Code Quality Control in the CTAO Project

A SonarQube-based approach: metrics, quality gates, and workflow

Sebastiano Spinello
Dr. Federico Incardona
Kevin Munari

What is SonarQube?

Automated code review

SonarQube is a platform for automated code review and static analysis.
It detects code quality and security issues across many languages.

Quality standards

It measures reliability, security, and maintainability using a configurable set of rules.
It reports issues, trends, and metrics to support engineering decisions.

Pull request integration

Run analysis in CI for each pull request and main-branch build.
Use quality gates to provide an objective pass or fail signal.

Where it fits in a DevOps pipeline

Build and test | **Analyze** | **Quality gate** | **Feedback**

Build and run tests, then publish coverage reports.
Run SonarQube analysis and upload results.
Evaluate the quality gate to decide whether to merge or release.
Send feedback to developers in the pull request and backlog.

What SonarQube Measures

Bugs

Reliability issues that can cause incorrect behavior

(for example, null dereference, resource leaks).

Vulnerabilities

Security weaknesses with high confidence

(for example, injection, hardcoded secrets).

Code Smells

Maintainability issues that increase technical debt

(for example, duplication, complexity).

Supporting metrics (how we quantify quality)

- Coverage
- Duplications
- Technical Debt
- Quality Gate

Coverage: proportion of code exercised by tests (line and condition coverage).

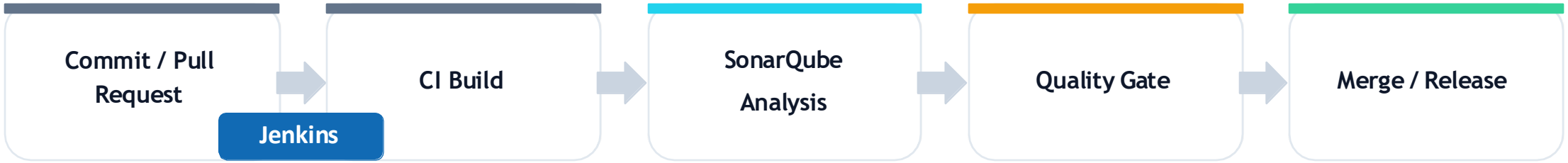
Duplications: repeated code blocks that increase change risk and maintenance cost.

Technical Debt: estimated remediation effort for maintainability issues.

Quality Gate: rules that convert metrics into pass or fail (often used to block merges).

CI Integration on CTAO Project

CTAO Project Setup



- Run analysis in CI for every pull request and main-branch build.
- Jenkins is a common CI/CD automation server used to orchestrate builds, tests, and deployments.
- Jenkins can be triggered by Git webhooks, run pipeline stages, and integrate with SonarQube to enforce quality gates.
- Publish the Quality Gate status to the pull request for fast feedback.
- Use trend views to confirm that improvements are sustained.

