Building a Quality Assurance Feedback System for Data Pipelines

MASTER THESIS

Celine Pöhl

Submitted on 27 October 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Julian Hirsch
Prof. Dr. Dirk Riehle, M.B.A.

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

München, 27 October 2025

License

München, 27 October 2025

Abstract

This thesis presents a lightweight, GitHub-native quality assurance (QA) system for data pipelines. The solution consists of two artifacts: a QA dashboard that summarizes the current quality state, and a QA pull request comment bot that shows change-focused metrics. Metrics focus on four quality characteristics: maintainability, functional suitability, performance efficiency, and security. A configuration-driven pipeline turns raw metrics into stable JSON artifacts (baseline and deltas) and renders them as Markdown, keeping results reproducible and easy to review. The approach remains extensible. Promising next steps include performance testing, more metrics, and refined visualization.

Contents

List of Figures

List of Tables

Acronyms

CC Cyclomatic Complexity

CI Continuous Integration

CI/CD Continuous Integration (CI) and Continuous Delivery (CD)

FP Function Point

GQM Goal-Question-Metric

IEC International Electrotechnical Commission

ISO International Organization for Standardization

KLOC Thousands of Lines of code

LOC Lines of Code

MI Maintainability Index

PR Pull Request

QA Quality Assurance

SQuaRE Software product Quality Requirements and Evaluation

1 Introduction

Modern software projects with many contributors create a constant stream of new code. Reviewers have limited time, and important quality signals can be easy to miss. Feedback is often delayed or spread across multiple tools, making it difficult to see whether a change improves or harms the system. This challenge is even stronger in student-heavy projects where many people work on the same codebase. Reviewers need compact and reliable cues for QA.

This thesis, *Building a Quality Assurance Feedback System for Data Pipelines*, presents a practical, GitHub-native solution made of two main parts: a QA dashboard that shows the current state of quality, and a QA PR comment bot that highlights how the quality changed. The goal is to give helpful, non-blocking feedback that supports coding and review without adding complex tools. Results are transparent and reproducible. The system is open for future extensions as quality goals evolve. The work focuses on four quality areas that fit continuous integration and everyday review: maintainability, functional suitability, performance efficiency, and security. It assumes a Python-based codebase in a GitHub environment. The chosen metrics are motivated by the ISO/IEC 25010 model and the GQM approach to keep the results meaningful for reviewers.

Software QA has always faced a gap between theory and practice. Many metric systems and quality models are well-designed in theory, but they often do not align with the daily rhythm of development teams. Dashboards or reports are sometimes produced only after a release, when the information is already outdated. In fast Continuous Integration (CI) and Continuous Delivery (CD) (CI/CD) environments, feedback must be immediate and connected to the developer's context. This thesis fills that gap by adapting established quality principles into a lightweight, automated feedback system that integrates naturally with GitHub.

The ISO/IEC 25010 product-quality model gives a theoretical base, and the GQM method adds quality goals to concrete questions and metrics. Combining both methods keeps the system evidence-based and goal-oriented. The selected metrics are not random numbers but answers to explicit quality questions.

This thesis demonstrates how to realize this combination using open-source tools and a clear configuration, making the setup reusable for other projects.

In addition to the choice of quality metrics, this work also aims to provide clear visibility into quality. By embedding measurement and feedback directly into GitHub, quality becomes a visible part of every code change. This visibility benefits both professional teams and educational projects, as transparent metrics can guide code review and learning. The two resulting artifacts (the QA dashboard and the QA PR comment bot) demonstrate how formal quality standards can be adapted for collaborative software development. In this way, the thesis connects models of software quality with the needs of everyday code review.

The main contributions are the two artifacts themselves and their usage. The QA dashboard provides an overview of the repository's current quality, making the current status easy to check. The QA PR comment bot signals the quality change the PR code would introduce, so attention is directed to the most relevant parts of the code. Both outputs are Markdown-based.

The thesis is organized as follows. In chapter 2 *Literature Review*, the quality framework and the metric families that guide the implementation is introduced. Building on this, chapter 3 *Requirements* describes the functional and non-functional needs and narrows the scope to four selected quality areas. It also describes how the selected quality characteristics lead to questions with the GQM model. All requirements are categorized through the MoSCoW method. In chapter 4 *Architecture*, the components architecture of the two deliverables is demonstrated, showing how they work together and how data moves from collection to presentation. In chapter 5 *Design and Implementation*, all components are explained in detail. Finally, chapter 6 *Evaluation* compares the system against the defined requirements, focusing on how many of the MoSCoW requirements are met. In chapter 7 *Conclusions*, the lessons learned are summarized and possible future improvements are highlighted.

