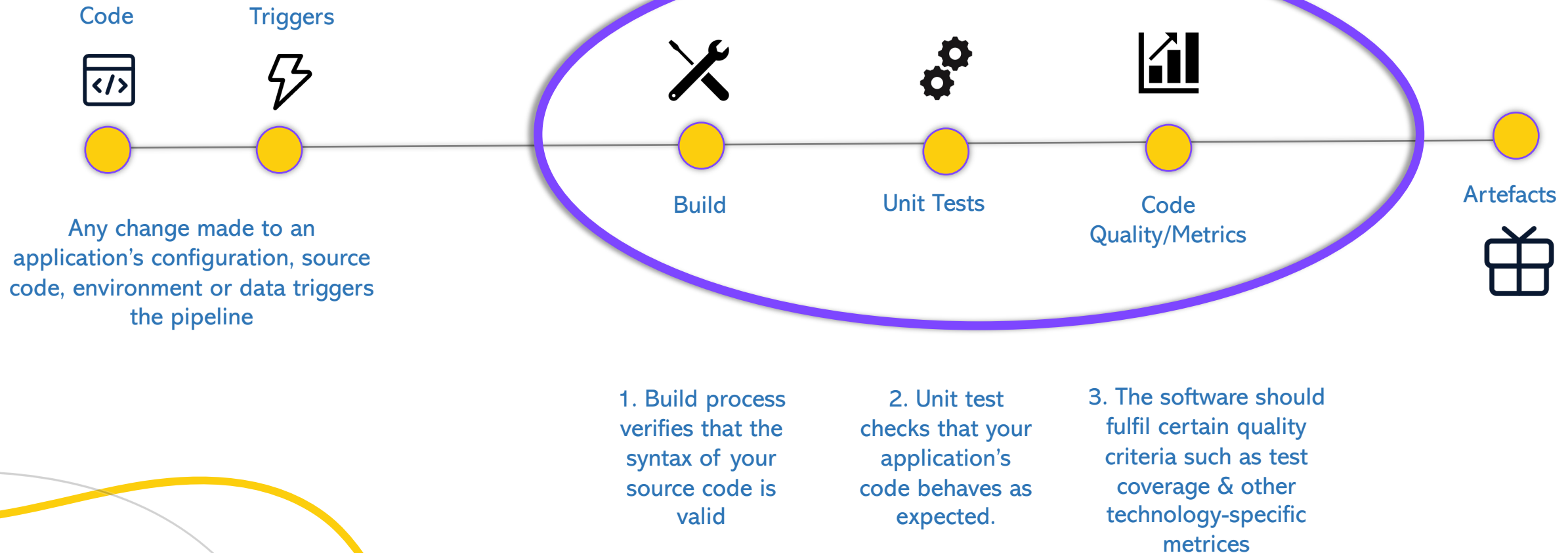




Sonarqube

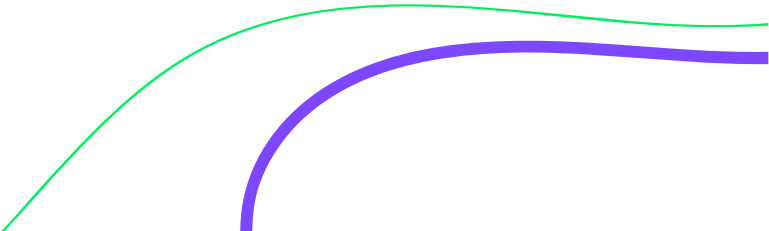
Code Quality and Code Security

Continuous Integration

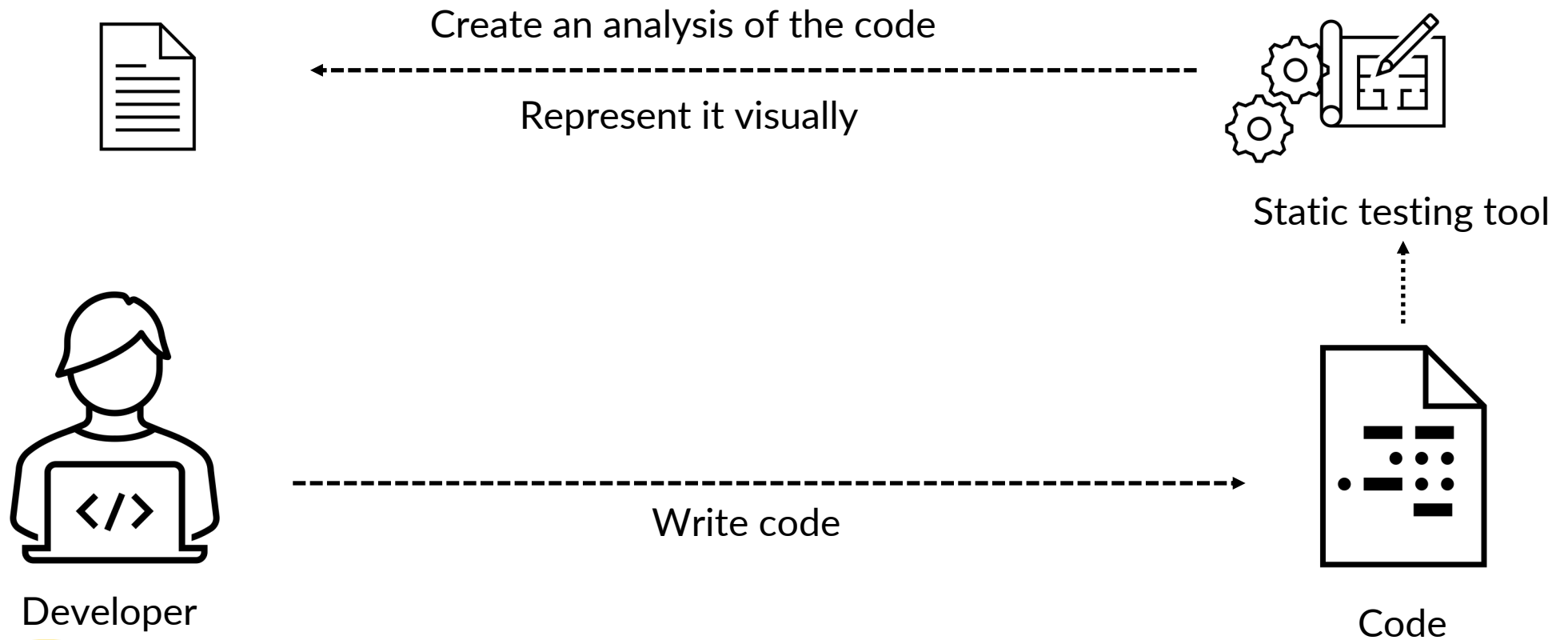




Static Testing Vs Dynamic Testing

- Static testing is a method of debugging by examining source code before the program is executed, i.e. test code without executing it.
 - It does so by analyzing the code against a pre-set of coding rules and ensuring that it conforms to the guidelines.
 - This is done by a static code analysis tool like Sonarqube
 - Dynamic testing happens during the execution of the code.
 - It can help identify defects or vulnerabilities at a level where our application integrates with other systems like databases, servers and services.
 - Usually done by an expert QA
- 

Static Code Analysis



Reason to use static code analysis

Finds error earlier in the development

Detects over complexity in the code
(Refactoring/Simplification)

Find security errors

Enforce best coding practice

Automation & integration with CI/CD tools

Project specific rules



Sonarqube

Release Quality Code Every. Time.

Quality Gates tells you whether your code is ready to release at every analysis.

