Experiment No 8

Aim: Create a Jenkins CICD Pipeline with SonarQube / GitLab Integration to perform a static analysis of the code to detect bugs, code smells, and security vulnerabilities on a sample Web / Java / Python application.

Theory:

What is SAST?

Static application security testing (SAST), or static analysis, is a testing methodology that analyzes source code to find security vulnerabilities that make your organization's applications susceptible to attack. SAST scans an application before the code is compiled. It's also known as white box testing.

What problems does SAST solve?

SAST takes place very early in the software development life cycle (SDLC) as it does not require a working application and can take place without code being executed. It helps developers identify vulnerabilities in the initial stages of development and quickly resolve issues without breaking builds or passing on vulnerabilities to the final release of the application.

SAST tools give developers real-time feedback as they code, helping them fix issues before they pass the code to the next phase of the SDLC. This prevents security-related issues from being considered an afterthought. SAST tools also provide graphical representations of the issues found, from source to sink. These help you navigate the code easier. Some tools point out the exact location of vulnerabilities and highlight the risky code. Tools can also provide in-depth guidance on how to fix issues and the best place in the code to fix them, without requiring deep security domain expertise.

It's important to note that SAST tools must be run on the application on a regular basis, such as during daily/monthly builds, every time code is checked in, or during a code release.

Why is SAST important?

Developers dramatically outnumber security staff. It can be challenging for an organization to find the resources to perform code reviews on even a fraction of its applications. A key strength of SAST tools is the ability to analyze 100% of the codebase. Additionally, they are much faster than manual secure code reviews performed by humans. These tools can scan millions of lines of code in a matter of minutes. SAST tools automatically identify critical vulnerabilities—such as

buffer overflows, SQL injection, cross-site scripting, and others—with high confidence.

What is a CI/CD Pipeline?

CI/CD pipeline refers to the Continuous Integration/Continuous Delivery pipeline. Before we dive deep into this segment, let's first understand what is meant by the term ‘pipeline’?

A pipeline is a concept that introduces a series of events or tasks that are connected in a sequence to make quick software releases. For example, there is a task, that task has got five different stages, and each stage has got some steps. All the steps in phase one have to be completed, to mark the latter stage to be complete.

Now, consider the CI/CD pipeline as the backbone of the DevOps approach. This Pipeline is responsible for building codes, running tests, and deploying new software versions. The Pipeline executes the job in a defined manner by first coding it and then structuring it inside several blocks that may include several steps or tasks.

What is SonarQube?

SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality. Sonar does static code analysis, which provides a detailed report of bugs, code smells, vulnerabilities, code duplications.

It supports 25+ major programming languages through built-in rulesets and can also be extended with various plugins.

Benefits of SonarQube

- **Sustainability -** Reduces complexity, possible vulnerabilities, and code duplications, optimising the life of applications.
- **Increase productivity -** Reduces the scale, cost of maintenance, and risk of the application; as such, it removes the need to spend more time changing the code
- **Quality code -** Code quality control is an inseparable part of the process of software development.
- **Detect Errors -** Detects errors in the code and alerts developers to fix them automatically before submitting them for output.
- **Increase consistency -** Determines where the code criteria are breached and enhances the quality

- **Business scaling -** No restriction on the number of projects to be evaluated
- **Enhance developer skills -** Regular feedback on quality problems helps developers to improve their coding skills

Integrating Jenkins with SonarQube:

Prerequisites:

- Jenkins installed
- Docker Installed (for SonarQube)
- SonarQube Docker Image **Integrating Jenkins with SonarQube.**