2 Literature Review

This chapter outlines the foundations of software quality measurement. It introduces important metrics such as Lines of Code (LOC), Function Point (FP), Cyclomatic Complexity (CC), and the Maintainability Index (MI), explaining their relevance for assessing different aspects of software quality. It also introduces standardized frameworks like ISO/IEC 25010 and the Goal-Question-Metric (GQM) paradigm, which guide structured and goal-oriented measurements. Finally, recent work on automated dashboards and review bots is discussed to show how successfully these concepts are applied in practice.

2.1 Milestones in Software Metrics Development

The development of software quality metrics is closely tied to the history of software engineering (Fenton & Neil, 1999). From the early days of programming, metrics were used to measure a software product's quality and productivity. These metrics were usually based on Lines of Code (LOC) measures (Jones, 2008, p. 72). Over time, other metrics and frameworks were developed that focused on broader quality aspects such as maintainability, reliability, and usability, e.g., the Goal-Question-Metric (GQM) approach of Basili and Rombach (1988) or object-oriented measures like coupling and cohesion (Fenton & Neil, 1999). The following outlines important software quality metrics, highlighting the key approaches and developments that have shaped the field.

2.1.1 Lines of Code

The earliest use of Lines of Code (LOC) as a measure of software size dates back to the 1950s, when it was used to estimate programmer productivity and defect density (Jones, 2008, p. 72). With the growth of application sizes and the introduction of higher-level languages, LOC became increasingly popular but also began to show its limitations.

LOC was a standard measure, applied both to evaluate programmer productivity (by calculating LOC per month) and to assess software quality through defect density per Thousands of Lines of code (KLOC). LOC functioned as an early metric for various concepts of software size (Fenton & Neil, 1999).

By the mid-1970s, the limitations of LOC as a measure for effort, functionality, or complexity had become evident (Fenton & Neil, 1999). Although simple to collect, it became clear that LOC suffered from significant drawbacks. The spread of diverse programming languages highlighted this issue, since a line of assembly code could not be meaningfully compared to a line of a high-level language. For instance, productivity metrics such as LOC per month or cost per LOC may give the impression that assembly language programmers are more productive than those using high-level languages, even though total programming costs are generally lower with high-level languages (Mills, 1988).

As a result, the period from the mid-1970s onward brought many new metrics, including complexity metrics such as those proposed by McCabe (1976) and Halstead (1977), as well as size-oriented methods like Albrecht's function points (1979) which aimed to be language-independent (Fenton & Neil, 1999).

2.1.2 Function Point Metric

Another disadvantage of the LOC metric is that it can only be calculated exactly once the software is fully implemented. However, it needs to be available much earlier in the development process to be useful. This limitation was a key driver behind the Function Point (FP) metric, as introduced by Albrecht (1979), which enables estimation independent of completed code (Mills, 1988). The FP metric was the first metric used for measuring economic productivity (Jones, 2008, p. 75). Unlike LOC, the FP metric measures the size of a system based on the functionality delivered to the user, considering inputs, outputs, inquiries, internal data files, and external interfaces. Because FP can be estimated from requirements rather than completed code, it quickly became an important, language-independent metric (Mills, 1988). Today, the FP metric lost its relevance because it must be captured manually and is only applicable in limited contexts (Chillarege, as cited in Voas & Kuhn, 2017).

2.1.3 Cyclomatic Complexity

The Cyclomatic Complexity (CC) metric was introduced by McCabe (1976) as one of the earliest systematic attempts to measure program complexity (Kafura, 2025). It measures the number of linearly independent paths through a program's control flow graph. Unlike LOC, this metric captures structural complexity and correlates better with maintainability and testing effort.

McCabe's approach also began to move metrics toward independence from programming language by focusing instead on logical structure. The CC metric has several favorable characteristics that make it popular to this day. It can be calculated simply, is language-independent, and useful for programming and testing. It has also proven helpful in identifying modules that may be overly complex and could benefit from redesign to enhance maintainability.