Maintainability = Productivity

Clean code means happier developers and higher team velocity

Code Security, for Developers

Detect security issues in code review with Static Application Security Testing (SAST)

Writing clean code

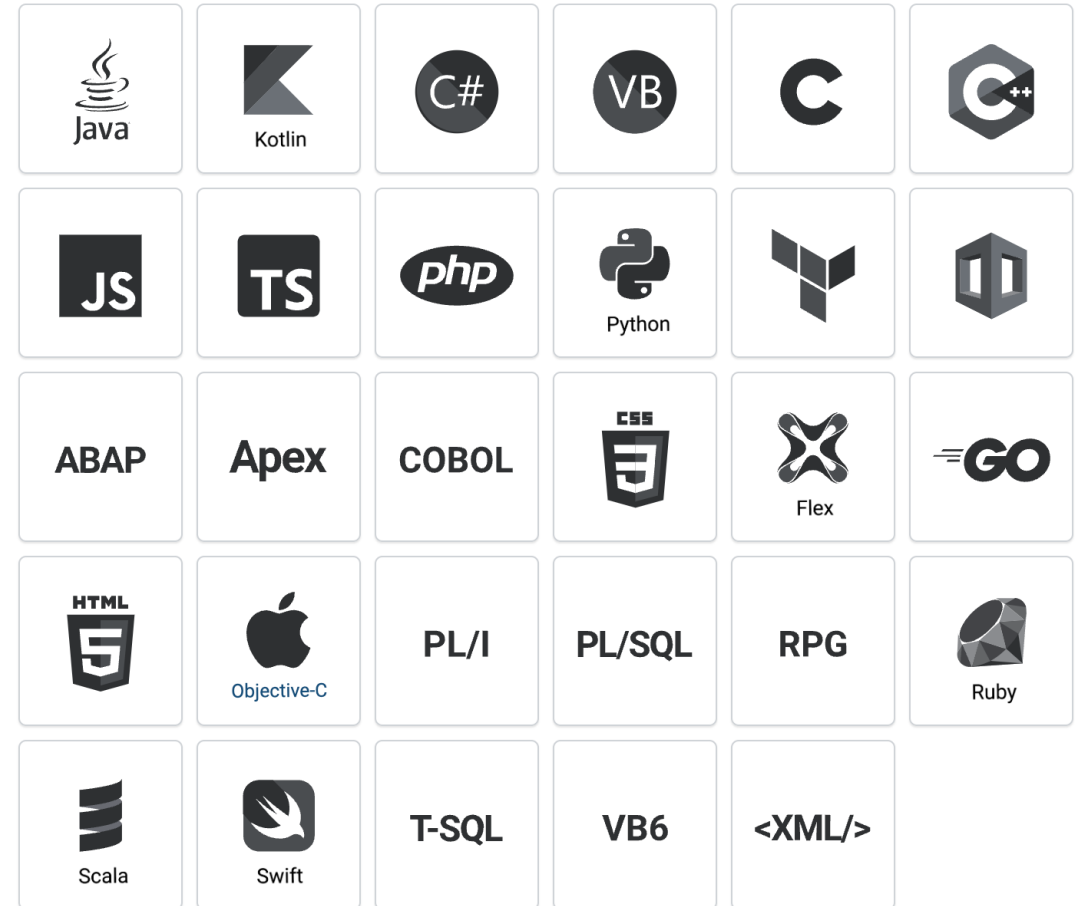
Writing Clean Code is essential to maintaining a healthy codebase.

Clean Code can be defined as code that meets a certain defined standard, i.e. code that is reliable, secure, maintainable, readable, and modular, in addition to having other key attributes.

This applies to all code: source code, test code, Infrastructure as Code, glue code, scripts, etc.

Sonarqube

- SonarQube supports **29 programming languages**
- With SonarQube static analysis you have one place to measure the Reliability, Security, and Maintainability of all the languages in your project, and all the projects in your sphere.



