singolo modulo
- dimensione nell'ordine delle migliaia di righe di codice.

Tool utilizzati:

- **Reporters:**

  - jest per il report della coverage ‘react-scripts test –coverage . –watchAll=false –passWithNoTests‘
  - dependency-check-cli [14] per produrre il report di analisi vulnerabilità

- **Scanner:** sonar-scanner-npm [15] che sostanzialmente imbusta in un pacchetto npm la sonar-scanner cli

4.3.1 Test in locale

Ho testato l’analisi eseguendola nel container docker dell’immagine di build in locale per

- definire quali tool sarebbero stati necessari nell’immagine di analisi
- rifinire i parametri di analisi

Sono arrivato alla conclusione che necessitavo dei CLI tools di dependency-check e del sonar-scanner, inoltre ho raggruppato in 1 script i comandi e i parametri di analisi per il progetto:

Script di analisi per "Modulo 2":

```
#!/usr/bin/env bash

npm install --loglevel verbose --no-audit;
npm run test -- --coverage . --watchAll=false --passWithNoTests;
dependency-check.sh --project \"[Modulo 2]\" --scan \"package-lock.json\" -f JSON -f HTML \
 --cveUrlModified "[CVE_URL_MODIFIED]" \
 --cveUrlBase "${CVE_URL}" --retireJsUrl "${RETIREJS_URL}" --kevURL "${KEV_URL}" \
 --disableOssIndex --disableNodeAudit;
sonar-scanner \
  -Dsonar.host.url=[SONAR_SERVER_URL] \
  -Dsonar.login="${SONAR_TOKEN}" \
  -Dsonar.projectKey="${SONAR_KEY}"\
  -Dsonar.projectName="${SONAR_NAME}" \
  -Dsonar.sources=src/ \
  -Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info \
  -Dsonar.sourceEncoding=UTF-8 \
  -Dsonar.test.inclusions=**/*.test.ts,**/*.test.tsx \
```

```
18   -Dsonar.exclusions=./src/test/**/__test__/** \
19   -Dsonar.dependencyCheck.htmlReportPath=./dependency-check-report
      .html \
20   -Dsonar.dependencyCheck.jsonReportPath=./dependency-check-report
      .json \
21   -Dsonar.qualitygate.wait=false
```

4.3.2 Immagine docker

Una volta definiti tool e parametri necessari dunque ho proceduto a creare un immagine docker a partire da quella usata per la build del progetto:

```
1    FROM nome-immagine-di-build:versione-immagine
2
3    ADD scripts/sonar-scan-[modulo2].sh "/opt/sonar-scan-[modulo2
    ].sh"
4    RUN chown build /opt/sonar-scan-[modulo2].sh
5
6    #install dependency check cli 8.2.1 (https://jeremylong.github
    .io/DependencyCheck/dependency-check-cli/index.html)
7    RUN mkdir /opt/tools
8    WORKDIR /opt/tools
9    COPY tools/dependency-check-8.2.1-release.zip .
10   RUN unzip /opt/tools/dependency-check-8.2.1-release.zip
11   RUN ln -s /opt/tools/dependency-check/bin/dependency-check.sh
    /usr/bin/dependency-check.sh
12   RUN chown build -R /opt/tools
13   WORKDIR /home/build
14
15   # npm module that basically wraps the sonar-scanner cli,
    documentation here:    https://github.com/SonarSource/sonar-
    scanner-npm
16   RUN npm install -g sonarqube-scanner
17   #this sets up the sonar scanner cli for the build user
18   USER build
19   RUN sonar-scanner -h
20
21   ENV CVE_URL=https://[path/to/nexus]/repository/nist_cve/nvdcve
    -1.1-%d.json.gz
22   ENV RETIREJS_URL=https://[path/to/nexus]/repository/retire_js/
    jsrepository.json
23   ENV KEV_URL=https://[path/to/nexus]/repository/cisa-known-vuln
    -expl/known_exploited_vulnerabilities.json
24   ENV AUTO_UPDATE=true
```

