SonarQube — Implementing Quality with SonarQube

Table of Contents

Table of Contents

Introduction to Software Quality

What is Software Quality?

Definition: Software quality is the degree to which a software product meets specified requirements and user expectations while being maintainable, reliable, and efficient.

Key Dimensions of Quality:

- **Functionality**: Does it do what it's supposed to do?
- **Reliability**: Does it work consistently without failures?
- **Performance**: Does it execute efficiently?
- **Maintainability**: Can it be easily modified and improved?
- **Security**: Is it protected against vulnerabilities?
- **Portability**: Can it run in different environments?

Introduction to Software Quality

Introduction to Software Quality

Why Quality Matters for Data Engineers:

- Data pipelines process critical business data
- Bugs can lead to incorrect analytics and bad business decisions
- Poorly written code increases technical debt
- Security vulnerabilities can expose sensitive data
- Maintenance costs escalate with poor quality code

The Case for Code Analysis

Why Analyze Code?

Business Arguments:

1. **Cost Reduction**: Finding bugs early is 10-100x cheaper than fixing them in production
2. **Risk Mitigation**: Prevent data breaches, compliance violations, and system failures
3. **Faster Delivery**: Clean code is easier to modify and deploy
4. **Team Efficiency**: Consistent code standards reduce onboarding time

The Case for Code Analysis

Technical Arguments:

1. **Early Bug Detection**: Catch issues before they reach production
2. **Security Vulnerability Identification**: Detect SQL injection, exposed credentials, etc.
3. **Code Consistency**: Enforce team coding standards automatically
4. **Technical Debt Management**: Quantify and track code health over time
5. **Knowledge Transfer**: Code reviews become more efficient

For Data Engineering Specifically:

- **Data Integrity**: Ensure transformation logic is correct
- **Pipeline Reliability**: Detect potential failures in ETL processes
- **Resource Optimization**: Identify inefficient queries and operations
- **Compliance**: Meet data governance and privacy standards (GDPR, CCPA)

Metrics - Measuring Code Quality

What is a Metric?

Definition: A metric is a quantifiable measure used to track and assess the status of a specific quality characteristic.

Internal vs External Metrics

Internal Metrics (White-box measures)

- Measured from the code itself
- Available during development
- Examples:
  - **Lines of Code (LOC)**: Size of codebase
  - **Cyclomatic Complexity**: Number of independent paths through code
  - **Code Coverage**: Percentage of code executed by tests
  - **Code Duplication**: Repeated code blocks
  - **Maintainability Index**: Composite measure of code complexity

Metrics - Measuring Code Quality

External Metrics (Black-box measures)

- Measured from system behavior
- Require execution/deployment
- Examples:
  - **Defect Density**: Bugs per 1000 lines of code
  - **Mean Time Between Failures (MTBF)**
  - **Response Time**: System performance
  - **User Satisfaction**: End-user feedback

Metrics - Measuring Code Quality

Key Metrics in SonarQube

Reliability Metrics:

- **Bugs**: Code errors that will likely lead to failures
- **Reliability Rating**: A-E scale based on bug severity

Security Metrics:

- **Vulnerabilities**: Security weaknesses
- **Security Hotspots**: Code requiring manual security review
- **Security Rating**: A-E scale

Maintainability Metrics:

- **Code Smells**: Maintainability issues
- **Technical Debt**: Estimated time to fix all code smells
- **Maintainability Rating**: A-E scale

Coverage Metrics:

- **Unit Test Coverage**: % of code covered by tests
- **Line Coverage**: % of lines executed by tests
- **Condition Coverage**: % of boolean conditions tested

Duplication:

- **Duplicated Lines**: % of duplicated code
- **Duplicated Blocks**: Number of repeated code sections

Python-Specific Metrics:

- PEP 8 compliance violations
- Cognitive complexity
- Import organization issues

Software Quality Models & Standards

Major Quality Models

1. ISO/IEC 25010 (SQuaRE)

- International standard for software quality
- 8 Quality Characteristics:
  - Functional Suitability
  - Performance Efficiency
  - Compatibility
  - Usability
  - Reliability
  - Security
  - Maintainability
  - Portability

2. CISQ (Consortium for IT Software Quality)

- Four key measures:
  - **Reliability**: Avoid failures
  - **Security**: Resist attacks
  - **Performance Efficiency**: Optimize resources
  - **Maintainability**: Facilitate changes

3. Technical Debt Metaphor (Ward Cunningham)

- Treats quality shortcuts as "debt"
- Must be "repaid" with interest (increased maintenance cost)
- SonarQube quantifies this in time/money

Software Quality Models & Standards

Standardization Efforts

Code Standards:

- **Python**: PEP 8, PEP 257 (docstrings)
- **Security**: OWASP Top 10, CWE/SANS Top 25
- **Cloud**: Google Cloud Python Style Guide

Data Engineering Standards:

- **Data Quality**: DAMA-DMBOK framework
- **Data Governance**: ISO/IEC 38505
- **Cloud Security**: CIS Google Cloud Platform Benchmarks

Implementation - Quality in Continuous Integration

Quality Gates

Definition: A set of conditions that code must meet before moving to the next stage

Typical Quality Gate Conditions:

- No blocker or critical bugs
- Security vulnerability rating ≥ A
- Code coverage ≥ 80%
- Technical debt ratio ≤ 5%
- Duplicated lines ≤ 3%

Metrics - Measuring Code Quality

📊 Core Quality Metrics

🐛 Reliability Rating

Formula:

Bug Density = Bugs / (LOC / 1000)

If bugs = 0 → A
If density < 0.1% → B
If density < 1% → C
If density < 5% → D
If density ≥ 5% → E

🔒 Security Rating

Formula:

Vuln. Density = Vulnerabilities / (LOC / 1000)

If vulnerabilities = 0 → A
If density < 0.1% → B
If density < 1% → C

🔧 Maintainability Rating

Formula:

Tech Debt Ratio = (Tech Debt / Dev Cost) × 100

If ratio ≤ 5% → A (Excellent)
If ratio ≤ 10% → B (Good)
If ratio ≤ 20% → C (Fair)

📈 Rating Scale (A to E)

A	B	C	D
Excellent	Good	Fair	Poor
No issues or minimal	Minor issues	Moderate issues	Significant issues

🚦 Quality Gate Conditions

Default "Sonar way" Quality Gate

Conditions on New Code:

Conditions on Overall Code:

💧 Leak Period (New Code Focus)

What is it? Focus on new and changed code since baseline

✓ Prevents "grandfathering" old issues
✓ Encourages continuous improvement
✓ More manageable for legacy projects

Example: Project with 10,000 old issues → Focus quality gate on new code only

🎯 Recommended Thresholds for Data Engineers

ETL Pipelines:

- Coverage ≥ 75% (unit tests can be challenging)
- Complexity < 20 (ETL logic can be complex)
- Zero security vulnerabilities (data access)

Production APIs:

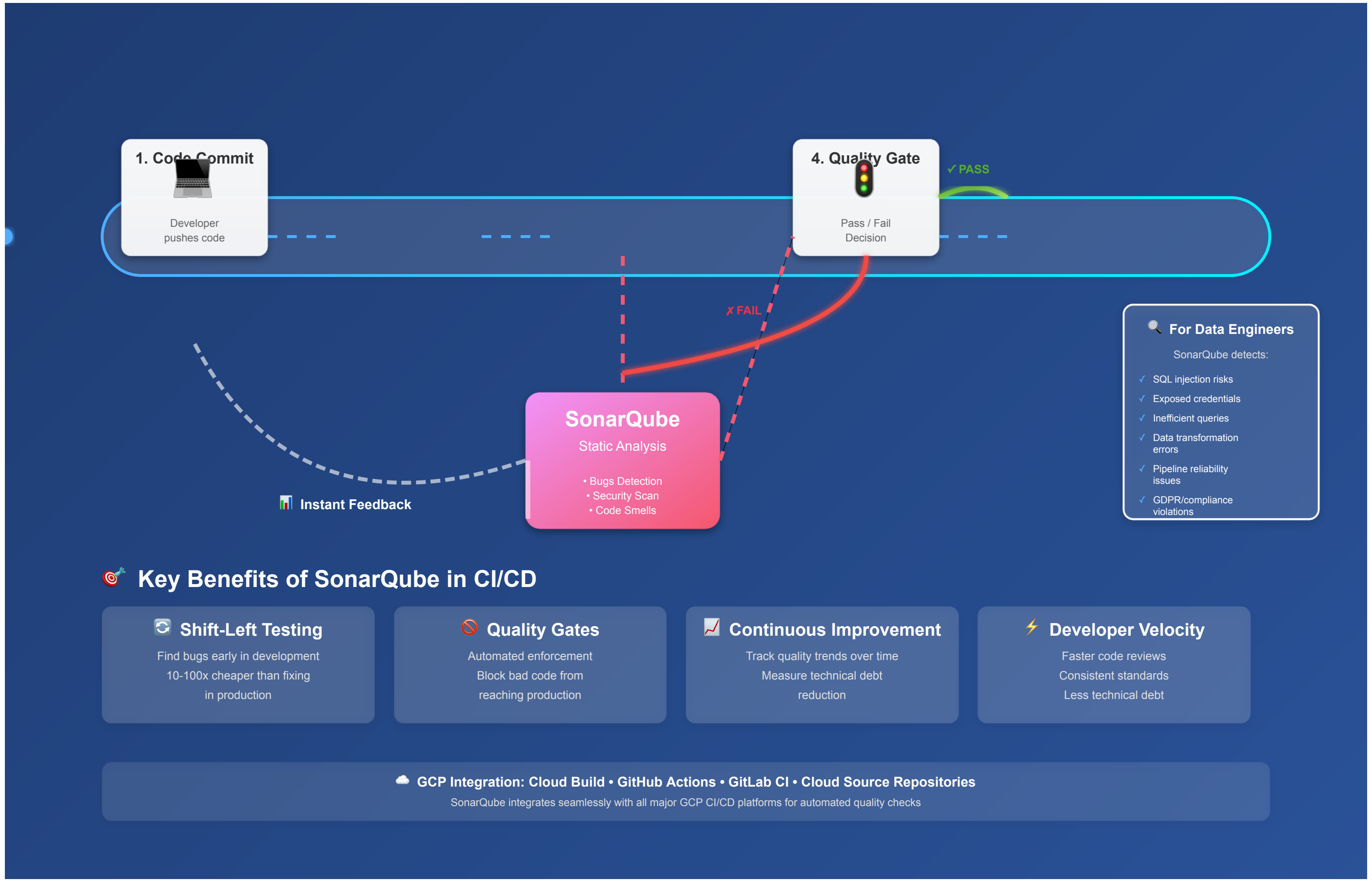
- Coverage ≥ 85% (mission-critical)
- All ratings = A
- Zero blocker/critical issues

Data Analysis Scripts:

- Coverage ≥ 50% (exploratory nature)
- Focus on security (data access, credentials)

Implementation - Quality in Continuous Integration

Quality Gates



Implementation - Quality in Continuous Integration

Continuous Integration & Quality

The CI/CD Quality Loop:

1. Developer commits code
2. CI triggers automated build
3. Unit tests execute
4. SonarQube analyzes code
5. Quality gate evaluates results
6. Pipeline succeeds/fails based on gate
7. Feedback to developer
8. Fix → Commit → Repeat

Benefits:

- Immediate feedback (shift-left approach)
- Prevents quality degradation
- Automates code review
- Creates quality culture

SonarQube in CI/CD Pipeline

SonarQube in GCP CI/CD

Integration Options:

1. Cloud Build + SonarQube

```yaml
# cloudbuild.yaml
steps:
  # Run tests with coverage
  - name: 'python:3.9'
    entrypoint: 'bash'
    args:
      - '-c'
      - |
        pip install pytest pytest-cov
        pytest --cov=. --cov-report=xml

  # SonarQube analysis
  - name: 'sonarsource/sonar-scanner-cli:latest'
    env:
      - 'SONAR_HOST_URL=${_SONAR_URL}'
      - 'SONAR_LOGIN=${_SONAR_TOKEN}'
    args:
      - '-Dsonar.projectKey=my-data-pipeline'
      - '-Dsonar.python.coverage.reportPaths=coverage.xml'
```

SonarQube in CI/CD Pipeline

How SonarQube Works (Step-by-Step)

Phase 1: Source Code Analysis

1. **SonarScanner** reads your Python files
2. Generates **Abstract Syntax Tree (AST)**
3. Applies **rule engines** (Python analyzer)
4. Detects issues based on predefined rules

Phase 2: Data Collection

1. Gathers metrics (LOC, complexity, duplication)
2. Processes test coverage reports
3. Compiles all findings

Phase 3: Server-Side Processing

1. Data sent to **SonarQube Server**
2. Server computes quality metrics
3. Evaluates against **Quality Gates**
4. Stores results in database
5. Calculates trends over time

Phase 4: Reporting

1. Results displayed in web UI
2. Quality Gate status returned to CI/CD
3. Notifications sent (if configured)

Classification & Overview of Analysis Tools

Categories of Analysis Tools

1. Static Application Security Testing (SAST)

- Analyzes source code without execution
- **Examples**: SonarQube, Bandit (Python), Checkmarx, Veracode
- **Use Case**: Find security vulnerabilities early

2. Dynamic Application Security Testing (DAST)

- Tests running applications
- **Examples**: OWASP ZAP, Burp Suite
- **Use Case**: Detect runtime vulnerabilities

3. Software Composition Analysis (SCA)

- Analyzes third-party dependencies
- **Examples**: Snyk, WhiteSource, OWASP Dependency-Check
- **Use Case**: Identify vulnerable libraries

4. Linters & Formatters

- Enforce code style and simple rules
- **Examples**:
  - **Python**: pylint, flake8, black, mypy
  - **General**: ESLint (JS), RuboCop (Ruby)
- **Use Case**: Maintain code consistency

5. Test Coverage Tools

- Measure test completeness
- **Examples**: Coverage.py, pytest-cov, JaCoCo
- **Use Case**: Ensure adequate testing

6. Code Complexity Analyzers

- Measure code complexity
- **Examples**: Radon, McCabe
- **Use Case**: Identify hard-to-maintain code

Classification & Overview of Analysis Tools

Tool Comparison Matrix

Tool	Type	Python Support	GCP Integration	Focus
SonarQube	SAST + Quality	✅ Excellent	✅ Yes	Comprehensive quality
Bandit	SAST	✅ Python-only	✅ Easy	Security only
Pylint	Linter	✅ Python-only	✅ Easy	Style + basic issues
Snyk	SCA	✅ Yes	✅ Yes	Dependency security
Black	Formatter	✅ Python-only	✅ Easy	Code formatting
MyPy	Type Checker	✅ Python-only	✅ Easy	Type safety

Classification & Overview of Analysis Tools

Why SonarQube for Data Engineers?