Sonarqube can help

Detect bugs

Detect Code Smells

Security Vulnerabilities

Activate rules needed

Execution path

Sonarqube can help

Automated code analysis

Integrate with CI/CD tools
with Webhooks and APIs

Integration with SCM tools
like Github

Analyze branches and
Decorate pull requests

Built-in methodologies

Discover memory leaks

Provides a good visualizer

Enforces a quality gate

Digs into issues

Provides plugins for IDEs
(Sonar lint)



Sonarqube License

1. Community Edition
2. Developer Edition
3. Enterprise Edition
4. Data Center





Community Edition

All the following features:

- ✓ Static code analysis for 17 languages
Java, C#, JavaScript, TypeScript, CloudFormation, Terraform, Kotlin, Ruby, Go, Scala, Flex, Python, PHP, HTML, CSS, XML and VB.NET
- ✓ Detect Bugs & Vulnerabilities
- ✓ Review Security Hotspots
- ✓ Track Code Smells & fix your Technical Debt
- ✓ Code Quality Metrics & History
- ✓ CI/CD integration
- ✓ Extensible, with 50+ community plugins

Developer Edition

Community Edition plus:

- ✓ C, C++, Obj-C, Swift, ABAP, T-SQL, PL/SQL support
- ✓ Detection of Injection Flaws in Java, C#, PHP, Python, JavaScript, TypeScript
- ✓ Analysis of feature and maintenance branches
- ✓ Pull Request decoration for:
 -  GitHub
 -  Bitbucket
 -  Azure DevOps
 -  GitLab

Enterprise Edition

Developer Edition plus:

- ✓ Portfolio Management & PDF Executive Reports
- ✓ Project PDF reports
- ✓ Security Reports
- ✓ Project Transfer
- ✓ Parallel processing of analysis reports
- ✓ Support for Apex, COBOL, PL/I, RPG, VB6