ACADA MON/AAS Project

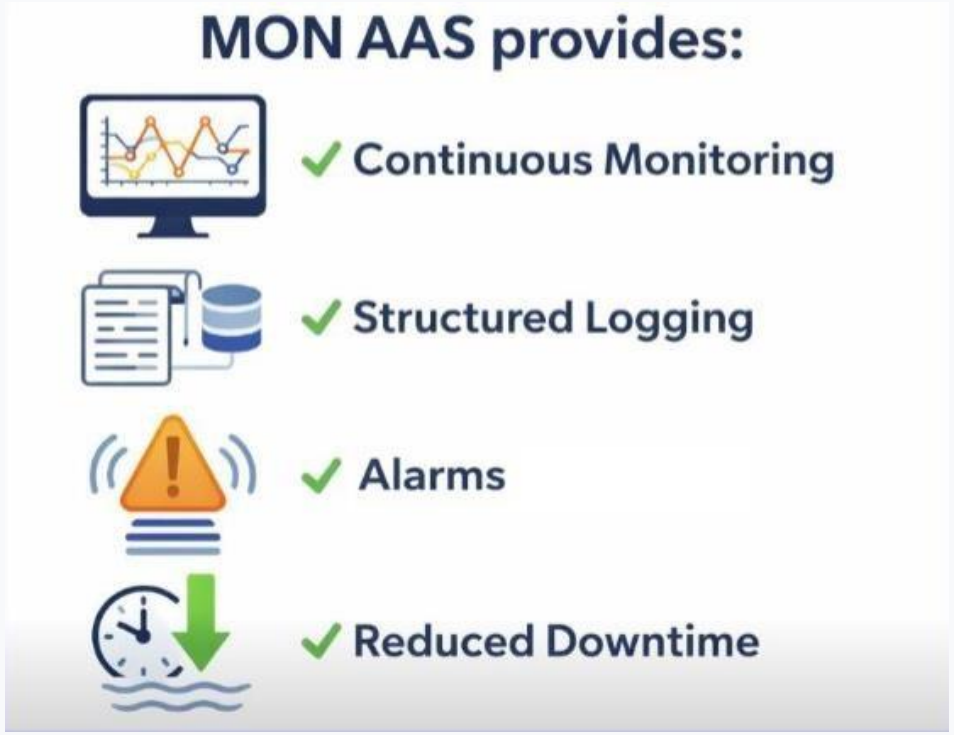
The core of CTAO inspection and diagnosis

Monitoring and Logging System (MON)

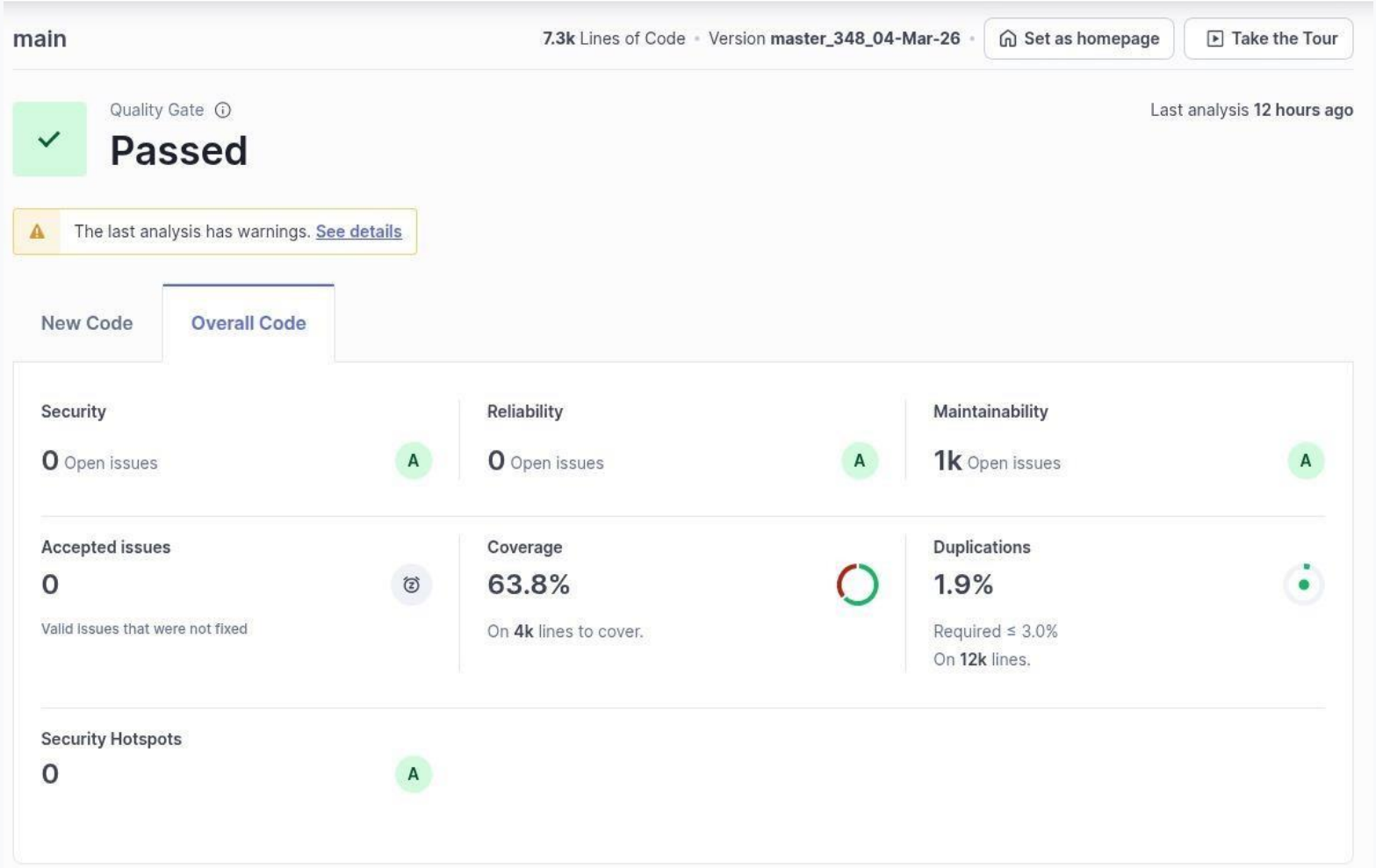
- Collect, elaborate, distribuite and archive data coming from telecopes, auxiliary and calibration instruments