Advantages:

1. **Comprehensive**: Combines multiple analysis types
2. **Historical Tracking**: See quality trends over time
3. **Quality Gates**: Enforces standards automatically
4. **Language Support**: Python + SQL + others
5. **GCP Compatible**: Works with Cloud Build, GitHub, GitLab
6. **Technical Debt**: Quantifies maintenance burden
7. **Enterprise Ready**: LDAP, SSO, role-based access

SonarQube Product Offering

SonarQube vs SonarLint vs SonarCloud

Product Comparison:

Feature	SonarLint	SonarQube	SonarCloud
Type	IDE Plugin	Self-hosted Server	Cloud SaaS
Usage	Individual Developer	Team/Enterprise	Team/Cloud-native
Cost	Free	Community (Free) + Commercial	Pay-per-use
Analysis	Real-time (local)	On-demand (server)	On-demand (cloud)

SonarQube Product Offering

SonarQube vs SonarLint vs SonarCloud

Feature	SonarLint	SonarQube	SonarCloud
Languages	25+	30+	30+
Quality Gates	❌ No	✅ Yes	✅ Yes
Historical Data	❌ No	✅ Yes	✅ Yes
CI/CD Integration	❌ No	✅ Yes	✅ Yes
Team Collaboration	❌ No	✅ Yes	✅ Yes

SonarQube Product Offering

SonarLint - IDE Integration

What is SonarLint?

- Free IDE plugin for instant feedback
- Works like a spell-checker for code
- Analyzes code as you type
- No server required

Supported IDEs:

- **VS Code** (most popular for Python)
- PyCharm
- IntelliJ IDEA
- Eclipse
- Visual Studio

Key Features:

```
Developer writes code
        ↓
SonarLint analyzes in real-time
        ↓
Issues highlighted immediately
        ↓
Quick fixes suggested
        ↓
Developer fixes before commit
```

Connected Mode:

- Links SonarLint to SonarQube server
- Syncs quality profiles and rules
- Shows project-specific issues
- Consistent standards across team

SonarQube Product Offering

SonarLint - IDE Integration

Example: SonarLint in VS Code

```python
# Bad code - SonarLint will flag this immediately
password = "hardcoded123"  # 🔴 Critical: Remove this hardcoded password

# SQL Injection risk
query = f"SELECT * FROM users WHERE id = {user_id}"  # 🔴 Security hotspot

# Unused import
import pandas as pd  # 🟡 Remove unused import

# Cognitive complexity
def complex_function(x, y, z, a, b, c):  # 🟡 Reduce cognitive complexity
    if x > 0:
        if y > 0:
            if z > 0:
                if a > 0:
                    if b > 0:
                        # Too many nested conditions
                        pass
```

SonarQube Product Offering

SonarQube Editions

1. Community Edition (FREE)

- 19 languages including Python
- Basic security analysis
- Quality gates
- CI/CD integration
- No commercial support

2. Developer Edition

- Branch analysis
- Pull request decoration
- More languages (C, C++, Objective-C)
- Enhanced security (taint analysis)
- 12+ additional rules for Python

3. Enterprise Edition

- Portfolio management
- Executive reporting
- Multi-instance deployment
- Advanced security (OWASP, SANS compliance)

4. Data Center Edition

- High availability
- Horizontal scaling
- Load balancing
- For large enterprises

For Data Engineers on GCP:

- **Start with**: Community Edition (free)
- **Consider**: Developer Edition for branch analysis
- **Deploy on**: GCP Compute Engine or GKE

SonarQube vs Other Tools & Plugins

Competitive Landscape

SAST Tools Comparison:

Tool	Strengths	Weaknesses	Best For
SonarQube	Comprehensive, multi-language, quality gates	Setup complexity	General purpose
Bandit	Python-focused, fast, simple	Security only, no quality metrics	Python security
Pylint	Deep Python analysis	Noisy, no server	Python linting
Semgrep	Custom rules, fast	Limited languages	Custom security rules
CodeClimate	Easy setup, cloud-native	Limited depth	Small teams

SonarQube vs Other Tools & Plugins

Why Choose SonarQube for Data Engineering?

Advantages:

1. **Multi-language support**: Python + SQL + YAML
2. **Historical tracking**: See quality trends
3. **Quality gates**: Automated enforcement
4. **Technical debt**: Quantifiable metrics
5. **Open source**: Free community edition
6. **Extensible**: Plugin ecosystem
7. **GCP compatible**: Easy integration

When to Use Alternatives:

- **Bandit**: Quick Python security scans in pre-commit hooks
- **Pylint**: Additional Python-specific checks
- **Snyk**: Dependency vulnerability scanning
- **Combine**: SonarQube + Bandit + Snyk for comprehensive coverage

SonarQube vs Other Tools & Plugins

SonarQube Plugin Ecosystem

Core Plugins (Pre-installed):

- Python Analyzer
- XML Analyzer
- YAML Analyzer
- Text Analyzer
- HTML Analyzer

Popular Community Plugins:

1. Language Plugins:

- **Sonar Python**: Enhanced Python rules (additional rules beyond core)
- **Sonar SQL**: SQL code analysis

2. Integration Plugins:

- **GitHub Plugin**: PR decoration, authentication
- **GitLab Plugin**: Merge request integration
- **LDAP Plugin**: Enterprise authentication
- **SAML Plugin**: SSO integration

3. Reporting Plugins:

- **PDF Report Plugin**: Generate PDF reports
- **Word Report Plugin**: Export to DOCX
- **Custom Metrics Plugin**: Define custom metrics

4. Quality Plugins:

- **SonarQube Quality Gates Plugin**: Additional gate conditions
- **Technical Debt Plugin**: Enhanced debt tracking

SonarQube vs Other Tools & Plugins

Installing Plugins:

Method 1: Marketplace (UI)

```
1. Login as admin
2. Administration → Marketplace
3. Search for plugin
4. Click "Install"
5. Restart SonarQube
```

Method 2: Manual Installation

```
# Download plugin JAR
wget https://github.com/plugin/releases/plugin.jar

# Copy to extensions directory
cp plugin.jar $SONARQUBE_HOME/extensions/plugins/

# Restart SonarQube
./sonar.sh restart
```

SonarQube Architecture

Component Details

1. Web Server

- **Port**: 9000 (default)
- **Role**: User interface and API
- **Technology**: Java-based web application
- **Features**:
  - Project dashboards
  - Issue tracking
  - Quality profile management
  - User/permission management
  - Quality gate configuration

2. Compute Engine

- **Role**: Background processing
- **Functions**:
  - Process analysis reports from scanners
  - Calculate quality metrics
  - Evaluate quality gates
  - Update database
  - Send webhooks
- **Scalability**: Can be scaled horizontally in Data Center edition

SonarQube Architecture

Component Details

3. Database

- **Supported**:
  - **PostgreSQL** (recommended)
  - Oracle
  - Microsoft SQL Server
- **Stores**:
  - Project configuration
  - Quality profiles and gates
  - Issues and vulnerabilities
  - Metrics history
  - User accounts
- **Size**: Grows with projects and history

4. Elasticsearch

- **Role**: Search indexing
- **Uses**:
  - Fast issue search
  - Code search
  - Text search across projects
- **Embedded**: Comes with SonarQube (can be external for scale)

5. SonarScanner

- **Role**: Client-side analyzer
- **Types**:
  - SonarScanner CLI (generic), for Maven, Gradle...

SonarQube Architecture

Analysis Flow

```
Step 1: Developer triggers analysis
         ↓
Step 2: SonarScanner reads source code
         ↓
Step 3: Scanner applies language analyzers
         • Python analyzer
         • SQL analyzer
         • YAML analyzer
         ↓
Step 4: Scanner generates analysis report
         • Issues found
         • Metrics calculated
         • Test coverage data
         ↓
Step 5: Report sent to SonarQube Server
         ↓
Step 6: Compute Engine processes report
         ↓
Step 7: Data stored in Database
         ↓
Step 8: Quality Gate evaluated
         ↓
Step 9: Results visible in Web UI
         ↓
Step 10: Webhook notification (optional)
```

Deployment on GCP

Option 1: Compute Engine VM

```
# Create VM
gcloud compute instances create sonarqube \
    --machine-type=n1-standard-4 \
    --image-family=ubuntu-2004-lts \
    --image-project=ubuntu-os-cloud \
    --boot-disk-size=50GB \
    --tags=sonarqube

# Install SonarQube
ssh into VM
sudo apt update
sudo apt install -y openjdk-11-jdk postgresql
# ... install SonarQube
```

Deployment on GCP

Option 2: Google Kubernetes Engine (GKE)

Option 3: Cloud Run (Not Recommended)

- SonarQube is stateful
- Requires persistent storage
- Better suited for VMs or Kubernetes

Core SonarQube Concepts

1. SonarScanner

What is SonarScanner?