Open up Jenkins Dashboard on localhost, port 8080

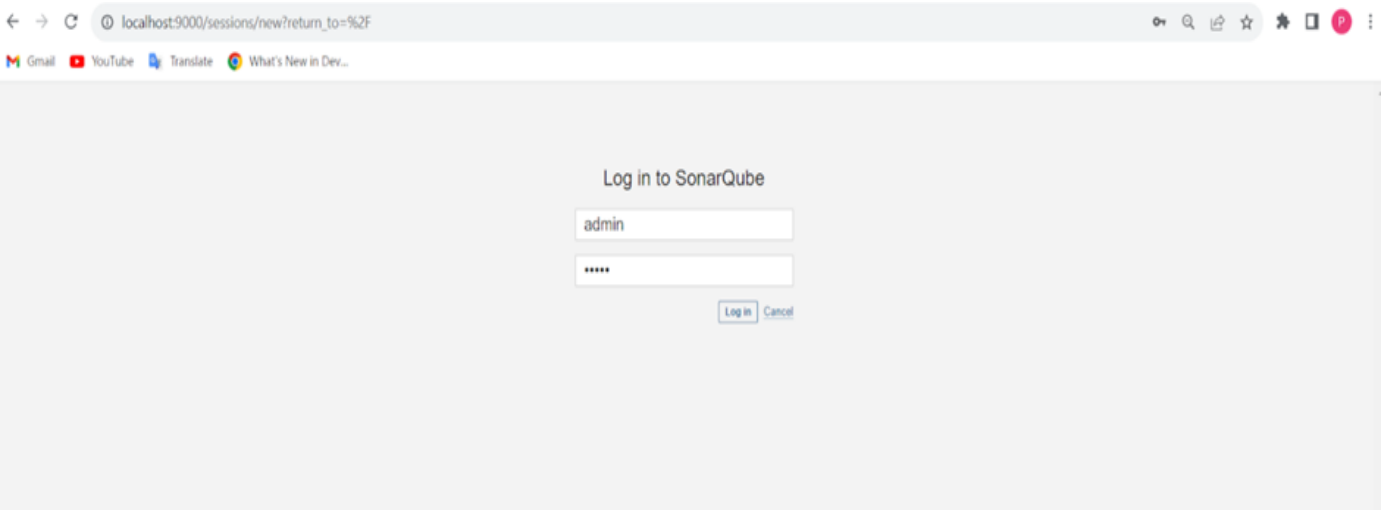
Run SonarQube in a Docker container using this command -

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

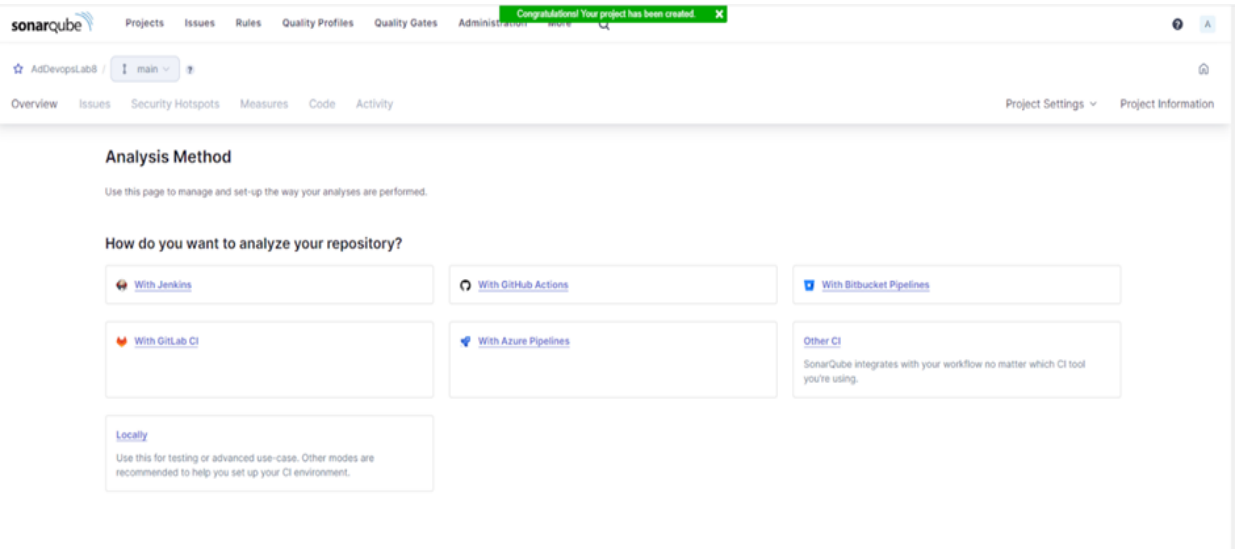
Sonarqube:latest

```
PS D:\Desktop\DockerFile> docker run -d --name sonarqube -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true -p 9000:9000
e:latest
Unable to find image 'sonarqube:latest' locally
latest: Pulling from library/sonarqube
44ba2882f8eb: Pull complete
2cabec57fa36: Pull complete
c20481384b6a: Pull complete
bf7b17ee74f8: Pull complete
38617faac714: Pull complete
b795b715553d: Pull complete
c5244f6c9231: Pull complete
Digest: sha256:1ffd122cfb37ce982289dc7f5d38bb702ba05af7b5a50f7cb077ae25e60b5b9a
Status: Downloaded newer image for sonarqube:latest
1442c4e613b25aaedec05c060f020a00802b1c6dbaa27e8c5c0dad4ed8fc1f76
```