Array Alarm System (AAS)

- Collect, filter, shelve, reduce, acknowledge and combine alarms from Array Elements (telescopes and instruments)



SonarQube Overview for MON



What you see

Quality Gate result (pass or fail) for the branch.

Ratings A to E for Security, Reliability, and Maintainability.

Coverage percentage, duplications percentage, and issue counters.

What it means

The gate is an automated decision point for merge or release (often on Overall Code parameter).

Security and Reliability summarize risk (vulnerabilities and bugs).

Coverage and duplications indicate change safety and refactoring cost.

Typical actions

If the gate fails on New Code, fix issues before merging.

Keep duplication low by extracting shared code.

Increase New Code coverage gradually and manage legacy as backlog.

Interpreting Ratings and Thresholds

A to E ratings (SonarQube defaults)

A — Security and Reliability: no vulnerabilities or bugs. Maintainability: technical debt ratio up to 5%.

B — Minor vulnerability or bug, or debt ratio from 5% to under 10%.

C — Major vulnerability or bug, or debt ratio from 10% to under 20%.

D — Critical vulnerability or bug, or debt ratio from 20% to under 50%.

E — Blocker vulnerability or bug, or debt ratio 50% or higher.

How ratings are used

Use **Security and Reliability** ratings to prioritize risk.

Use **Maintainability** rating (debt ratio) to guide refactoring decisions.

Combine ratings with **New Code** conditions in the quality gate.

Review thresholds periodically based on incidents and delivery goals.

Technical debt refers to the future cost associated with relying on shortcuts or suboptimal decisions made during software development.

Quality Gate Conditions

Optimal conditions on new code

New **Bugs** must be zero (Reliability).

New **Vulnerabilities** must be zero (Security).

Coverage on New Code must be at least 70 to 90 percent.

Duplications on New Code must be at most approximately 3 percent.

Maintainability on New Code remains A.

"Clean as You Code" model

A software development methodology that aims to maintain high code quality by focusing only on new or modified parts, rather than trying to clean up entire legacy codebases.

New Code (Policy Focus)

Keep new changes clean: add tests and avoid new critical issues.

This prevents quality from degrading over time.

Apply stricter gate rules to New Code to stop quality regression early.

Overall Code (Backlog)

Existing issues are managed as a backlog.

Prioritize by severity, effort, and change frequency.

Treat legacy issues as a backlog and improve incrementally during planned work.

SonarQube Rules and Accepted Issues

Rules

Rules are checks executed on code during analysis.
When code violates a rule, **SonarQube** creates an issue.

Issue categories

Bugs, vulnerabilities, code smells, and security hotspots.
See the next slide for definitions.

Accepted issues

Teams can mark issues as Accepted or False positive.
These resolutions remove the issues from the quality gate and ratings.