4.3.3 Pipeline

Infine ho ristrutturato la pipeline secondo le best practice indicate sul sito di Jenkins per quanto riguarda l'uso di pipeline multi-container [11] ottenendo:

```
pipeline {
    agent none
    stages {
        stage('build') {
            agent {
                docker {
                 label 'docker'
                 image "[nome-immagine-di-build:versione-immagine]"
                 registryUrl "[registry-aziendale]"
                 registryCredentialsId '[credenziali-registry]'
                 args "--user build [...]"
              }
            }
            steps {
                [...]
            }
        }
        [...]
        stage('Sonar scan') {
            agent {
                docker {
                    label 'docker'
                    image '[immagine-di-scan]:[versione-immagine]'
                    registryUrl " [registry-aziendale]"
                    registryCredentialsId '[credenziali-registry]'
                    args "--user build [...]"
                }
```

```
28                }
29                environment{
30                    SVNCRED = [...]
31                    NEXUSCRED = [...]
32                    SONAR_KEY=[...]
33                    SONAR_TOKEN=[...]
34                    SONAR_NAME=[...]
35                }
36                steps {
37                    sh 'cat /opt/sonar-scan-[module2].sh'
38                    sh '/opt/sonar-scan-[module2].sh'
39                }
40            }
41        }
42
43    }
```

4.3.4 Quality Gate

Il Quality Gate è stato uniformato a quello del progetto "Modulo 1": è stato impostato su condizioni esclusivamente legate alle metriche del New Code ed è stato «rilassato» solo per quanto riguarda la metrica relativa alla soglia di coverage, portandola a 0, per dare tempo al team di impostare l'attività di testing e prendere confidenza con Jest.

4.4 Terzo progetto: "Libreria multi-modulo"

Il terzo progetto in cui è stato integrato sonar è una libreria multi-modulo esistente in più versioni, ognuna con il suo branch nominato [Libreria multi-modulo]_Versione (dove versione è un numero intero progressivo) e la sua pipeline corrispondente. Essendo questa Libreria un progetto di grosse dimensioni utile alla produzione di uno dei prodotti principali dell'azienda, ad ogni nuovo active branch/versione del prodotto corrisponde una nuova versione di questa libreria, di conseguenza con il susseguirsi degli sprint vengono progressivamente generati nuovi branch e abbandonate le versioni vecchie.

"Libreria multi-modulo" è così strutturato a grandi linee:

- hostato su Subversion
- Una pipeline per ogni branch subversion
- prevalentemente scritto in Java
- multi-modulo
- dimensione nell'ordine delle centinaia di migliaia di righe di codice.

Tool utilizzati:

- **Reporters:**
  - dependency-check-maven[16] **versione 7.3.2** per il report di vulnerabilità, è importante notare che versioni superiori alla 8.x non sono supportate da SonarQube 9.7 dal momento che producono report con un formato incompatibile
  - maven-checkstyle-plugin [17] per produrre il report di analisi vulnerabilità
  - 'jacoco-maven-plugin [18] invocato con jacoco:report' per report di coverage e test
- **Scanner:** sonar-maven-plugin [19], il tool indicato dalla documentazione di sonar come da usare in caso di progetti multi-modulo gestiti con maven

4.4.1 Test in locale

In seguito ad aver effettuato i test di analisi eseguendola nel container docker dell'immagine di build in locale, ho raggruppato in uno script i comandi e i parametri di analisi per il progetto, in cui prima vengono generati il report di vulnerabilità, il report sui test results e il report sulla coverage, poi viene generato il report di Checkstyle e infine viene eseguito lo scan con sonar in cui vengono collezionati i report generati durante step precedenti:

Script di analisi per "Libreria multi-modulo":

```
1 #!/usr/bin/env bash
2
3 set -e
```

```
4
5  if [ -z "${SONAR_PROJECT_KEY}" ]; then
6    echo "Missing env variable SONAR_PROJECT_KEY. Please create a
       Sonar project and put its project key in the SONAR_PROJECT_KEY
       env variable."
7    exit 1
8  fi
9
10 if [ -z "${SONAR_PROJECT_TOKEN}" ]; then
11   echo "Missing env variable SONAR_PROJECT_TOKEN. Please generate
       a token for the sonar project and assign it to the
       SONAR_PROJECT_TOKEN env variable."
12   exit 1
13 fi
14
15 if [ -z "${AUTO_UPDATE}" ]; then
16   echo "Missing env variable AUTO_UPDATE. Please create it and set
        it either to true or false. documentation at: https://
       jeremylong.github.io/DependencyCheck/dependency-check-maven/
       check-mojo.html#autoUpdate"
17   exit 1
18 fi
19
20 if [ !"${AUTO_UPDATE}" = true -a !"${AUTO_UPDATE}" = false ]; then
21   echo "Set env variable AUTO_UPDATE either to true or false."
22   echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       autoUpdate"
23   exit 1
24 fi
25
26 if [ -z "${CVE_URL}" ]; then
27   echo "Missing env variable CVE_URL. Please provide a Base Data
       Mirror URL for CVE 1.2."
28   echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       cveUrlBase"
29   exit 1
30 fi
31
32 if [ -z "${RETIREJS_URL}" ]; then
33   echo "Missing env variable RETIREJS_URL. Please provide The
       Retire JS repository URL."
34   echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       retireJsUrl"
35   exit 1
36 fi
37
```

```
38 if [ -z "${KEV_URL}" ]; then
39   echo "Missing env variable KEV_URL. Please provide the mirror
       URL to the CISA Known Exploited Vulnerabilities JSON datafeed..
       "
40   echo "Documentation at: https://jeremylong.github.io/
       DependencyCheck/dependency-check-maven/check-mojo.html#
       knownExploitedUrl"
41   exit 1
42 fi
43
44 set +e
45
46 mvn -Dformats=HTML,JSON -DossindexAnalyzerEnabled=false \
47  -DcveUrlBase="${CVE_URL}" \
48   -DretireJsUrl="${RETIREJS_URL}" \
49   -DknownExploitedUrl="${KEV_URL}" \
50   -DautoUpdate="${AUTO_UPDATE}" \
51 clean jacoco:prepare-agent \
52 package jacoco:report org.owasp:dependency-check-maven:7.3.2:
       aggregate --batch-mode;
53
54 set -e
55
56 mvn checkstyle:checkstyle; \
57 JAVA_HOME=/usr/lib/jvm/java-11-openjdk/ mvn \
58  -Dsonar.projectKey="${SONAR_PROJECT_KEY}" \
59  -Dsonar.host.url=[SONAR_SERVER_URL] \
60  -Dsonar.login="${SONAR_PROJECT_TOKEN}" \
61  -Dsonar.dependencyCheck.summarize=true \
62  -Dsonar.dependencyCheck.jsonReportPath=target/dependency-check-
       report.json \
63  -Dsonar.dependencyCheck.htmlReportPath=target/dependency-check-
       report.html \
64  -Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml
       \
65  -Dsonar.scm.disabled=true \
66  org.sonarsource.scanner.maven:sonar-maven-plugin:3.9.1.2184:sonar
       --batch-mode
```

4.4.2 Immagine docker

Una volta definiti tool e parametri necessari dunque ho proceduto a creare un immagine docker a partire da quella usata per la build del progetto, alla quale ho anche aggiunto la openjdk-11 per supportare l'esecuzione del sonar scanner.

```
FROM [path/to/nexus]:[immagine-di-build]
#to execute the sonar scan process java 11 or higher is required (
    https://docs.sonarqube.org/8.9/analyzing-source-code/moving-
   analysis-to-java-11/ )

RUN sudo yum install java-11-openjdk-devel -y
COPY settings.xml /home/build/.m2/
RUN chown build:build /home/build/.m2/settings.xml
ADD scripts/sonar-scan-[libreria].sh "/opt/sonar-scan-[libreria].
   sh"
RUN chown build:build /opt/sonar-scan-[libreria].sh
RUN chmod +x /opt/sonar-scan-[libreria].sh

ENV CVE_URL=https://[path/to/nexus]/repository/nist_cve/nvdcve
   -1.1-%d.json.gz
ENV RETIREJS_URL=https://[path/to/nexus]/repository/retire_js/
   jsrepository.json
ENV KEV_URL=https://[path/to/nexus]/repository/cisa-known-vuln-
   expl/known_exploited_vulnerabilities.json
ENV AUTO_UPDATE=true
```