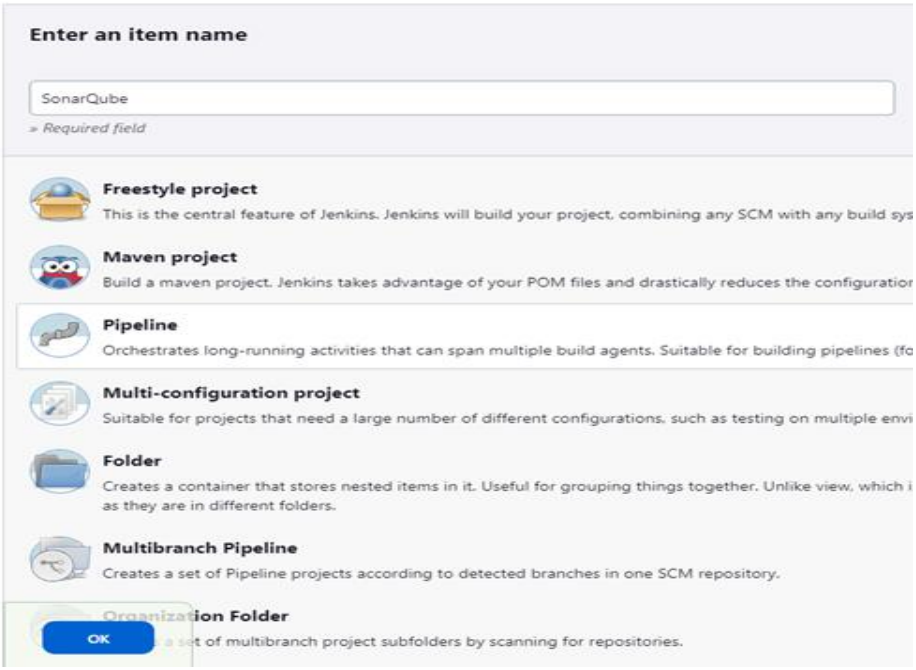
sonarqube



Create project manually



In Jenkins create a pipeline here named “SonarQube”



Enter the following in pipeline script:

```
node {
   stage('Cloning the GitHub Repo') {
      git 'https://github.com/PrajaktaUpadhye6/MSBuild_firstproject.git'
         }
   stage('SonarQube analysis') {
      withSonarQubeEnv('sonarqube') {
         bat "D:/sonar-scanner-cli-5.0.1.3006-windows/sonar-scanner-5.0.1.3006-windows/bin/sonar-scanner.bat \
            -D sonar.login=admin \
            -D sonar.password=abc \
            -D sonar.projectKey=AdDevops \
```

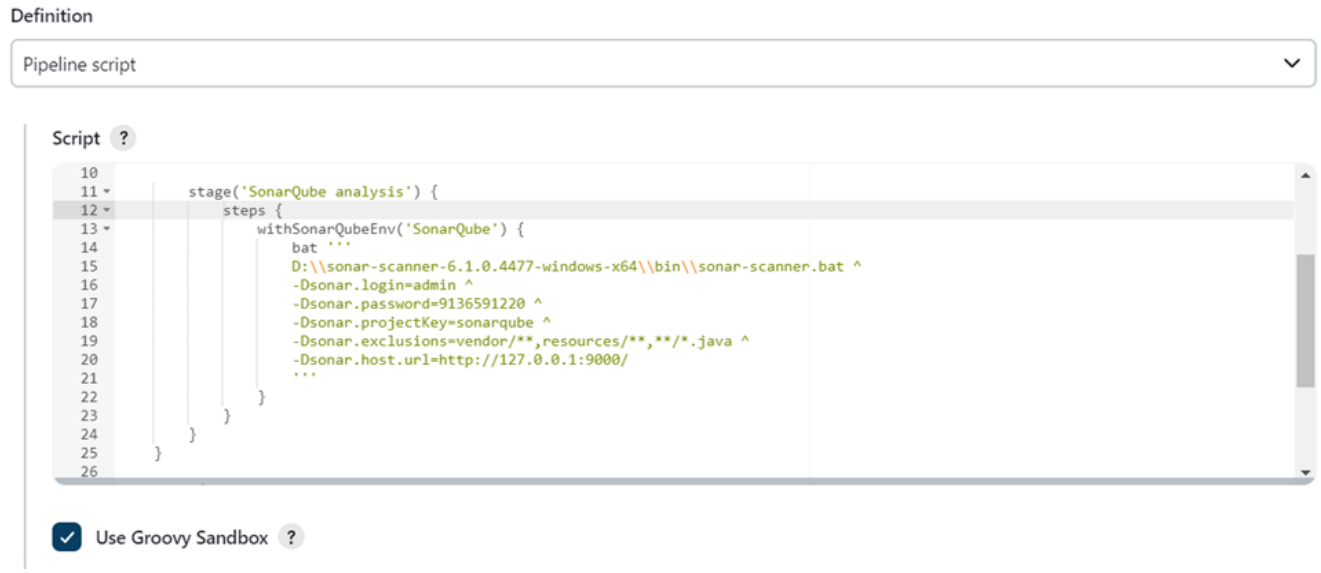
-D sonar.exclusions=vendor/,resources/**,**/*.java **