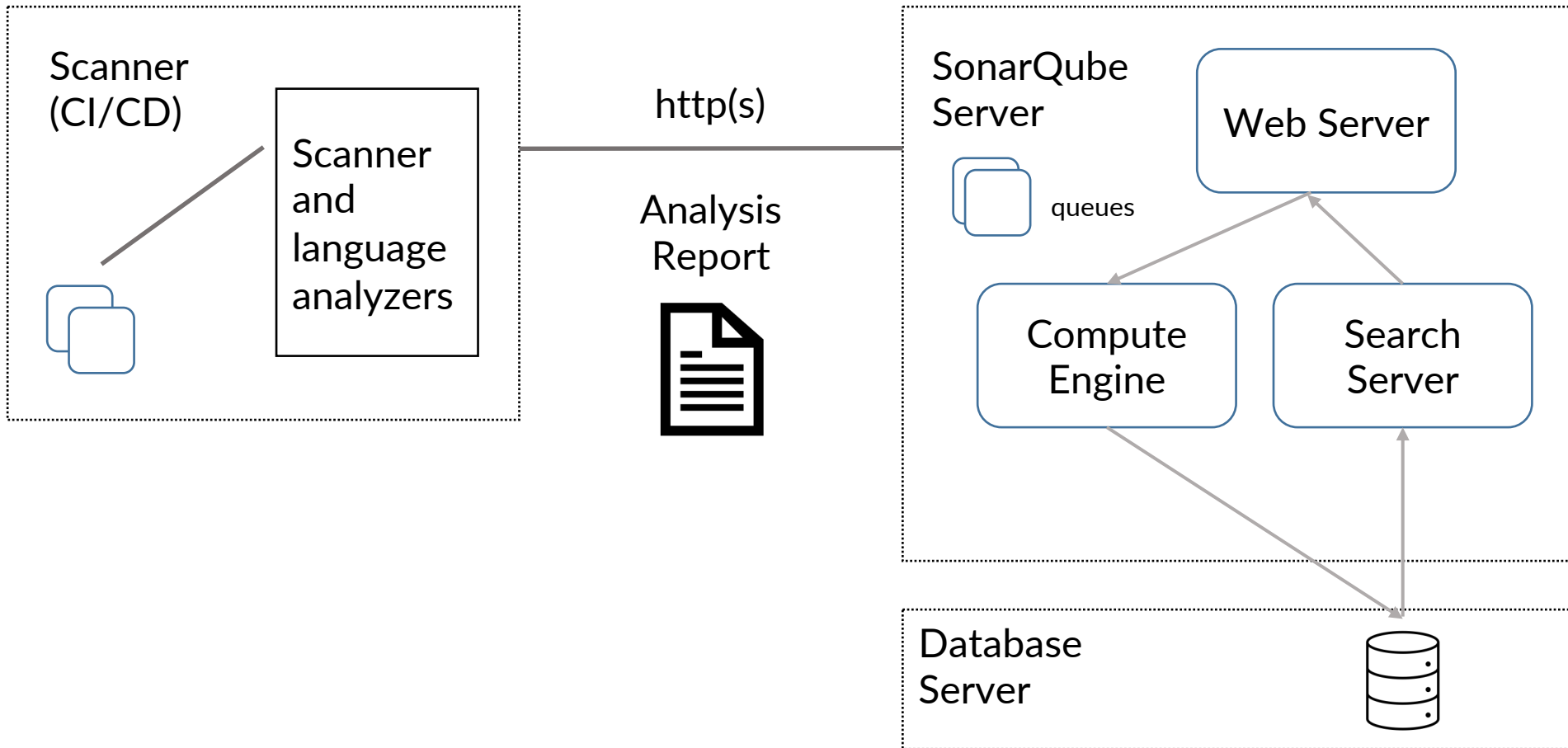
Data Center Edition

Enterprise Edition plus:

- ✓ Component redundancy
- ✓ Data resiliency
- ✓ Horizontal Scalability



SonarQube Architecture



Sonarqube

1. The SonarQube server runs the following processes:
 1. A **web server** that serves the SonarQube user interface.
 2. A search server based on **Elasticsearch**.
 3. The **compute engine** oversees processing code analysis reports and saving them in the SonarQube database.
2. The database to store the following:
 1. **Metrics and issues** for code quality and security generated during code scans.
 2. The SonarQube **instance configuration**.
3. One or more **scanners** running on your build or **continuous integration servers** to analyze projects.



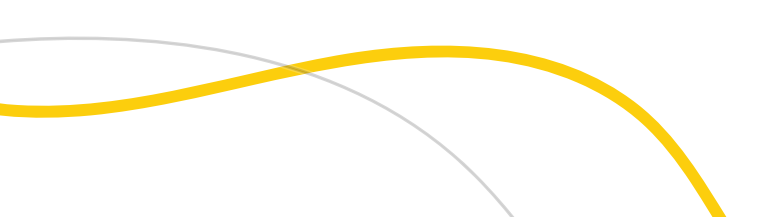
Issues lifecycle



Issues

While running an analysis, SonarQube raises an issue every time a code breaks a coding rule.

The set of coding rules is defined through the associated Quality Profile for each language in the project.



Issue Types

There are three types of issues:

1. **Bug** – A coding mistake that can lead to an error or unexpected behaviour at runtime.
2. **Vulnerability** – A point in your code that's open to attack.
3. **Code Smell** – A maintainability issue that makes your code confusing and difficult to maintain.

Issue Severity

Each issue has one of five severities:

1. **BLOCKER** -- Bug with a high probability to impact the behaviour of the application in production: memory leak, unclosed JDBC connection; **the code MUST be fixed immediately.**
2. **CRITICAL** -- Either a bug with a low probability to impact the behaviour of the application in production or an issue which represents a security flaw: empty catch block, SQL injection; **the code MUST be immediately reviewed.**
3. **MAJOR** -- Quality flaw which can highly impact the developer's productivity: an uncovered piece of code, duplicated blocks, unused parameters.
4. **MINOR** -- Quality flaw which can slightly impact the developer productivity: lines should not be too long, "switch" statements should have at least 3 cases
5. **INFO** -- Neither a bug nor a quality flaw, just a finding.

Issues Lifecycle

Statuses

After creation, issues flow through a lifecycle, taking one of the following statuses:

1. **Open** - set by SonarQube on new issues
2. **Confirmed** - set manually to indicate that the issue is valid
3. **Resolved** - set manually to indicate that the next analysis should Close the issue
4. **Reopened** - set automatically by SonarQube when a Resolved issue hasn't been corrected
5. **Closed** - set automatically by SonarQube for automatically created issues.