- Client-side tool that analyzes code
- Sends analysis report to SonarQube server
- Multiple scanner types available

Core SonarQube Concepts

Configuration File: `sonar-project.properties`

```
# Required properties
sonar.projectKey=data-pipeline-project
sonar.projectName=Data Pipeline
sonar.projectVersion=1.0

# Source code location
sonar.sources=src
sonar.tests=tests

# Python-specific settings
sonar.python.version=3.9
sonar.python.coverage.reportPaths=coverage.xml
sonar.python.xunit.reportPath=test-results.xml

# Exclusions
sonar.exclusions=**/migrations/**,**/tests/**,**/__pycache__/**
sonar.test.exclusions=**/tests/**

# Encoding
sonar.sourceEncoding=UTF-8

# Additional parameters
sonar.language=py
```

Core SonarQube Concepts

Running SonarScanner:

```
# Basic scan
sonar-scanner

# Scan with custom properties
sonar-scanner \
  -Dsonar.projectKey=my-project \
  -Dsonar.sources=src \
  -Dsonar.host.url=http://sonarqube-server:9000 \
  -Dsonar.login=<token>
# Scan specific branch
sonar-scanner \
  -Dsonar.branch.name=feature/new-pipeline
# Debug mode
sonar-scanner -X
```

Authentication:

```
# Generate token in SonarQube UI
# User → My Account → Security → Generate Token
# Use token in scanner
sonar-scanner -Dsonar.login=<your-token>
# Or set environment variable
export SONAR_TOKEN=<your-token>
sonar-scanner
```

Core SonarQube Concepts

2. Rules

What are Rules?

- Coding standards and best practices
- Each rule checks for a specific issue
- Categorized by type, severity, and language

Rule Types:

Type	Icon	Description	Example
Bug	🐛	Code error that will cause failure	Null pointer dereference
Vulnerability	🔒	Security weakness	SQL injection risk
Code Smell	👃	Maintainability issue	Complex function
Security Hotspot	🔥	Code requiring security review	Password validation

Core SonarQube Concepts

Rule Severity:

```
Blocker    🔴  Must fix immediately (blocks deployment)
Critical   🟠  Should fix ASAP
Major      🟡  Should fix
Minor      🟢  Nice to fix
Info       ℹ️   FYI
```

Core SonarQube Concepts

Python-Specific Rules Examples:

```python
# Rule: S1481 - Unused local variable
def process_data(df):
    temp = df.copy()  # 🟡 Variable 'temp' is never used
    return df

# Rule: S1192 - String literals should not be duplicated
def validate_user(user):
    if user.role == "admin":  # 🟡 String "admin" is duplicated
        log("admin access")
    elif user.role == "admin":  # Should use constant
        pass

# Rule: S3457 - Printf-style format should not be used
msg = "Hello %s" % name  # 🟡 Use f-strings or .format() instead
msg = f"Hello {name}"    # ✅ Correct

# Rule: S5852 - Regular expressions should not be vulnerable to Denial of Service
pattern = "(a+)+"  # 🔴 Vulnerable to ReDoS attack

# Rule: S1313 - Hardcoded IP addresses should not be used
server = "192.168.1.1"  # 🟡 IP should be in configuration

# Rule: S5245 - Using shell without validation
os.system(user_input)  # 🔴 Command injection vulnerability
```

Core SonarQube Concepts

Rule Metadata:

Each rule includes:

- **Description**: What it checks
- **Why**: Rationale (why it matters)
- **How to fix**: Remediation guidance
- **Examples**: Non-compliant and compliant code
- **Tags**: security, bug, cwe, owasp
- **Technical Debt**: Estimated fix time

Core SonarQube Concepts

3. Violations (Issues)

What is a Violation?

- Instance where code breaks a rule
- Also called "issue" in SonarQube
- Tracked throughout code lifecycle

Issue Attributes:

```
Issue ID: AVxyz123
Rule: S1234 - Avoid empty catch blocks
Type: Bug
Severity: Major
Status: Open
Creation Date: 2025-10-20
Location: src/pipeline.py:45
Effort: 5 min
Assignee: john.doe
Tags: error-handling, bad-practice
```

Core SonarQube Concepts

Issue Status:

```
Open ──────────► Confirmed ──────────► Resolved ──────────► Closed
  │                 │                     │                   │
  │                 │                     │                   │
Detected         Team agrees          Fixed/Won't Fix      Verified
```

Status Values:

- **Open**: New issue detected
- **Confirmed**: Team confirmed it's a real issue
- **Resolved**: Marked as fixed, false positive, or won't fix
  - **Fixed**: Code corrected
  - **False Positive**: Not actually a problem
  - **Won't Fix**: Accepted technical debt
- **Reopened**: Issue reappeared
- **Closed**: Verified as fixed after rescan

Core SonarQube Concepts

Issue Workflow Example:

```python
# Initial code - Issue detected (OPEN)
def load_data(file):
    try:
        return pd.read_csv(file)
    except:  # 🔴 Empty except block
        pass

# Developer confirms (CONFIRMED)
# Assigns to themselves

# Developer fixes (RESOLVED - Fixed)
def load_data(file):
    try:
        return pd.read_csv(file)
    except FileNotFoundError as e:  # ✅ Specific exception
        logger.error(f"File not found: {e}")
        raise
    except pd.errors.ParserError as e:
        logger.error(f"Parse error: {e}")
        raise

# Next scan verifies fix (CLOSED)
```

Core SonarQube Concepts

4. Quality Profiles

What is a Quality Profile?

- Collection of rules for a language
- Defines which rules are active
- Can customize rule severity
- Applied to projects

Default Profiles:

```
Sonar way (Python)
├── 465 rules activated
├── Focuses on: Bugs, Vulnerabilities, Code Smells
└── Recommended for most projects

Sonar way Recommended (Python)
├── Extended ruleset
├── More strict than "Sonar way"
└── For teams wanting higher quality bar
```

Core SonarQube Concepts

Creating Custom Profile:

```
Quality Profiles → Create → Python

Name: Data Engineering Standard
Parent: Sonar way
Language: Python

Activate additional rules:
✓ S3776: Cognitive Complexity (max: 10)
✓ S1541: Functions should not have too many lines (max: 50)
✓ S104: Files should not have too many lines (max: 300)
✓ All security rules
✓ All bug rules

Deactivate:
✗ S103: Lines should not be too long (we use Black)
```

Core SonarQube Concepts

Applying Profile to Project:

```
Project → Administration → Quality Profiles
Select: Data Engineering Standard
```

Profile Inheritance:

```
Built-in: Sonar way
    ↓ (extends)
Custom: Data Engineering Base
    ↓ (extends)
Project-specific: Pipeline Quality Profile
```