4.4.3 Pipeline

Infine ho ristrutturato la pipeline secondo le best practices indicate sul sito di Jenkins per quanto riguarda l'uso di pipeline multi-container [11] ottenendo:

```
pipeline {
    agent none
    stages {
        stage('build') {
            agent {
                docker {
                 label 'docker'
                 image "[nome-immagine-di-build:versione
   -immagine]"
                 registryUrl "[registry-aziendale]"
                 registryCredentialsId '[credenziali-
   registry]'
                 args "--user build [...]"
             }
```

```
13              }
14              steps {
15                  [...]
16              }
17          }
18          [...]
19          stage('Sonar scan') {
20              agent {
21                  docker {
22                    label 'docker'
23                      image '[immagine-di-scan:versione-
    immagine]'
24                    registryUrl "[registry-aziendale]"
25                    registryCredentialsId '[credenziali-
    registry]'
26                    args "--user build [...]"
27                   }
28              }
29              environment{
30                  SVNCRED = [...]
31                  NEXUSCRED = [...]
32                  SONAR_PROJECT_KEY=[...]
33                  SONAR_PROJECT_TOKEN=[...]
34              }
35              steps {
36                  sh 'whoami'
37                  sh 'env'
38                  sh "cat /opt/sonar-scan-[libreria].sh"
39                  sh "/opt/sonar-scan-[libreria].sh"
40              }
41          }
42       }
43
44  }
```

4.4.4 Quality Gate

A causa della gestione incrementale delle versioni del progetto, per superare la limitazione dovuta alla non-disponibilità della branch analysis si è optato per la

creazione di un singolo progetto sonar al quale inviare i report fatti in ordine delle versioni del progetto in modo da mantenere uno storico dell'evoluzione delle issues tra le versioni e poterne scegliere una come base per la definizione di "New Code". Il "New Code" dunque è stato definito come il codice introdotto a partire dall'analisi generata dalla versione precedente del progetto. Il Quality Gate è stato impostato su condizioni esclusivamente legate alle metriche del New Code ed è stato «rilassato» solo per quanto riguarda la metrica relativa alla soglia di coverage, portandola a 0.

4.5 Quarto progetto: "Progetto multi-modulo"

Il quarto progetto in cui è stato integrato sonar è la versione attiva del progetto di prova descritto nella sezione 3.1.1, come il precedente progetto "Libreria multi-modulo" è presente in più versioni, ognuna con il suo branch nominato [Libreria multi-modulo]_Versione (dove versione è un numero intero progressivo) e la sua pipeline corrispondente. Essendo questo un progetto di grosse dimensioni utile alla produzione di uno dei prodotti principali dell'azienda, ad ogni nuovo active branch/versione del prodotto corrisponde una nuova versione di questo progetto, di conseguenza con il susseguirsi degli sprint vengono progressivamente generati nuovi branch e abbandonate le versioni vecchie.

L'implementazione della soluzione è rimasta quella descritta nella sezione 3.3

4.5.1 Quality Gate

Analogamente al caso della "Libreria multi-modulo", sfruttando la gestione incrementale delle versioni del progetto, per superare la limitazione dovuta alla non-disponibilità della branch analysis si è optato per la creazione di un singolo progetto sonar al quale inviare i report fatti in ordine delle versioni del progetto in modo da mantenere uno storico dell'evoluzione delle issues tra le versioni e poterne scegliere una come base per la definizione di "New Code". Il "New Code" dunque è stato definito come il codice introdotto a partire dall'analisi generata dalla versione precedente del progetto. Il Quality Gate è stato impostato su condizioni esclusivamente legate alle metriche del New Code ed è stato «rilassato» solo per quanto riguarda la metrica relativa alla soglia di coverage, portandola a 0.