2.1.4 Halstead Metrics

Halstead (1977) tried to counter the ambiguity of LOC with his Halstead metrics (Jones, 2008, p. 116). To address this, he proposed a new approach known as *software science*, which analyzed programs in terms of their fundamental components. He proposed a set of metrics based on the number of unique operators, the number of unique operands, the total number of operators, and the total number of operands. From these, he derived secondary measures, such as the program's vocabulary (the sum of unique operators and operands) and its length (the sum of all operators and operands). These measures attempted to quantify aspects such as effort, difficulty, and even the time required to implement or understand code. Halstead's metrics represented a significant step in treating software as an object of scientific measurement.

2.1.5 Goal-Question Metric

By the late 1980s, it became clear that no single metric could serve all purposes. Basili and Rombach (1988) introduced the GQM paradigm, which emphasizes tailoring measurement programs to specific organizational goals (Jones, 2008, p. 181). The GQM approach provides a structured framework for defining and applying software metrics. Instead of starting with predefined metrics, GQM began with goals, derived questions to evaluate those goals, and only then identified the appropriate metrics. It follows a hierarchical logic: starting from high-level business goals, moving to specific questions that clarify those goals, and finally deriving concrete metrics that allow progress to be measured. This approach shifted the focus from universal measures to context-sensitive measurement programs. In section 2.3, a detailed explanation of GQM will be provided, and subsection 3.6.2 in chapter 3 *Requirements* will apply GQM to this project as part of the requirements for the measurement approach.

2.1.6 Maintainability Index

The Maintainability Index (MI) was introduced by Oman and Hagemeister (1992) as a possibility to assess software maintainability (Heričko & Šumak, 2023). Initially formulated using Halstead's Volume, CC, and LOC, the MI aims to provide a single numerical value indicating how easy a software can be maintained.

2.1.7 Current Practice of Software Metrics

In the 1990s, Fenton and Neil (1999) observed that most IT companies measure software quality with metrics. However, the authors criticize the overreliance on simple measures such as LOC and defect counts. Despite decades of research, the state of software quality metrics reveals less radical change than one might expect. The at the time more recent approaches, such as Bayesian networks or machine learning for defect prediction, remain largely confined to research. In practice, organizations tend to rely on a small set of well-established, easy-to-collect metrics rather than adopting novel proposals.

Today, the field of software metrics looks different from its early days. The paper *What Happened to Software Metrics?* by Voas and Kuhn (2017) presents the outcomes of a roundtable discussion with seven experts in software metrics, reflecting on the successes and failures of the past 40 years. Over the past few decades, software metrics have played a crucial role in enhancing software quality, although their effectiveness often depends on the specific context. As Alain Abran warns, "The software industry, by contrast, has unrealistic expectations that poorly defined 'metrics' can solve complex problems at almost zero cost," (p. 88) and he recommends "a full set of measurement standards" (p. 88) rather than a single metric. Victor Basili emphasizes goal-oriented measurement, saying, "You should select measures based on what you want to know and what you're going to do with that information" (p. 89). Porter reminds us, "The job defines which tools are right, not the other way around" (p. 89), highlighting that metrics are most useful when carefully chosen for the task at hand.

The historical evolution of metrics laid the groundwork for later efforts to formalize quality assessment through the development of international standards and models. To provide a structured approach to evaluating software quality, the ISO/IEC 25000 Software product Quality Requirements and Evaluation (SQuaRE) series was developed (International Organization for Standardization & International Electrotechnical Commission, 2014). The following section will focus on quality characteristics defined by this set of standards.

2.2 Standardized Quality Characteristics

SQuaRE is an international framework for defining and specifying software product quality (International Organization for Standardization & International Electrotechnical Commission, 2014). It was developed to replace and unify older quality standards, primarily ISO/IEC 9126 and ISO/IEC 14598.