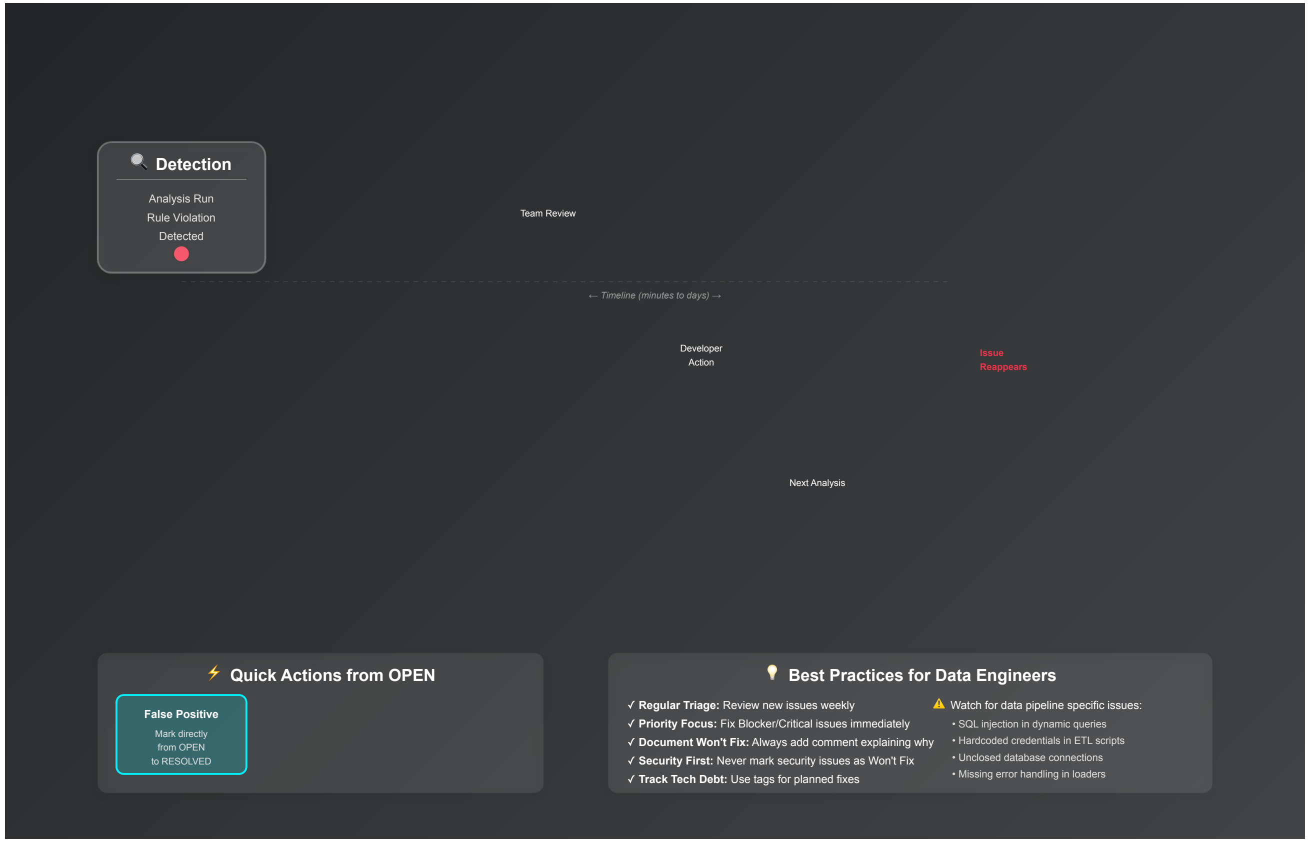
Comparing Profiles:

```
Quality Profiles → Compare
Profile 1: Sonar way
Profile 2: Data Engineering Standard

Differences:
+ 25 additional rules activated
- 3 rules deactivated
↕️ 10 rules with different severity
```

Issue Lifecycle Management

Issue Lifecycle Diagram



Issue Lifecycle Management

Managing Issues in SonarQube

Bulk Actions:

```
Project → Issues → Select multiple issues

Actions:
• Assign to developer
• Change severity
• Mark as false positive
• Add tags
• Add comments
• Set type
```

Issue Lifecycle Management

Issue Assignment Strategy:

Option 1: Automatic Assignment

- Assign to last commit author
- Requires SCM integration (Git)

Option 2: Manual Assignment

- Tech lead reviews and assigns
- Good for critical issues

Option 3: Team Ownership

- Each team owns specific modules
- Issues auto-assigned based on file location

Issue Lifecycle Management

Issue Tracking Best Practices

1. Regular Issue Review

```
Weekly meeting:
• Review all Blocker/Critical issues
• Triage new Major issues
• Close resolved issues
```

2. False Positive Management

```python
# Mark as false positive with comment
# SonarQube will remember this

# Example: Using eval() for safe math expression
def calculate(expression):
    # Only used with validated input from config
    return eval(expression)  # Marked as false positive
```

3. Won't Fix Documentation

```
Mark as Won't Fix + Add comment explaining:
"This pandas warning is acceptable in our use case
because we always work with clean data from our ETL."
```

4. Technical Debt Tracking

```
Create custom tag: "technical-debt-sprint-23"
• Track issues to fix in next sprint
• Monitor progress
```

Quality Metrics & Formulas

Core Metrics

1. Reliability Rating (A-E)

Formula:

```
If bugs = 0                    → A
If bug density < 0.1%          → B
If bug density < 1%            → C
If bug density < 5%            → D
If bug density ≥ 5%            → E

Bug Density = (Critical+Major Bugs) / (Lines of Code / 1000)
```

Quality Metrics & Formulas

Core Metrics

2. Security Rating (A-E)

Formula:

```
If vulnerabilities = 0                        → A
If vulnerability density < 0.1%               → B
If vulnerability density < 1%                 → C
If vulnerability density < 5%                 → D
If vulnerability density ≥ 5%                 → E

Vulnerability Density = Vulnerabilities / (LOC / 1000)
```

Quality Metrics & Formulas

Core Metrics

3. Maintainability Rating (A-E)

Based on Technical Debt Ratio:

Formula:

```
Technical Debt Ratio = (Technical Debt / Development Cost) × 100
If ratio ≤ 5%           → A
If ratio ≤ 10%          → B
If ratio ≤ 20%          → C
If ratio ≤ 50%          → D
If ratio > 50%          → E
Technical Debt = Sum of all remediation efforts (in minutes)
Development Cost = (Lines of Code × 0.06) days
```

Quality Metrics & Formulas

Core Metrics

Example Calculation:

```
Project: 10,000 lines of Python
Technical Debt: 2 days (960 minutes)
Development Cost: 10,000 × 0.06 = 600 minutes = 10 hours

Debt Ratio = (960 / 600) × 100 = 160%
Rating = D (Very poor maintainability)
```

Quality Metrics & Formulas

Core Metrics

4. Coverage

Formula:

```
Coverage % = (Covered Lines / Total Executable Lines) × 100

Line Coverage = Lines executed by tests / Total lines
Branch Coverage = Branches executed / Total branches
```

Quality Metrics & Formulas

Core Metrics

5. Duplications

Formula:

```
Duplication Density = (Duplicated Lines / Total Lines) × 100

Duplicated Blocks = Number of duplicated code blocks
Duplicated Files = Files containing duplication
```

6. Code Smells

Count of maintainability issues:

```
Total Code Smells = Blocker + Critical + Major + Minor + Info
```

Quality Metrics & Formulas

Core Metrics

7. Cognitive Complexity

Measures code understandability:

```python
def process_order(order):  # Complexity = 0
    if order.is_valid():   # +1 (if)
        if order.has_stock():  # +2 (nested if)
            if order.payment_ok():  # +3 (nested if)
                return True
    return False

# Cognitive Complexity = 6
```

Quality Metrics & Formulas

Metric Thresholds (Recommended)

```
Coverage:            ≥ 80%   (Excellent)
                     ≥ 60%   (Good)
                     < 50%   (Poor)

Duplication:         < 3%    (Excellent)
                     < 5%    (Good)
                     ≥ 10%   (Poor)

Complexity:          < 10    (Simple)
                     < 20    (Moderate)
                     ≥ 30    (Complex)

Technical Debt:      < 5%    (Excellent)
                     < 10%   (Good)
                     ≥ 20%   (Poor)
```

Leak Period & Quality Gates

Leak Period (New Code)

What is Leak Period?