-D sonar.host.url=http://127.0.0.1:9000/"

}

}

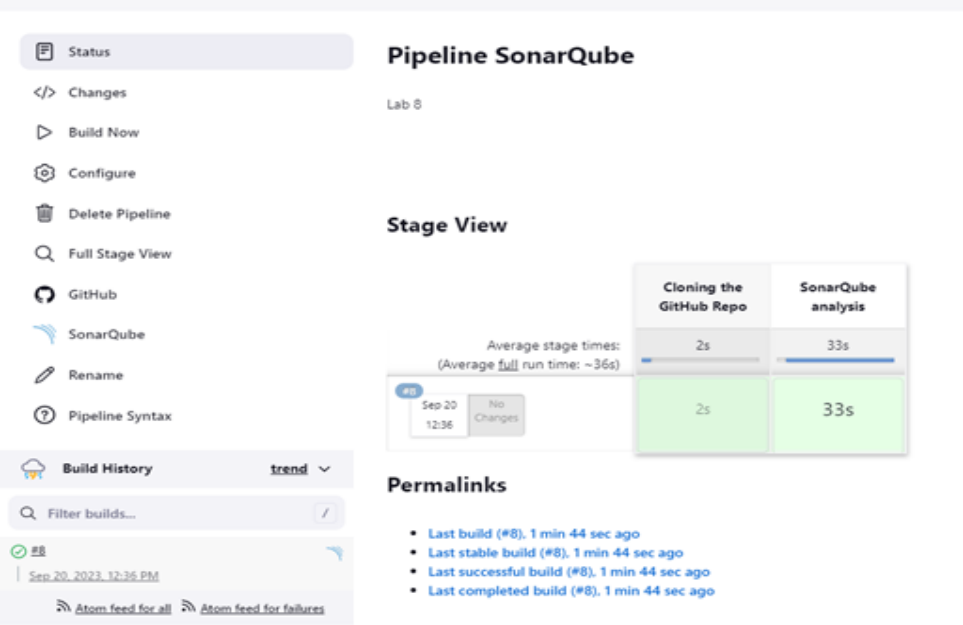
}



It is a java sample project which has a lot of repetitions and issues that will be detected by

SonarQube.

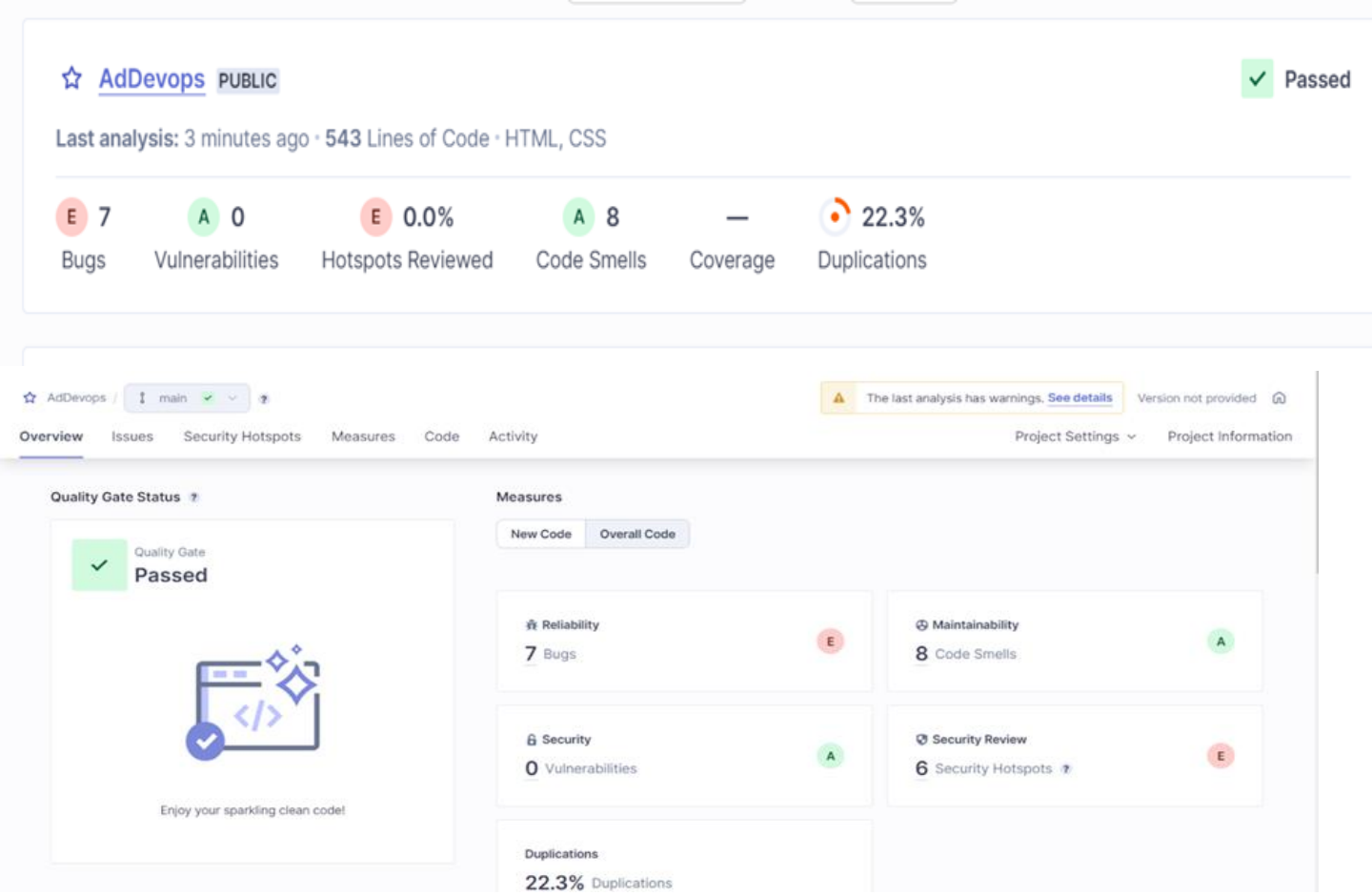
Build and run:



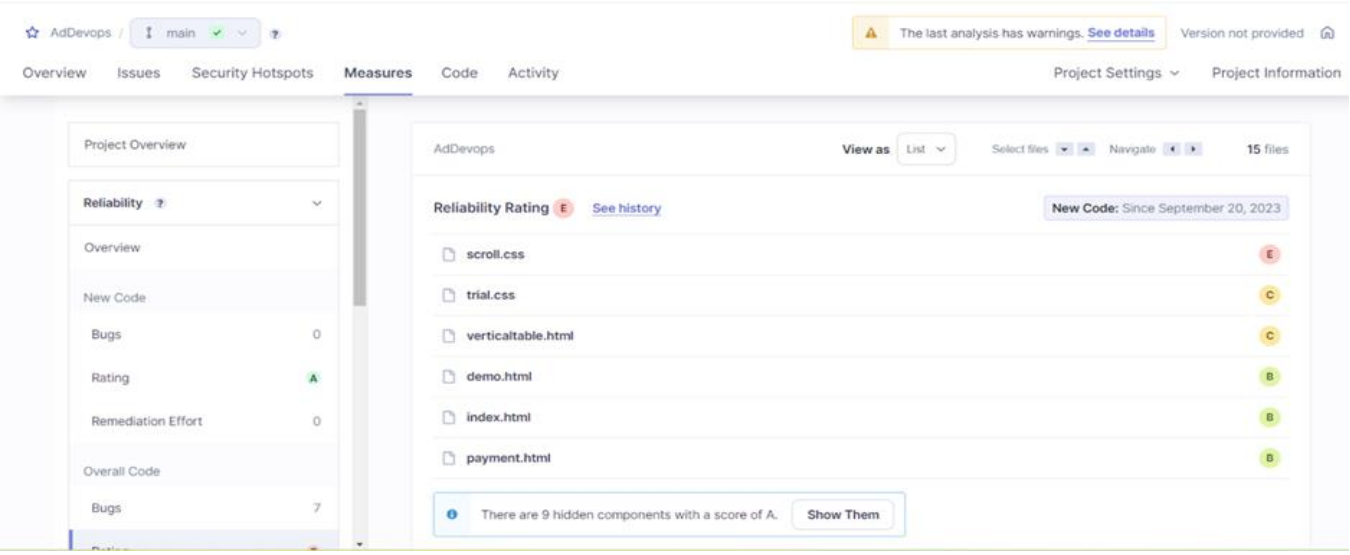
Console output:

```
17:49:39.413 INFO  CPD Executor CPD calculation finished (done) | time=159837ms
17:49:39.428 INFO  SCM revision ID 'ba799ba7e1b576f04a4612322b0412c5e6e1e5e4'
17:52:16.075 INFO  Analysis report generated in 4781ms, dir size=127.2 MB
17:52:33.696 INFO  Analysis report compressed in 17599ms, zip size=29.6 MB
17:52:34.987 INFO  Analysis report uploaded in 1290ms