Best practices and limitations

- **Use sparingly**
- **Require justification**
- **Review regularly**
- **Avoid hiding risk**

Use Accepted and False positive only after the team reviews the context.
Record the rationale and owner in comments or tags.
If many issues become false positives, adjust the quality profile or analysis scope.
Do not use acceptance to bypass security or release controls.

Security Hotspots and Vulnerabilities

Key distinction

Security Hotspot: a security-sensitive piece of code that doesn't impact the overall application security.

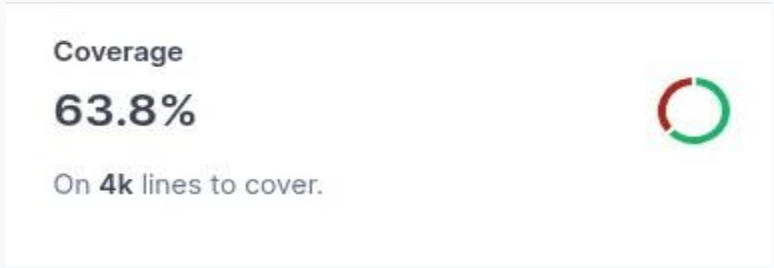
An **example**: password, API key hard-coded in the source code or using obsolete cryptography algorithms generate a security hotspot.

Vulnerability: a problem that impacts the application's security and needs to be fixed immediately.

An **example**: Accepting serialized objects from untrusted sources, which can lead to remote code execution or SQL Injection, unvalidated user input passed directly into a database query, allowing an attacker to manipulate data.

It's up to the developer to review the code to determine whether or not a fix is needed to secure the code. Hotspots are not necessarily exploitable until reviewed in context.

Key Metrics Summary



Practical interpretation

Coverage: set a realistic target (e.g., 70%) and increase it gradually; prioritize *critical modules* and *bug-prone areas*.

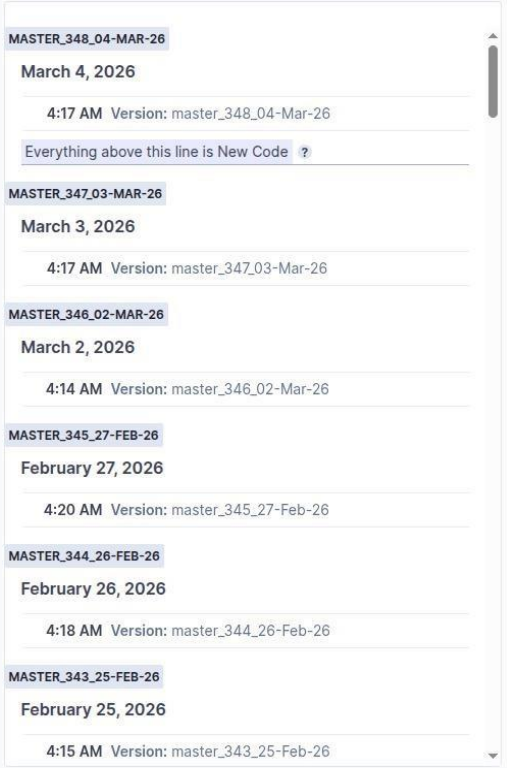
Maintainability: reduce technical debt by tackling **high severity** and **low effort** issues first; focus on the most frequently changed files to maximize impact.

Duplications: address duplication spikes in core modules; extract shared methods/components to reduce **regression risk** and change cost.

Trends (Activity): track week-over-week movement; the goal is steady improvement without "bounce-backs" after releases and hotfixes.

Risk lens: prioritize items that impact **security**, **stability**, and **delivery time** (lead time / rework).

Trends: Coverage Over Time (Activity)



What you see

Coverage trend across versions and releases.

Lines to cover compared with covered lines.

Events and versions to correlate with changes.

What it means

Drops after releases often indicate new code without tests.

Stable trends indicate consistent discipline.

Use trends to identify regressions early.