- Focus on **new** and **changed** code
- Compare current analysis with baseline
- Prevents "grandfathering" old issues
- Encourages continuous improvement

Leak Period & Quality Gates

Leak Period (New Code)

Concept:

```
Old Code (Before)                 New Code (Leak Period)
      │                                     │
      │                                     ├─ New Lines Added
      │                                     ├─ Modified Lines
      │                                     └─ New Issues Introduced
      │
 Baseline Analysis                  Current Analysis
```

Leak Period & Quality Gates

Leak Period Settings:

Option 1: Previous Version

```
Compare with: Last released version
Use case: After each release
```

Option 2: Number of Days

```
Compare with: Code changed in last 30 days
Use case: Continuous development
```

Option 3: Specific Analysis

```
Compare with: Analysis from specific date
Use case: After major refactoring
```

Option 4: Reference Branch

```
Compare with: main/master branch
Use case: Feature branch development
```

Leak Period & Quality Gates

Configuring Leak Period:

```
Project → Administration → General Settings → New Code

Options:
○ Previous version
○ Number of days: [30]
○ Specific analysis
● Reference branch: [main]
```

Leak Period & Quality Gates

New Code Metrics:

All metrics available for new code:

- New Bugs
- New Vulnerabilities
- New Code Smells
- New Coverage %
- New Duplications %
- New Lines

Leak Period & Quality Gates

Why Leak Period Matters:

```
Scenario: Legacy project with 10,000 issues

Without Leak Period:
→ Overwhelming, team gives up
→ Quality gate always fails
→ No improvement

With Leak Period:
→ Focus only on new code
→ Quality gate checks new code only
→ Old code gradually improves
→ Team motivation maintained
```

Leak Period & Quality Gates

Quality Gates

What is a Quality Gate?

- Set of boolean conditions
- Checked after each analysis
- Determines if code quality is acceptable
- Can block deployment

Leak Period & Quality Gates

Quality Gate Architecture:

```
Analysis Complete
       ↓
Check Condition 1: New Bugs = 0                    ✓ Pass
       ↓
Check Condition 2: New Coverage ≥ 80%              ✓ Pass
       ↓
Check Condition 3: New Code Smells = 0             ✗ Fail
       ↓
Quality Gate Status: FAILED
       ↓
Pipeline blocked (if configured)
```

Leak Period & Quality Gates

Default Quality Gate: "Sonar way"

```
Conditions on New Code:
  - New Bugs: is greater than 0 → FAIL
  - New Vulnerabilities: is greater than 0 → FAIL
  - New Security Hotspots Reviewed: is less than 100% → FAIL
  - New Coverage: is less than 80% → FAIL
  - New Duplicated Lines: is greater than 3% → FAIL

Conditions on Overall Code:
  - Security Rating: is worse than A → FAIL
  - Reliability Rating: is worse than A → FAIL
```

Leak Period & Quality Gates

Creating Custom Quality Gate:

```
Quality Gates → Create
Name: Data Engineering Gate
Conditions on New Code:
✓ New Coverage ≥ 75%           (relaxed for data scripts)
✓ New Bugs = 0                 (strict)
✓ New Vulnerabilities = 0      (strict)
✓ New Code Smells ≤ 5          (allow minor issues)
✓ New Duplications < 3%
✓ New Blocker Issues = 0
✓ New Critical Issues = 0
Conditions on Overall Code:
✓ Security Rating ≥ B          (allow some inherited issues)
✓ Reliability Rating ≥ A       (zero bugs on overall)
✓ Security Hotspots Reviewed ≥ 80%
```

Leak Period & Quality Gates

Applying Quality Gate:

```
Project → Administration → Quality Gate
Select: Data Engineering Gate
```

Leak Period & Quality Gates

Quality Gate Strategies:

Strategy 1: Strict (High-Quality Team)

```
All conditions = 0
Coverage ≥ 90%
No exceptions
```

Strategy 2: Pragmatic (Most Teams)

```
Critical issues = 0
Coverage ≥ 80%
Allow minor code smells
```

Strategy 3: Progressive (Legacy Code)

```
Focus on new code only
Gradually improve thresholds
Monthly quality reviews
```

Leak Period & Quality Gates

Quality Gate Best Practices for Data Engineers

1. Start Lenient, Then Tighten

```
Month 1: Coverage ≥ 60%
Month 2: Coverage ≥ 70%
Month 3: Coverage ≥ 80%
```

Leak Period & Quality Gates

Quality Gate Best Practices for Data Engineers

2. Separate Gates for Different Project Types

```
ETL Pipelines Gate:
- Coverage ≥ 75% (unit tests can be challenging)
- Complexity < 20 (ETL logic can be complex)

Data Analysis Scripts Gate:
- Coverage ≥ 50% (exploratory nature)
- Focus on security (data access)

Production APIs Gate:
- Coverage ≥ 85% (mission-critical)
- All ratings = A
```

Leak Period & Quality Gates

Quality Gate Best Practices for Data Engineers

3. Monitor Quality Gate Trends

```
Dashboard → Quality Gate
Track:
- Pass rate over time
- Most failed conditions
- Projects consistently failing
```

Leak Period & Quality Gates

Quality Gate Best Practices for Data Engineers

4. Use Quality Gate Notifications

```
Project → Administration → Notifications

Email when:
✓ Quality gate status changes
✓ New vulnerabilities detected
✓ Coverage drops below threshold
```

Conducting a Quality Audit

What is a Quality Audit?

Definition: A quality audit is a systematic examination of a software project's code, processes, and adherence to quality standards to identify areas of improvement and ensure compliance with organizational goals.

Purpose:

- **Assessment**: Evaluate current state of code quality
- **Identification**: Discover technical debt and vulnerabilities
- **Prioritization**: Determine critical issues requiring immediate attention
- **Baseline**: Establish metrics for future improvement tracking
- **Compliance**: Verify adherence to coding standards and regulations

Conducting a Quality Audit

Types of Quality Audits

1. Initial Audit (Baseline)

- First-time analysis of existing project
- Establishes quality baseline
- Identifies technical debt
- Creates improvement roadmap

2. Periodic Audit

- Regular scheduled reviews (quarterly, bi-annual)
- Tracks quality trends over time
- Monitors technical debt evolution
- Validates improvement initiatives

3. Pre-Release Audit

- Before major releases
- Ensures deployment readiness
- Validates quality gate compliance
- Risk assessment for production

4. Compliance Audit

- Verify regulatory compliance (GDPR, HIPAA, SOX)
- Security standards validation (OWASP, CWE)
- Industry-specific requirements
- Documentation and traceability

Conducting a Quality Audit

Audit Preparation

1. Define Audit Scope

```
Questions to Answer:
• Which projects/modules to audit?
• What quality dimensions to focus on?
  - Security, reliability, maintainability, performance?
• What time period to cover?
• Which teams are involved?
```

2. Select Audit Criteria

```
Quality Standards:
  - ISO/IEC 25010 compliance
  - OWASP Top 10 security
  - Company coding standards
  - Language-specific guidelines (PEP 8 for Python)

Metrics to Measure:
  - Code coverage percentage
  - Technical debt ratio
  - Security vulnerability count
  - Code complexity metrics
  - Duplication percentage
```

Conducting a Quality Audit

Audit Execution Steps

Step 1: Automated Analysis

```
# Configure SonarQube project
sonar-scanner \
  -Dsonar.projectKey=audit-project \
  -Dsonar.sources=src \
  -Dsonar.tests=tests \
  -Dsonar.python.coverage.reportPaths=coverage.xml \
  -Dsonar.host.url=http://sonarqube:9000 \
  -Dsonar.login=<audit-token>

# Run additional security scans
bandit -r src/ -f json -o bandit-report.json

# Dependency vulnerability scan
pip-audit --format json --output dependencies-audit.json

# Check code formatting
black --check src/
flake8 src/ --statistics
```