Chapter 5

Risultati

Il periodo di osservazione della durata di un mese, seguente all'introduzione di SonarQube nelle pipelines dei quattro progetti di test, ha portato alla luce un parziale successo e delle problematiche legate al processo di utilizzo di questo strumento.

Successi: l'integrazione con i progetti "Modulo 1" e "Modulo 2" di complessità e dimensioni della codebase medio-basse (migliaia-decine di migliaia di LOC) è progredita, dopo una settimana di utilizzo i developers erano sufficientemente a loro agio da poter rendere lo stage bloccante: lo stage di sonar è stato spostato come primo stage della pipeline ed è stato impostato in modo da fallire (e dunque far fallire la pipeline intera) in caso di fallimento del quality gate. I team dunque hanno deciso di impegnarsi a non considerare accettabile la consegna di codice che non passa il quality gate e si sono assegnati come obiettivo l'introduzione graduale di una soglia sempre maggiore di coverage fino a raggiungere l'80% (corrispondente al valore del Quality Gate "Sonar Way"). Per fare ciò, la definizione del "New Code" deve essere modificata prendendo come riferimento l'ultima analisi fatta prima di ogni cambiamento della soglia di coverage. Infine SonarQube è stato aggiunto a pipelines anche di altri progetti (sia su Jenkins che su GitlabCI)

Difficoltà: Le problematiche evidenziate invece si possono riassumere in questo modo:

- esclusione dell'utilizzo di SonarQube dalle task assegnate ai developers: le task

definite per ogni sprint si possono dividere in task di sviluppo e di bugfixing di bug riportati dai team di quality assurance. Non sono presenti dunque task specifici riguardanti la risoluzione delle issues di SonarQube, questa attività viene lasciata all'iniziativa del developer.

- assenza della funzionalità di branch analysis: la community edition di sonar non presenta la possibilità di far confluire l'analisi di più branch di uno stesso progetto git/svn in un unico progetto sonar. Le conseguenze sono due a seconda del workaround utilizzato:
  - creare un progetto sonar per ogni branch: questo workaround comporta un esplosione dei progetti su sonar, inoltre i developer devono creare un progetto sonar e fare il setup richiesto ad ogni creazione di un nuovo branch, inoltre devono ricordarsi di eliminare progetti sonar corrispondenti a branch abbandonati o eliminati: ciò rappresenta un notevole overhead di lavoro che non gli viene riconosciuto e dunque un forte disincentivo.
  - spostare l'analisi sulla versione più recente del progetto:
    - è il workaround utilizzato nei due progetti "libreria multi modulo" e "Progetto multi modulo".
    - evita il mantenimento di numerosi progetti sonar in contemporanea e permette di mantenere una issues history progressiva nelle versioni del progetto ma il problema è che non solo la versione più recente del progetto è un «active branch»: versioni più vecchie subiscono interventi di bugfixing e di implementazione di features che dopo periodi di tempo medio-lunghi (mesi) possono essere introdotti nel branch più recente tramite merge. Tutte le attività che avvengono nelle versioni precedenti del progetto non sono dunque analizzate fino a che non impattano la versione corrente.
- lo stage ha un grosso impatto percentuale (attorno al 50%) nei tempi di completamento delle pipeline

Soluzione proposta Per superare le problematiche evidenziate, si identificano tre interventi possibili:

- l'adozione della funzionalità di branch analysis tramite l'uso del plugin open source "sonarqube-community-branch-plugin" che non è mantenuto da SonarSource o tramite l'acquisto di licenze SonarQube "Developer".
- l'introduzione di task ricorrenti all'interno dello sprint (ogni settimana), assegnate ad ogni developer del team e con una specifica durata (partire da

un tempo standard per ogni team e poi cambiarlo in base alla relazione tra il numero di issues e gli obiettivi di qualità da raggiungere) dedicate nello specifico alla risoluzione delle issues di SonarQube.