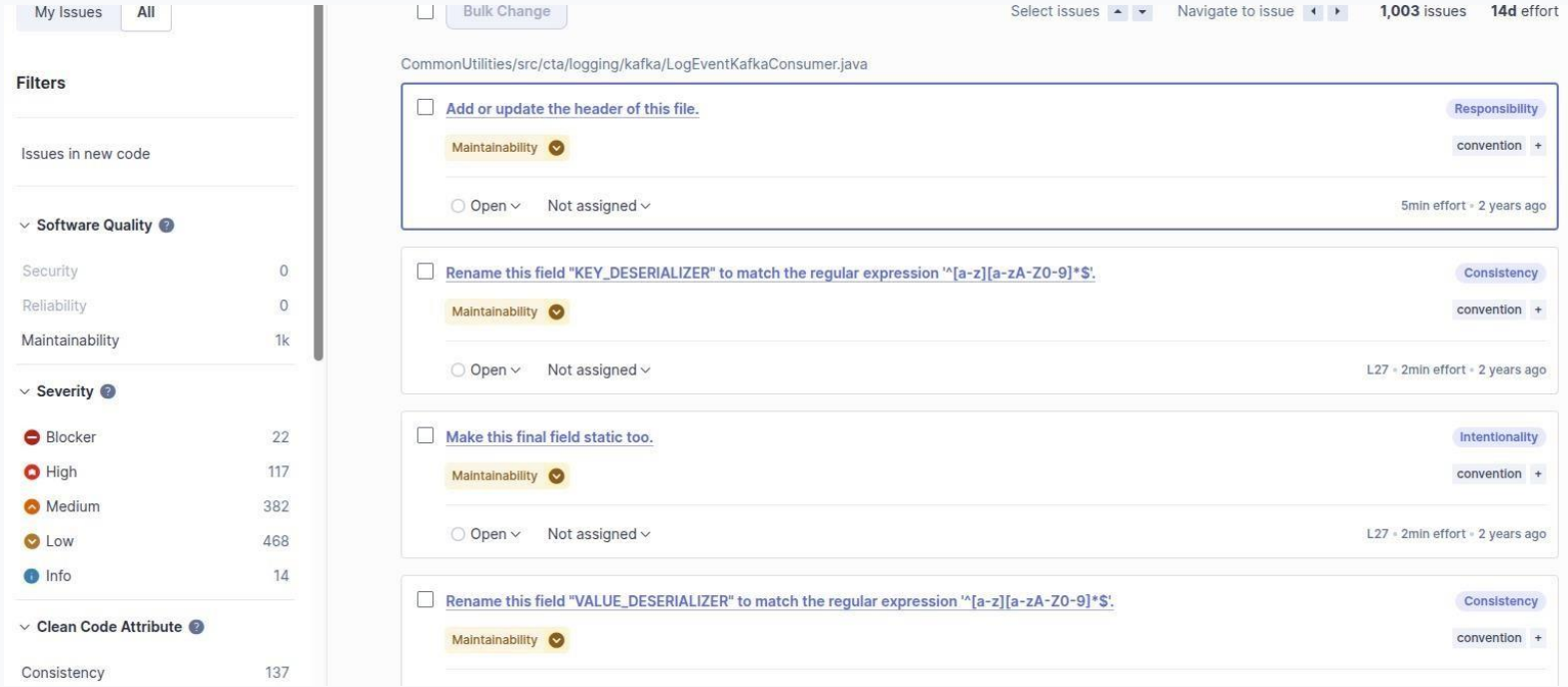
Typical actions

Enforce New Code coverage thresholds in the quality gate.

Investigate drops and identify which modules changed.

Add tests around critical paths and recent changes.

Issues: Maintainability Backlog



What you see

List of issues with type, severity, and location.

Effort estimates (minutes or hours) per item.

Filters to focus on modules, severities, and rules.

What it means

Severity indicates priority (address Blocker and High first).

Effort supports planning quick wins compared with larger refactors.

The backlog view turns analysis into actionable work.