Conducting a Quality Audit

Step 2: Data Collection

Gather metrics from SonarQube:

Metric Category	Key Indicators
Reliability	Total bugs, bug density, reliability rating
Security	Vulnerabilities, security hotspots, security rating
Maintainability	Code smells, technical debt, maintainability rating
Coverage	Unit test coverage %, line coverage, branch coverage
Duplications	Duplicated lines %, duplicated blocks
Complexity	Cyclomatic complexity, cognitive complexity
Size	Lines of code, files, classes, functions

Conducting a Quality Audit

Step 3: Manual Code Review

Automated tools don't catch everything:

```
# Example: Architecture issues that tools miss

# Anti-pattern: God class (too many responsibilities)
class DataProcessor:
    def fetch_data(self): pass
    def clean_data(self): pass
    def validate_data(self): pass
    def transform_data(self): pass
    def store_data(self): pass
    def send_notifications(self): pass
    def generate_reports(self): pass
    def audit_logging(self): pass
    # ⚠️ SonarQube won't flag this as architectural issue
# Business logic issues
def calculate_discount(price, user_type):
    # Missing business rule validation
    # No documentation on discount logic
    # Hard to test edge cases
    return price * 0.9 if user_type == "premium" else price
```

Manual Review Checklist:

- Architecture patterns adherence
- Design principles (SOLID, DRY, KISS)

- Business logic correctness - Error handling strategies - Documentation quality

Conducting a Quality Audit

Step 4: Security Assessment

```
Security Audit Focus Areas:

1. Authentication & Authorization
   ✓ Password policies
   ✓ Token management
   ✓ Access control implementation

2. Data Protection
   ✓ Encryption at rest and in transit
   ✓ Sensitive data handling
   ✓ PII compliance (GDPR, CCPA)

3. Input Validation
   ✓ SQL injection prevention
   ✓ XSS protection
   ✓ Command injection risks

4. Dependency Security
   ✓ Vulnerable libraries
   ✓ Outdated packages
   ✓ License compliance
```

Conducting a Quality Audit

Step 5: Documentation Review

```
Documentation Audit:

Code Documentation:
□ Function/method docstrings present
□ Class documentation complete
□ Complex logic explained with comments
□ API documentation up-to-date

Project Documentation:
□ README comprehensive
□ Architecture documentation
□ Deployment guides
□ Contribution guidelines
□ Change logs maintained

Data Engineering Specific:
□ Data pipeline diagrams
□ Schema documentation
□ Data lineage tracking
□ ETL process documentation
```

Conducting a Quality Audit

Audit Report Structure:

```
# Quality Audit Report
## Executive Summary
- Overall quality rating: B
- Critical findings: 3
- Recommendations: 12
## Metrics Overview
| Metric | Current | Target | Status |
|--------|---------|--------|--------|
| Coverage | 65% | 80% | ⚠️ Below target |
| Bugs | 23 | <10 | ❌ Action needed |
| Security Rating | A | A | ✅ Compliant |
| Tech Debt Ratio | 12% | <10% | ⚠️ Moderate |
## Critical Findings
1. [CRITICAL] SQL injection vulnerability in user_queries.py:145
2. [CRITICAL] Hardcoded credentials in config.py:23
3. [CRITICAL] Missing input validation in data_ingestion.py:67
## Detailed Analysis
[Category-by-category breakdown]
## Recommendations
[Prioritized action items]
## Improvement Roadmap
[Timeline and milestones]
```

Conducting a Quality Audit

Audit Follow-up Actions

1. Prioritization Matrix

```
          High Impact
              │
     Critical │ Quick Wins
     (Fix Now) │ (Schedule Soon)
──────────────┼──────────────
     Low Prio │ Nice to Have
     (Backlog) │ (Monitor)
              │
          Low Impact
```

2. Create Action Plan

- Assign owners to each finding
- Set deadlines for critical issues
- Schedule follow-up audits
- Track remediation progress

3. Continuous Monitoring

- Integrate SonarQube into CI/CD
- Set up quality gates
- Regular automated scans
- Monthly quality reviews

Links with Agile Methodologies

Quality in Agile: The Perfect Match

Why Quality Tools Fit Agile:

Agile Principles:

- Working software
- Continuous delivery
- Sustainable development
- Technical excellence
- Responding to change

SonarQube Alignment:

- Immediate feedback on code quality
- Automated quality checks in CI/CD
- Prevents technical debt accumulation
- Enables refactoring with confidence
- Supports iterative improvement

Result: Quality becomes part of "Definition of Done", not an afterthought

Links with Agile Methodologies

Integration with Agile Ceremonies

1. Sprint Planning

```
Quality Considerations:
• Review technical debt from last sprint
• Allocate time for quality improvements (10-20% capacity)
• Define quality acceptance criteria for user stories
• Plan refactoring tasks based on SonarQube reports

Example User Story:
"As a user, I want to upload CSV files"

Acceptance Criteria:
✓ Functional requirements met
✓ Unit test coverage ≥ 80%
✓ No new critical/blocker issues
✓ SonarQube quality gate passes
✓ Security hotspots reviewed
```

Links with Agile Methodologies

2. Daily Standups

```
Quality-Focused Questions:

Instead of just:
• What did you do yesterday?
• What will you do today?
• Any blockers?

Add:
• Did the quality gate pass?
• Any new critical issues discovered?
• Need help resolving quality issues?

Example:
"Yesterday I completed the authentication feature.
The quality gate failed due to 65% coverage (need 80%).
Today I'll add missing tests and refactor complex methods
flagged by SonarQube."
```

Links with Agile Methodologies

3. Sprint Review/Demo

```
Quality Metrics in Demo:

Show stakeholders:
┌──────────────────────────────────────────┐
│ Sprint 23 Quality Dashboard              │
├──────────────────────────────────────────┤
│ Features Delivered: 5                    │
│ Quality Gate Status: ✅ PASSED           │
│ New Code Coverage: 82%                   │
│ Technical Debt Added: 2 hours            │
│ Technical Debt Removed: 8 hours          │
│ Net Quality Improvement: +6 hours        │
│ Security Rating: A                       │
│ Bugs Introduced: 0                       │
└──────────────────────────────────────────┘

Demonstrates:
• Not just features, but quality
• Sustainable velocity
• Professional engineering practices
```

Links with Agile Methodologies

4. Sprint Retrospective

```
Quality-Focused Retrospective Topics:

What went well?
✓ "All quality gates passed this sprint"
✓ "Coverage increased from 60% to 75%"
✓ "Zero production bugs from last sprint"

What didn't go well?
✗ "Spent 3 days fixing security vulnerabilities"
✗ "Quality gate blocked deployment twice"
✗ "Code review caught issues SonarQube missed"

Action items:
→ Add custom quality profile for our data pipelines
→ Enable SonarLint for immediate IDE feedback
→ Schedule monthly technical debt reduction sprints
→ Train team on secure coding practices
```

Links with Agile Methodologies

Agile-Specific Quality Practices

1. Definition of Done (DoD)

```
Feature is Done when:
  Code Complete:
    ✓ Functionality implemented
    ✓ Code reviewed and approved
  Quality Checks:
    ✓ Unit tests written and passing
    ✓ Integration tests passing
    ✓ Code coverage ≥ 80% for new code
    ✓ SonarQube quality gate passed
    ✓ No blocker/critical issues
    ✓ Security hotspots reviewed
    ✓ No new technical debt > 1 day
  Documentation:
    ✓ Code comments added
    ✓ API documentation updated
    ✓ User documentation updated
  Deployment:
    ✓ Deployed to staging
    ✓ Acceptance criteria validated
    ✓ Product owner approved
```