17:52:34.989 INFO  ANALYSIS SUCCESSFUL, you can find the results at: http://127.0.0.1:9000/dashboard?id=sonarqube
17:52:34.989 INFO  Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
17:52:34.989 INFO  More about the report processing at http://127.0.0.1:9000/api/ce/task?id=8c04859e-050f-41e5-b320-ee32d44d4259
17:52:43.609 INFO  Analysis total time: 16:49.875 s
17:52:43.612 INFO  SonarScanner Engine completed successfully
17:52:44.416 INFO  EXECUTION SUCCESS
17:52:44.419 INFO  Total time: 17:04.839s
[Pipeline] }
[Pipeline] // withSonarQubeEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Pipeline completed successfully!
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

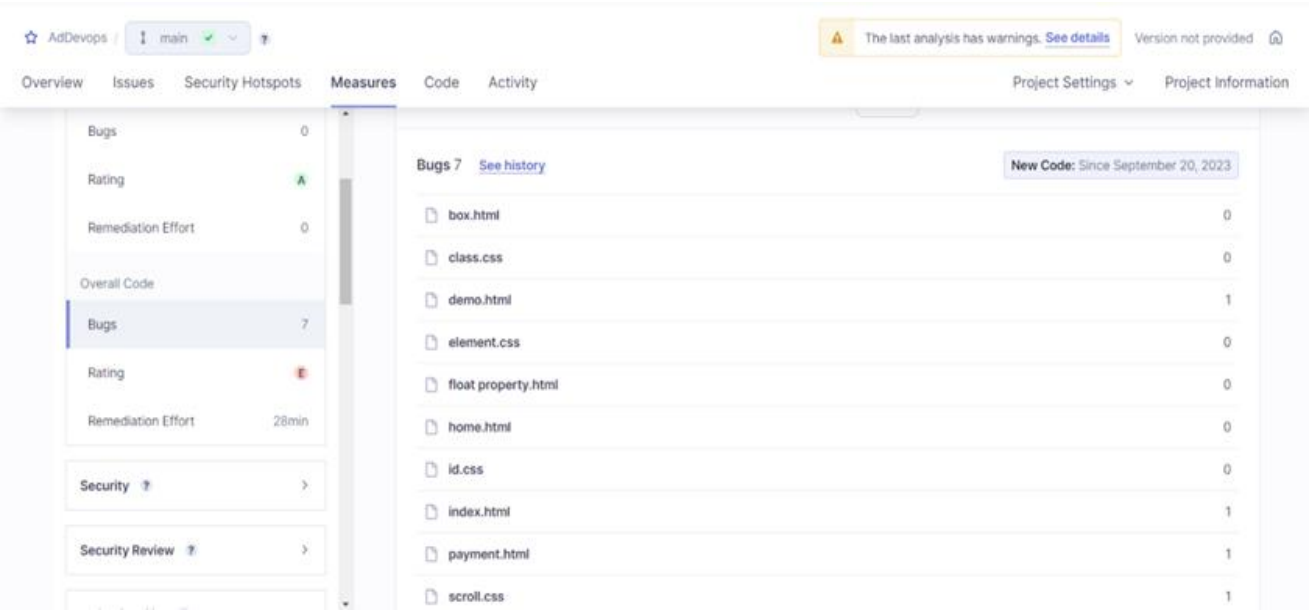
sonarqube:



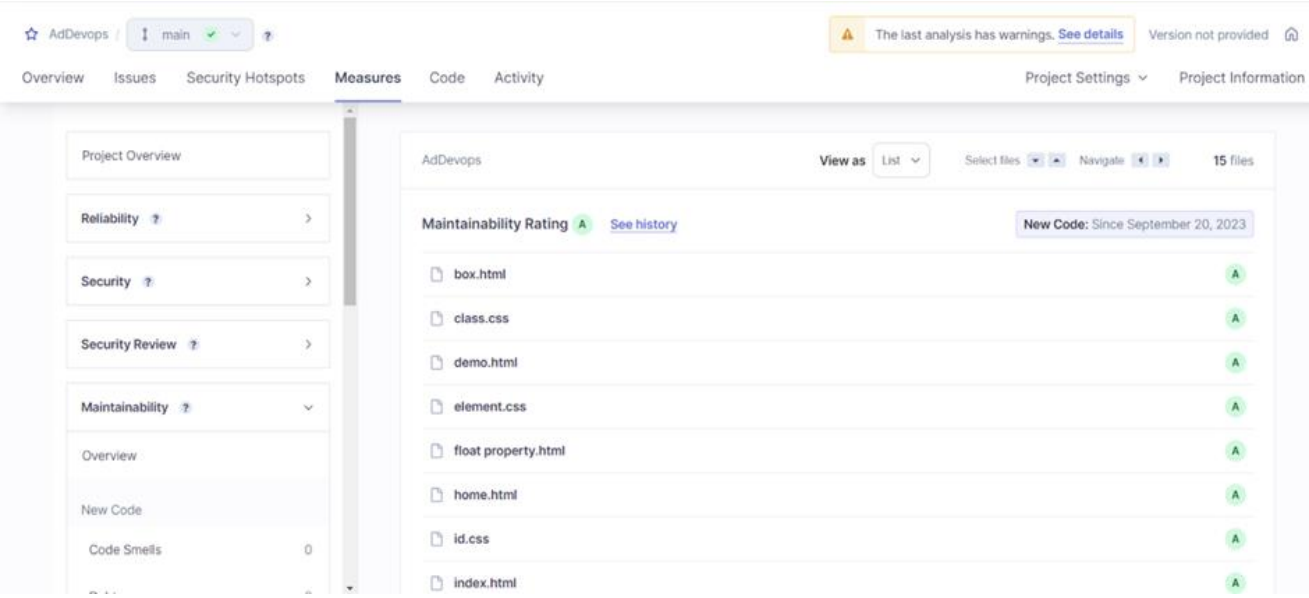
Reliability:



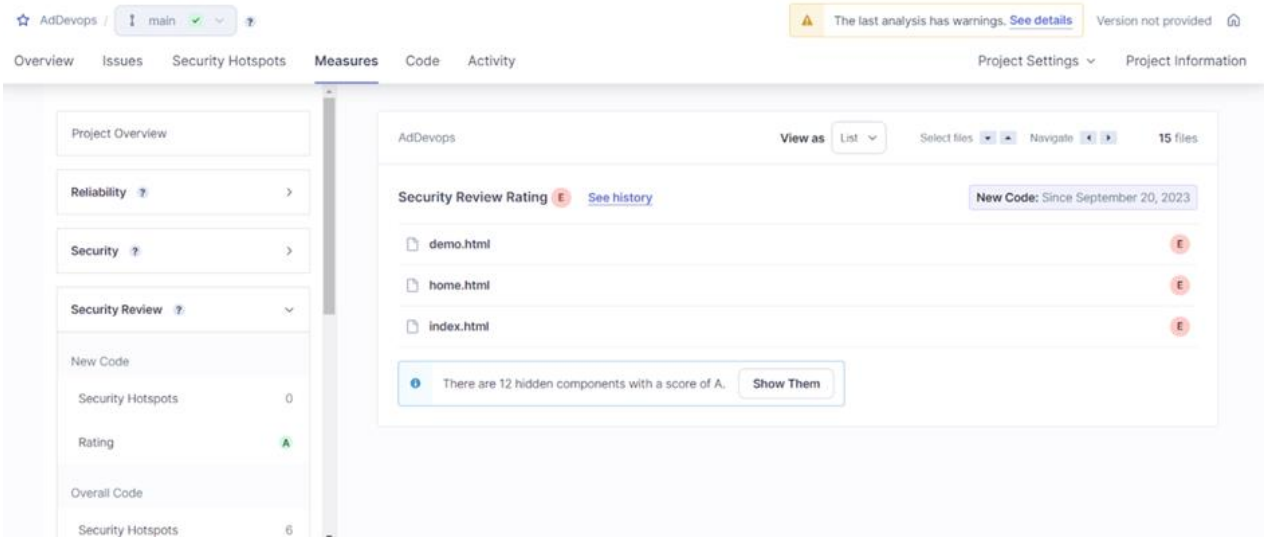
Bugs:



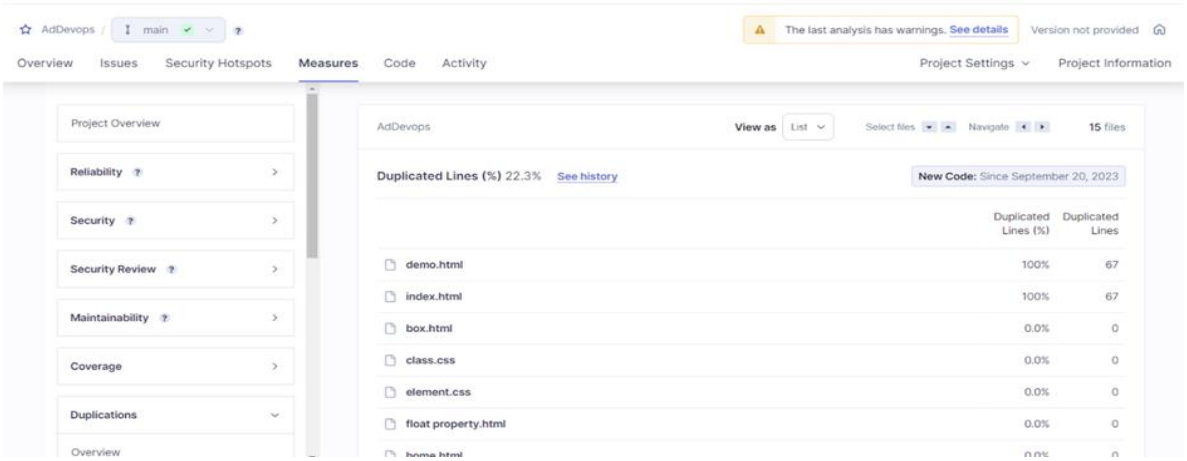
Maintainability:



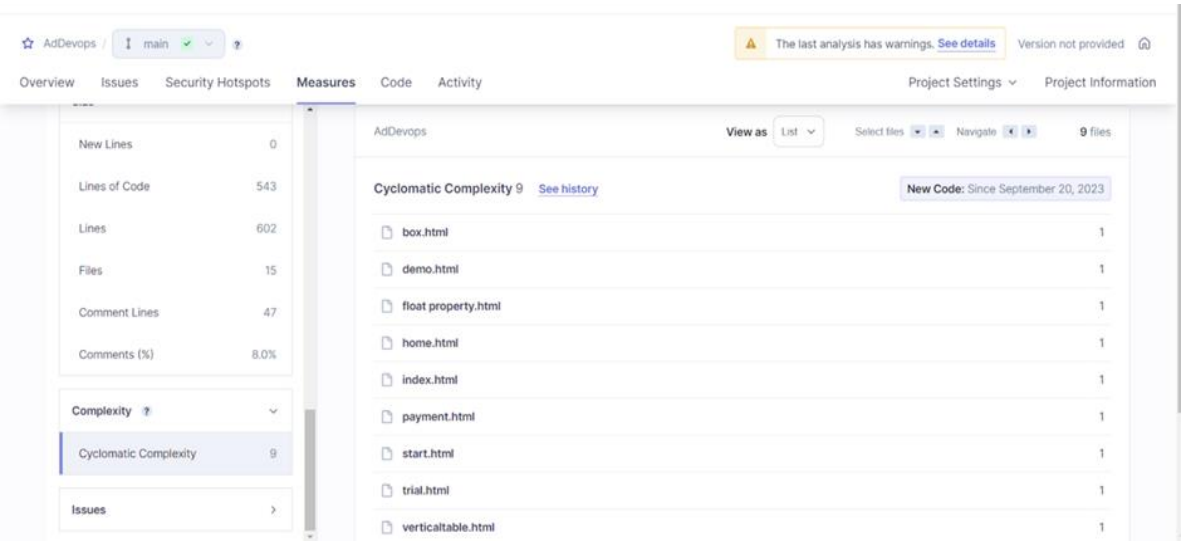
Security:



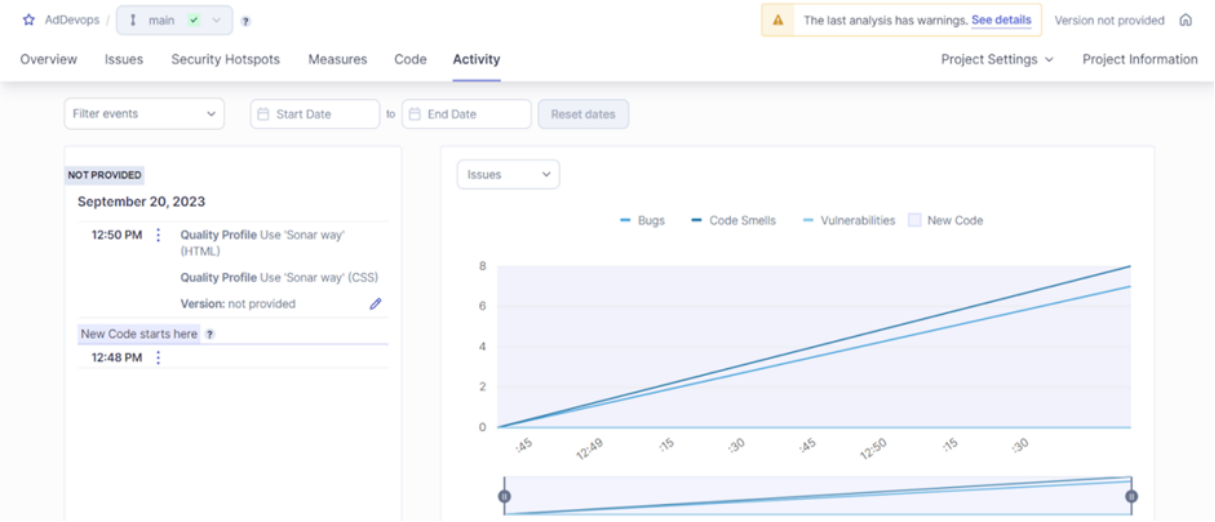
Duplications:



Cyclomatic complexity:



Activity:



Conclusion:

In this experiment, we performed a static analysis of the code to detect bugs, code smells, and security vulnerabilities on our sample Java application.