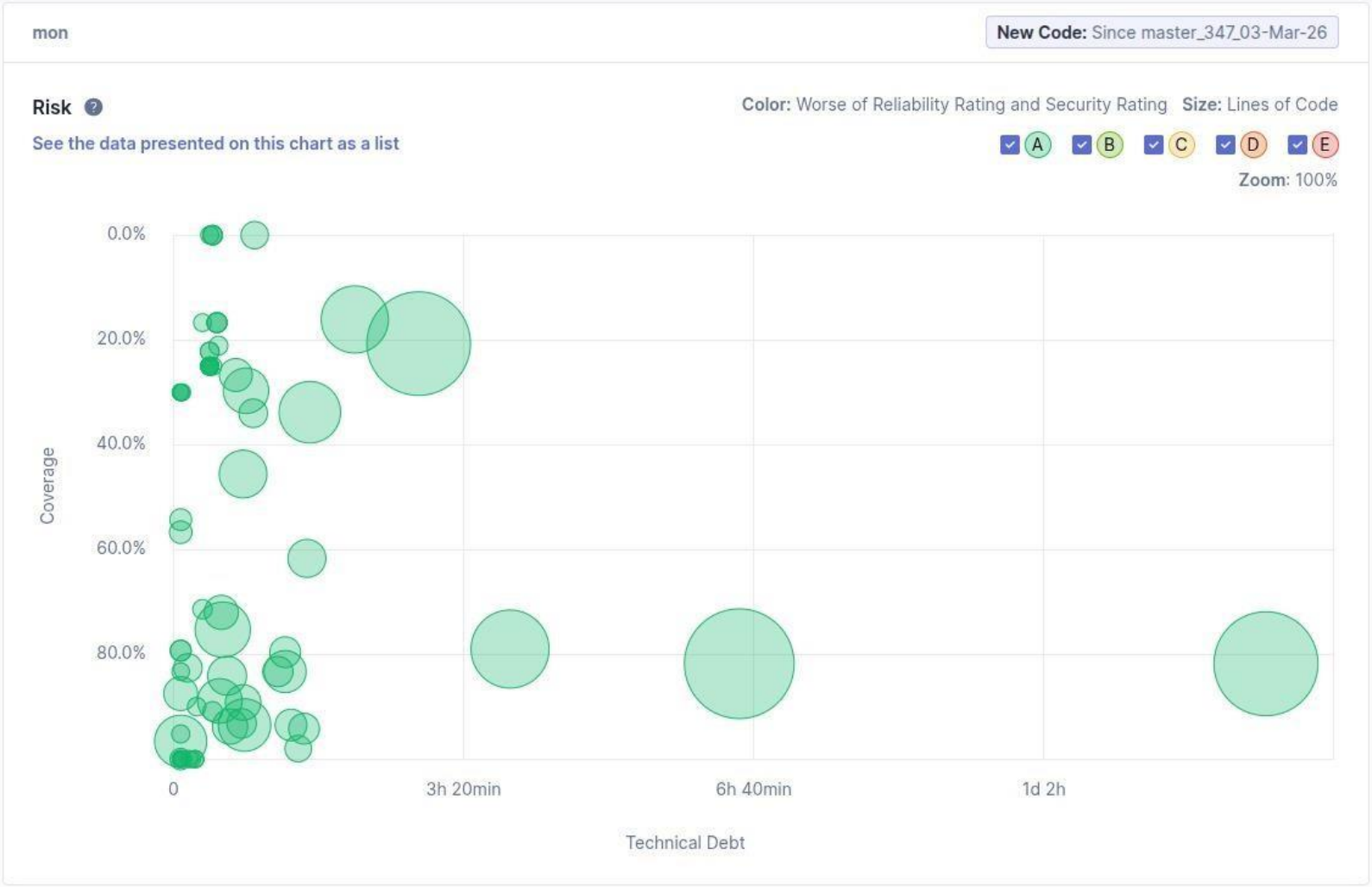
Typical actions

Fix New Code issues in pull requests to keep the quality gate passing.

Batch quick wins by sorting by effort.

Schedule larger refactors as planned work.

Measures: Risk Map



What you see

Bubble chart per component or module.

X-axis: technical debt (remediation effort).

Y-axis: coverage; bubble size: lines of code.

What it means

Large modules with low coverage indicate high regression risk.

Large modules with high debt slow delivery over time.

This view helps prioritize refactoring by return on investment.