Links with Agile Methodologies

2. Technical Debt Management

The Boy Scout Rule: "Leave the code better than you found it"

```
Practical Implementation:

When fixing a bug:
→ Fix the bug
→ Add test to prevent regression
→ Refactor surrounding code
→ Improve related code quality

When adding a feature:
→ Implement feature
→ Clean up code in same file
→ Address nearby code smells
→ Reduce local technical debt

Track in SonarQube:
→ Monitor debt ratio trend
→ Celebrate debt reduction
→ Make quality improvements visible
```

Links with Agile Methodologies

3. Continuous Improvement Cycles

```
Agile-Quality Feedback Loop:

Week 1: Sprint starts
    ↓
Daily: Developers commit code
    ↓
    → SonarQube analyzes each commit
    → Quality gate checks in CI/CD
    → Immediate feedback to developer
    ↓
Week 2: Mid-sprint quality check
    ↓
    → Review quality trends
    → Address emerging issues
    → Adjust sprint if needed
    ↓
Week 2: Sprint ends
    ↓
    → Retrospective includes quality metrics
    → Identify quality process improvements
    ↓
    → Apply learnings to next sprint

Result: Quality improves every sprint
```

Links with Agile Methodologies

Agile Anti-Patterns to Avoid

❌ Anti-Pattern 1: "We'll fix quality later"

```
Problem:
• Technical debt accumulates
• Quality never improves
• Velocity decreases over time

Solution:
• Quality is part of DoD
• Fix issues in same sprint
• Allocate time for quality
```

❌ Anti-Pattern 2: "Quality slows us down"

```
Problem:
• Skipping tests to go faster
• Ignoring code reviews
• Bypassing quality gates

Solution:
• Quality increases velocity long-term
• Bugs slow down more than tests
• Measure: time to fix bugs vs write tests
```

❌ Anti-Pattern 3: "100% coverage is the goal"

```
Problem:
• Focus on metrics, not quality
• Low-value tests
• False sense of security

Solution:
• Focus on meaningful tests
• Coverage is one metric among many
• Quality over quantity
```

❌ Anti-Pattern 4: "Quality is QA's job"

```
Problem:
• Developers don't own quality
• Late feedback
• Blame culture

Solution:
• Shift-left: quality is everyone's job
• Developers run quality checks
• Automated quality in CI/CD
```

SonarQube Security

Core Practices

- Enforce SSO/SAML or delegated LDAP for authentication
- Map least-privilege groups (`sonar-users`, `sonar-developers`, `sonar-admins`)
- Rotate project tokens every 90 days; use `sonar-scanner` tokens per CI job
- Enable HTTPS termination (reverse proxy or ingress) with TLS 1.2+
- Restrict public projects; review project visibility quarterly

Data Protection

- Activate encryption for settings via `encryptionKey` in `sonar.properties`
- Store database credentials in Vault/Secret Manager
- Backup database before every version upgrade
- Review security hotspots dashboard weekly
- Track audit events under Administration → Security → Audit Log

Log Management & Archiving

Primary Logs (`$SONARQUBE_HOME/logs/`):

- `sonar.log`: platform lifecycle, license, cluster info
- `web.log`: HTTP layer, authentication, UI errors
- `ce.log`: Compute Engine tasks, background processing
- `es.log`: Elasticsearch health and GC activity

Retention & Rotation

```
# sonar.properties
sonar.log.rollingPolicy = time:yyyy-MM-dd
sonar.log.maxFiles = 30
sonar.log.level = INFO
```

Archiving Workflow

1. Forward logs to centralized collector (Elastic, Splunk, Cloud Logging)
2. Tag entries with environment (`env=prod`, `env=nonprod`)
3. Set 30-day hot storage, 180-day cold archive
4. Automate alerts on `ERROR` and `FATAL` patterns

Notifications Administration

Channels

- Email (SMTP), Microsoft Teams, Slack (via webhook), custom webhooks
- Configure SMTP under Administration → General Settings → Email

Best Practices

1. Use role-based mailing lists (`sq-quality-leads@`, `sq-ops@`)
2. Scope notifications: personal (issue assignments) vs global (quality gate changes)
3. Enable project-level subscriptions for release managers
4. Document escalation path: failure → team → platform ops → security

Auditing

- Review `User → My Account → Notifications` during onboarding
- Export notification matrix quarterly to ensure coverage

Plugin Governance

Lifecycle

1. Evaluate plugin (compatibility, license, maintenance cadence)
2. Test on staging instance; run regression analysis on key projects
3. Approve via change advisory board
4. Install via Marketplace or manual JAR deployment
5. Document version, owner, rollback plan

Risk Controls

- Maintain allowlist; block unsigned or deprecated plugins
- Track plugin security advisories from SonarSource
- Schedule quarterly review for upgrades or removals
- For custom rules, store source in version control and package via CI

Sonar Processes & Background Tasks

Key Processes

- `Compute Engine`: processes analysis reports, executes background tasks
- `Elasticsearch`: rebuilds indexes, powers search
- `Web Server`: serves UI/API, handles authentication flows

Background Task Console

```
Administration → Projects → Background Tasks
Columns: Status, Type, Duration, Worker, Error
```

Typical Jobs

- Report processing (`REPORT_PROCESS`)
- Quality gate evaluation (`QGATE`)
- Issue synchronization (`ISSUE_SYNC`)
- Project purge (`PURGE`)

Operational Tips

- Maintain queue under five pending tasks; investigate spikes
- After upgrades, watch for "Fail to process" errors in `ce.log`
- Schedule housekeeping to purge inactive projects quarterly

Operational Monitoring Points

Health Checklist

- Web latency < 300 ms (reverse proxy metrics)
- Compute Engine throughput: average task < 2 minutes
- JVM heap usage < 75% sustained (JMX exporters)
- Elasticsearch cluster status `green`
- Database connection pool usage < 80%

Dashboards

1. Infrastructure: CPU, RAM, disk I/O, network
2. Application: logged errors, background queue depth, webhook success rate
3. Quality KPIs: gate pass rate, new vulnerabilities per sprint

Alerting

- Hook Prometheus/Grafana or Stackdriver to JVM/JMX exporters
- Configure HTTP probes for `/api/system/status` expecting `UP`
- Notify ops if storage usage > 85% or logs stop rotating

Analysis Lifecycle Review

Step-by-Step

1. Scanner packages sources, metrics, coverage artifacts
2. Report uploaded to server endpoint
3. Compute Engine task created (`REPORT_SUBMITTED`)
4. Task processed: issue detection and gate evaluation
5. Background jobs update indexes and measures
6. Results persisted to database and caches
7. Notifications and webhooks triggered
8. Dashboards refreshed; developers receive feedback

Troubleshooting

- Stuck tasks: check `ce.log` and restart Compute Engine service
- Missing coverage: verify report paths and scanner properties
- Duplicate analysis: ensure unique `sonar.projectKey`

Webhook Integrations

Use Cases

- Notify CI/CD orchestrators to proceed or roll back
- Update ALM tools (Jira, Azure DevOps) with quality status
- Trigger Slack or Teams alerts when gates fail
- Feed DataOps dashboards for compliance evidence

Configuration

```
Administration → Configuration → Webhooks → Create
Name: Quality Gate to GitLab
URL: https://gitlab.example.com/api/qualityhook
Secret: <shared HMAC token>
```

Payload Snapshot

```json
{
  "taskId": "AYkv8eKpYyS6QfQ",
  "status": "SUCCESS",
  "qualityGate": { "name": "Data Pipelines", "status": "OK" },
  "project": { "key": "etl-dataflow", "name": "ETL Dataflow" },
  "branch": "main"
}
```

Operational Notes

- Retry policy: SonarQube retries up to three times with backoff
- Keep endpoints idempotent and validate HMAC signature
- Monitor delivery stats via Webhooks page (Success vs Failed)