Resolutions

Closed issues will have one of the following resolutions:

1. **Fixed** - set automatically when a subsequent analysis shows that the issue has been corrected or the file is no longer available (removed from the project, excluded or renamed)
 2. **Removed** - set automatically when the related rule is no longer available. The rule may not be available either because it has been removed from the Quality Profile or because the underlying plugin has been uninstalled.
- Resolved issues will have one of the following resolutions:
 - **False Positive** - set manually
 - **Won't Fix** - set manually

Status

After creation, issues flow through a lifecycle, taking one of the following status:

1. **Open** - set by SonarQube on new issues
2. **Confirmed** - set manually to indicate that the issue is valid
3. **Resolved** - set manually to indicate that the next analysis should Close the issue
4. **Reopened** - set automatically by SonarQube when a Resolved issue hasn't been corrected
5. **Closed** - set automatically by SonarQube for automatically created issues.

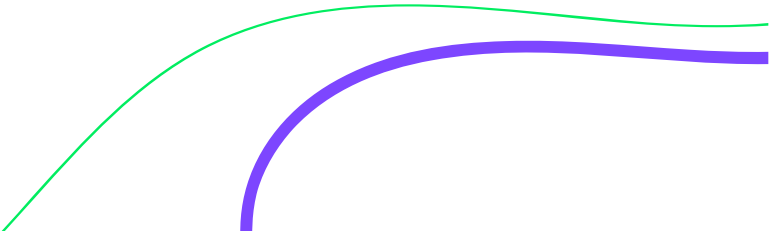
Issue Workflow

Issues are automatically closed (status: Closed) when:

- an issue (of any status) has been properly fixed = Resolution: Fixed
- an issue no longer exists because the related coding rule has been deactivated or is no longer available (ie: plugin has been removed) = Resolution: Removed
- Issues are automatically reopened (status: Reopened) when:
 - an issue that was manually Resolved as Fixed (but Resolution is not False positive) is shown by a subsequent analysis to still exist.



Metrics Definitions

1. Complexity
 2. Duplications
 3. Issues
 4. Maintainability
 5. Quality Gates
 6. Reliability
 7. Security
 8. Size
 9. Tests
- 

Complexity

It is the Cyclomatic Complexity calculated based on the number of paths through the code.

Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do.

Duplications

- Duplicated blocks (duplicated_blocks) -- Number of duplicated blocks of lines
- Duplicated files (duplicated_files) -- Number of files involved in duplications.
- Duplicated lines (duplicated_lines) -- Number of lines involved in duplications.
- Duplicated lines (%) (duplicated_lines_density) -- $\text{duplicated_lines} / \text{lines} * 100$

Maintainability

- **Code Smells** (code_smells) -- Total count of Code Smell issues.
- **New Code Smells** (new_code_smells) -- Total count of Code Smell issues raised for the first time on New Code.
- **Maintainability Rating** (sqale_rating)
(Formerly the SQALE rating.) Rating given to your project related to the value of your Technical Debt Ratio.
- The default Maintainability Rating grid is:
A=0-0.05, B=0.06-0.1, C=0.11-0.20, D=0.21-0.5, E=0.51-1

Maintainability

- **Technical Debt** (sqale_index)
Effort to fix all Code Smells. The measure is stored in minutes in the database. An 8-hour day is assumed when values are shown in days.
- **Technical Debt on New Code** (new_technical_debt)
Effort to fix all Code Smells raised for the first time on New Code.
- **Technical Debt Ratio** (sqale_debt_ratio)
Ratio between the cost to develop the software and the cost to fix it. The Technical Debt Ratio formula is:
$$\text{Remediation cost} / \text{Development cost}$$

Which can be restated as:
$$\text{Remediation cost} / (\text{Cost to develop 1 line of code} * \text{Number of lines of code})$$

The value of the cost to develop a line of code is 0.06 days.

Maintainability

- **Technical Debt Ratio on New Code** (`new_sqale_debt_ratio`)
Ratio between the cost to develop the code changed on New Code and the cost of the issues linked to it.