- uso di stage di analisi paralleli agli altri nelle pipeline dei branch di sviluppo, mantenendoli bloccanti (dunque per forza in serie) solo nelle pipeline dei branch master

Chapter 6

Lavori Futuri

È possibile continuare il lavoro in molti modi, per quanto riguarda l'introduzione di sonar nello specifico si potrebbe studiare un metodo di creazione degli stage di analisi meno articolato in modo da renderlo facilmente accessibile a tutti i developer, generando delle immagini di analisi comuni per tutti i progetti invece che specifiche per il progetto ad esempio, dividendo gli step di generazione dei report di terze parti e di test da quello di sonar e precedenti ad esso, considerando l'eliminazione degli script bash nelle immagini di analisi in favore di pipeline più lunghe ma con step più semplici. Inoltre si potrebbero spostare i parametri di analisi in un file sonar.properties da tenere nei repository in modo da renderli più facilmente impostabili dagli sviluppatori. Si potrebbe inoltre raccogliere e catalogare i dati riguardanti lo storico delle attività di bugfixing/bug reporting per identificare aree del codice/moduli ai quali dare priorità di intervento, modulando gli obiettivi da raggiungere in termini di risoluzione issues nel tempo e misurando l'effort effettivamente speso per la risoluzione delle issues dai team per compararlo con quello predetto da sonarqube, in modo da costruire un modello predittivo migliore per il raggiungimento degli obiettivi e basare su di esso il numero e la durata delle task da assegnare ai developers per il refactoring. Guardando ad un più ampio obiettivo legato alla gestione del technical debt, da una parte si potrebbe definire un processo di code review strutturato e standard per tutti i team, dall'altro si potrebbe alimentare una "cultura del testing", possibilmente introducendo metodologie di sviluppo agile come il Test Driven Development (nel caso di team piccoli) o il Behavioural Driven Development.

Bibliography

[1] Sofia Feist, Bruno Ferreira, and António Leitão. «Collaborative Algorithmic-based Building Information Modelling». In: Apr. 2017. DOI: 10.52842/conf.caadria.2017.613.

[2] Cansin Cagan Acarer. *CI/CD – Continuous Integration, Continuous Delivery, and Continuous Deployment*. 2020. URL: https://cacarer.com/ci-cd-continuous-integration-continuous-delivery-and-continuous-deployment/.

[3] SonarSource S.A. URL: https://docs.sonarqube.org/9.7/setup-and-upgrade/install-the-server/.

[4] Tom Gregory. *SonarQube branch analysis*. 2021. URL: https://tomgregory.com/sonarqube-branch-analysis/.

[5] G. Ann Campbell. *What is 'taint analysis' and why do I care?* SonarSource S.A. 2020. URL: https://www.sonarsource.com/blog/what-is-taint-analysis/.

[6] OWASP. URL: https://owasp.org/about/.

[7] MITRE. URL: https://cve.mitre.org/.

[8] CISA. URL: https://www.cisa.gov/known-exploited-vulnerabilities-catalog.

[9] NIST. URL: https://nvd.nist.gov/vuln.

[10] Checkstyle. URL: https://checkstyle.org/.

[11] *Using multiple containers*. Jenkins. URL: https://www.jenkins.io/doc/book/pipeline/docker/#using-multiple-containers.

[12] Ph.D. Gregory Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Tech. rep. 2002. URL: https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf.

[13] SonarSource S.A. URL: https://www.sonarsource.com/solutions/our-unique-approach/.

[14] OWASP. URL: https://jeremylong.github.io/DependencyCheck/dependency-check-cli/index.html.

[15] SonarSource. URL: https://github.com/SonarSource/sonar-scanner-npm.

[16] OWASP. URL: https://jeremylong.github.io/DependencyCheck/dependency-check-maven/.

[17] Checkstyle. URL: https://maven.apache.org/plugins/maven-checkstyle-plugin/.

[18] JaCoCo. URL: https://mvnrepository.com/artifact/org.jacoco/jacoco-maven-plugin.

[19] SonarSource. URL: https://docs.sonarqube.org/9.7/analyzing-source-code/scanners/sonarscanner-for-maven/.