Typical actions

Select one or two high-risk modules per sprint.

Add tests first, then refactor to reduce debt and duplication.

Re-check after changes to validate improvement.

File-Level Metrics

	Lines of Code	Security	Reliability	Maintainability	Security Hotspots	Coverage	Duplications
mon	7,264	0	0	1,003	0	63.8%	1.9%
CommonUtilities	3,490	0	0	379	0	84.1%	1.1%
LoggingCollector	299	0	0	15	0	16.1%	3.3%
LoggingStorage	89	0	0	11	0	34.0%	13.2%
LoggingSupervisor	113	0	0	55	0	26.7%	0.0%
MonitoringCollector	2,583	0	0	380	0	58.4%	2.6%
MonitoringStorage	497	0	0	66	0	20.7%	2.5%
MonitoringSupervisor	193	0	0	97	0	45.6%	0.0%

What you see

Per-file metrics: coverage, duplications, and issues.

Highlights files concentrating risk and debt.

Useful to decide where to invest first.

What it means

Low coverage reduces confidence during change.

High duplication increases inconsistency risk and maintenance cost.

High issue density indicates complexity and future cost.

Typical actions

Write tests around current behavior first.

Reduce duplication through extraction and reuse.

Re-run analysis to verify improvements.

Inline Code View

What you see

Inline markers on exact lines in the source file.

Navigation between issues in the file.

Red lines refer to test, grey lines to duplicatio, green lines refer to correctly tested code.

What it means

Rule identifiers point to documented best practices.

Messages explain risk (readability and correctness).

Some issues are quick fixes; others require structural refactoring.

Typical actions

Fix simple issues immediately (rename, simplify).

Add tests before structural refactoring.

If appropriate, document and resolve as "Accepted Issue".

ACADA Release Conditions

ACADA-REL2-QG

Conditions

Conditions on Overall Code

Conditions on overall code apply to branches only.

Metric	Operator	Value
Duplicated Lines (%)	is greater than	3.0%
Line Coverage	is less than	60.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	C
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

These metrics are reviewed monthly across subsystems during sprint reviews to track trends and ensure consistent release readiness. Teams validate metrics to monitor progress and prevent quality regressions.

Quality Metrics

Quality criteria based on CTA-PLA-ACA-303000-0008_ACADA Verification, Validation and QA Execution Plan Chapter 5.1.1

🟢 Value meets quality criteria

🔴 Value does not meet quality criteria

⬆️ Value improved

⬇️ Value got worse

IMPORTANT NOTE: As SonarQube doesn't correctly cover code based on C++ templates, some values are below the actual line coverage.

2026-03-02 11:34:19.270432

Subsystem	Maintainability (max 5%)	Reliability (max C)	Security (max A)	Security Hotspots Reviewed (100.0%)	Duplicated lines (< 3.0%)	Line coverage (> 60.0%)
aas	🟢 2.5 (2.5)	🟢 A (A)	🟢 A (A)	🟢 None (None)	🟢 1.8 (1.8)	🟢 60.7 (60.7)
acada-cdb-acs	🟢 0.0 (0.0)	🟢 A (A)	🟢 A (A)	🟢 None (None)	🟢 0.0 (0.0)	🟢 70.4 (70.4)
acada-cli-tools	🟢 0.1 (0.1)	🟢 A (A)	🟢 A (A)	🟢 100.0 (100.0)	🟢 0.0 (0.0)	🟢 73.0 (73.0)
array-data-handler_cpp	🟢 3.9 (3.9)	🟢 A (E) ⬆️	🟢 A (A)	🟢 100.0 (100.0)	🟢 2.2 (2.2)	🟢 67.1 (66.9) ⬆️
array-data-handler_py	🟢 0.5 (0.5)	🟢 B (A) ⬇️	🟢 A (A)	🔴 0.0 (nan)	🟢 2.2 (2.3) ⬆️	🟢 74.0 (73.7) ⬆️
cdb	🟢 0.0 (0.1) ⬆️	🟢 A (A)	🟢 A	🔴 40.0	🟢 0.0	🟢 79.5