Quality Gates

- **Quality Gate Status** (alert_status)
State of the Quality Gate associated to your Project. Possible values are : ERROR, OK WARN value has been removed since 7.6.
- **Quality Gate Details** (quality_gate_details)
For all the conditions of your Quality Gate, you know which condition is failing and which is not.

Reliability

- **Bugs** (bugs) -- Number of bug issues.
- **New Bugs** (new_bugs) -- Number of new bug issues.
- **Reliability Rating** (reliability_rating)
 - A = 0 Bugs
 - B = at least 1 Minor Bug
 - C = at least 1 Major Bug
 - D = at least 1 Critical Bug
 - E = at least 1 Blocker Bug
- **Reliability remediation effort** (reliability_remediation_effort)
Effort to fix all bug issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.
- **Reliability remediation effort on new code** (new_reliability_remediation_effort)
Same as *Reliability remediation effort* but on the code changed on New Code.

Reliability

- **Bugs** (bugs) -- Number of bug issues.
- **New Bugs** (new_bugs) -- Number of new bug issues.
- **Reliability Rating** (reliability_rating)
 - A = 0 Bugs
 - B = at least 1 Minor Bug
 - C = at least 1 Major Bug
 - D = at least 1 Critical Bug
 - E = at least 1 Blocker Bug
- **Reliability remediation effort** (reliability_remediation_effort)
Effort to fix all bug issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.
- **Reliability remediation effort on new code** (new_reliability_remediation_effort)
Same as *Reliability remediation effort* but on the code changed on New Code.

Security

- **Vulnerabilities** (vulnerabilities)
Number of vulnerability issues.
- **Vulnerabilities on new code** (new_vulnerabilities)
Number of new vulnerability issues.
- **Security Rating** (security_rating)
 - A = 0 Vulnerabilities
 - B = at least 1 Minor Vulnerability
 - C = at least 1 Major Vulnerability
 - D = at least 1 Critical Vulnerability
 - E = at least 1 Blocker Vulnerability

Security

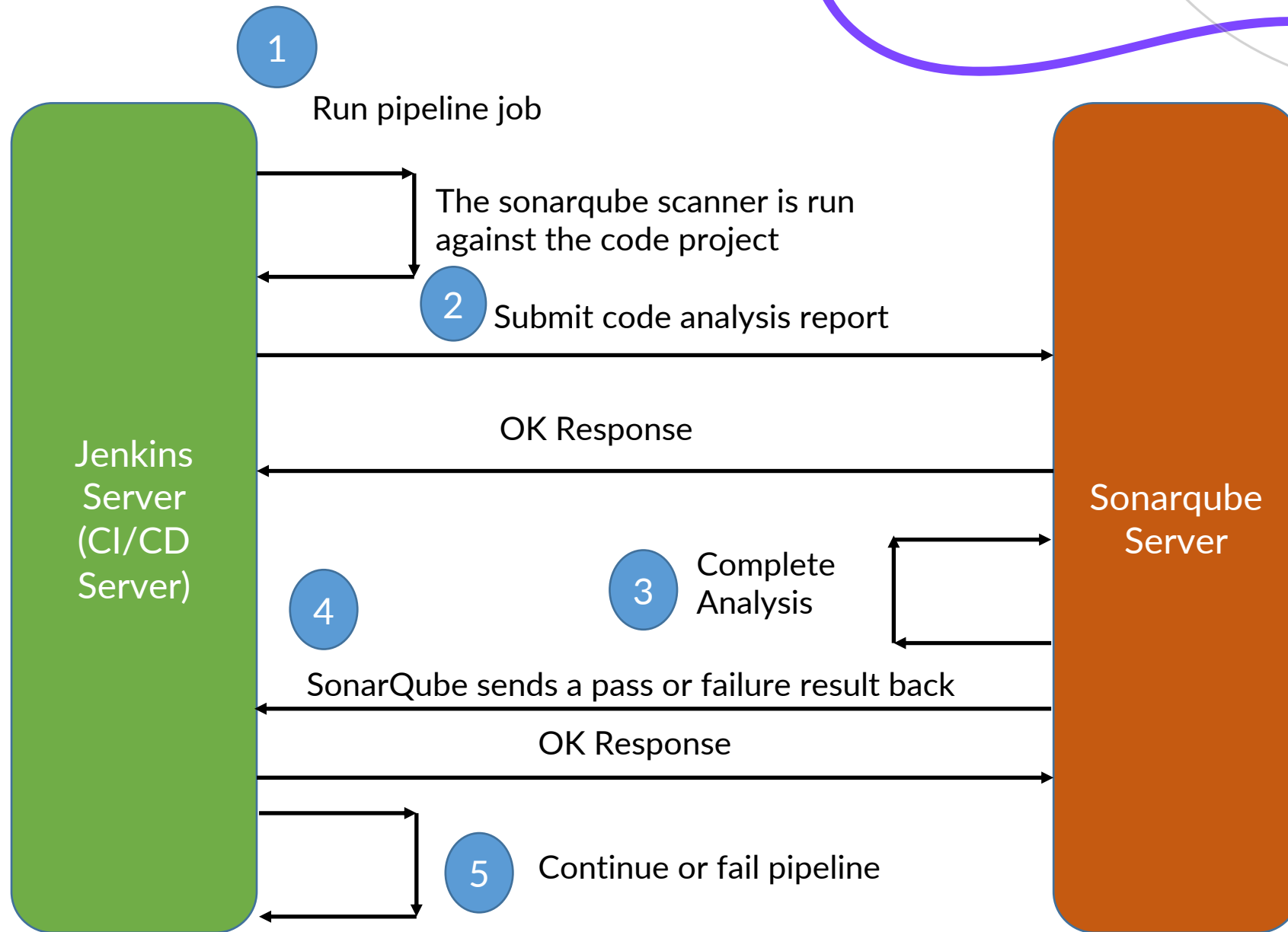
- **Security remediation effort** (security_remediation_effort)
Effort to fix all vulnerability issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.
- **Security remediation effort on new code** (new_security_remediation_effort)
Same as *Security remediation effort* but on the code changed on New Code.
- **Security Hotspots** (security_hotspots) Number of Security Hotspots
- **Security Hotspots on new code** (new_security_hotspots) Number of new Security Hotspots on New Code.
- **Security Review Rating** (security_review_rating)

Security

- **Security Review Rating on new code** (new_security_review_rating)
Security Review Rating for New Code.
- **Security Hotspots Reviewed** (security_hotspots_reviewed)
Percentage of Reviewed Security Hotspots.
Ratio Formula: $\text{Number of Reviewed Hotspots} \times 100 / (\text{To_Review Hotspots} + \text{Reviewed Hotspots})$
- **New Security Hotspots Reviewed**
Percentage of Reviewed Security Hotspots on New Cod



Integration with CI/CD tool





Rules

Rules

SonarQube evaluates your source code against its set of rules to generate issues.

There are four types of rules:

1. Code Smell (Maintainability domain)
2. Bug (Reliability domain)
3. Vulnerability (Security domain)
4. Security Hotspot (Security domain)

Filters in Rules

Language: the language to which a rule applies.

Type: Bug, Vulnerability, Code Smell or Security Hotspot rules.

Tag: it is possible to add tags to rules in order to classify them and to help discover them more easily.

Repository: the engine/analyzer that contributes rules to SonarQube.

Default Severity: the original severity of the rule - as defined by SonarQube.

Filters in Rules

Status: rules can have 3 different statuses:

Beta: The rule has been recently implemented and we haven't gotten enough feedback from users yet, so there may be false positives or false negatives.

Deprecated: The rule should no longer be used because a similar, but more powerful and accurate rule exists.

Ready: The rule is ready to be used in production.

Available Since: date when a rule was first added on SonarQube. This is useful to list all the new rules since the last upgrade of a plugin for instance.

Template: display rule templates that allow to create custom rules (see later on this page).

Quality Profile: inclusion in or exclusion from a specific profile