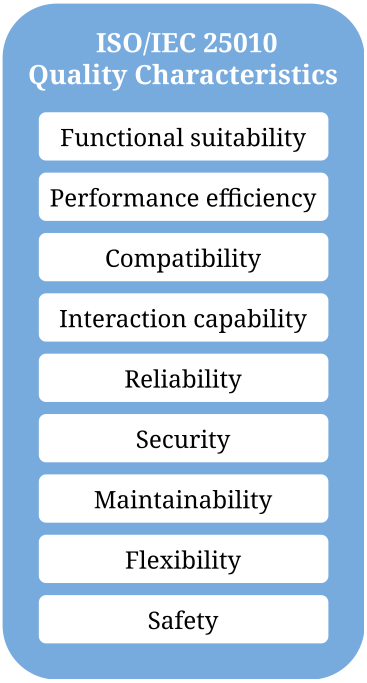

Figure 2.1: Nine Quality Characteristics (International Organization for Standardization & International Electrotechnical Commission, 2023)

Within this series, ISO/IEC 25010 defines a quality model that identifies key software quality characteristics (International Organization for Standardization & International Electrotechnical Commission, 2023). As seen in Figure 2.1, the nine quality characteristics are *functional suitability*, *performance efficiency*, *compatibility*, *interaction capability*, *reliability*, *security*, *maintainability*, *flexibility*, and *safety*.

Functional suitability "degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions" (ISO 25000 Portal, 2024). It includes functional completeness (coverage of all required tasks), functional correctness (accuracy of results), and functional appropriateness (support for user goals). High functional suitability ensures that the software fulfills its intended purpose.

Performance efficiency evaluates how "well a system performs its functions within specified time and throughput parameters and is efficient in the use of resources" (ISO 25000 Portal, 2024). It includes time behavior (response and processing times), resource utilization (use of memory, CPU, bandwidth), and capacity (the limits of the system).

Compatibility measures how well a system can operate with other systems or components (ISO 25000 Portal, 2024). It covers co-existence (ability to share an environment without conflict) and interoperability (ability to exchange information and use it effectively). Strong compatibility ensures that software integrates smoothly into heterogeneous system environments.

Interaction capability refers to the degree to which a system enables efficient interaction with users (ISO 25000 Portal, 2024). It includes aspects such as learnability, user engagement, inclusivity, or user assistance.

Reliability indicates the ability of software to perform its "specified functions under specified conditions for a specified period of time" (ISO 25000 Portal, 2024). It includes faultlessness, availability, fault tolerance, and recoverability. Reliable systems build user trust and minimize operational disruptions.

Security assesses how well the software "defends against attack patterns by malicious actos and protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization" (ISO 25000 Portal, 2024). It includes confidentiality, integrity, non-repudiation, accountability, authenticity, and resistance. Strong security ensures that the system can resist malicious attacks and maintain data integrity and trustworthiness.

Maintainability describes how easily a system "can be modified to improve it, correct it or adapt it to changes in environment, and in requirements" (ISO 25000 Portal, 2024). Its subcharacteristics include modularity, reusability, analysability, modifiability, and testability. High maintainability reduces long-term costs and increases the aptability to evolving requirements.

Flexibility measures how easily a system can be adapted, configured, or extended for evolving requirements (ISO 25000 Portal, 2024). It includes adaptability, scalability, installability, and replaceability, emphasizing the system's ability to respond to change without excessive redesign or redevelopment effort. High flexibility contributes to longer software lifecycles and better alignment with dynamic business or technical environments.

Safety assesses the extent to which a system mitigates the risk of harm to people, property, or the environment during operation (ISO 25000 Portal, 2024). It includes risk identification, being fail-safe, hazard warnings, and safe integration. It focuses on preventing minimizing potential damage caused by failures.

In subsection 3.6.2 *Mapping Quality Characteristics to Questions* of chapter 3 *Requirements*, the quality scope of this project's requirements is set to four of these quality characteristics: *functional suitability*, *performance efficiency*, *security*, and *maintainability*.

2.3 Goal-Question Metric (GQM)

As discussed in subsection 2.1.5 *Goal-Question Metric*, by the late 1980s, it became clear that software metrics could not be applied universally. No single metric is suitable for all projects or contexts. In response to this realization, Basili and Rombach introduced the GQM paradigm in 1988.

GQM provides a methodological framework for tailoring metrics to the specific needs and objectives of a use-case (Jones, 2008, p. 181). Instead of beginning with a predefined set of metrics, as had often been the case, GQM starts with articulating goals. From these goals, questions are derived that clarify them and ultimately identify the metrics needed to provide answers to those questions. This approach enables metrics to be directly aligned with organizational concerns, increasing their relevance and usefulness.

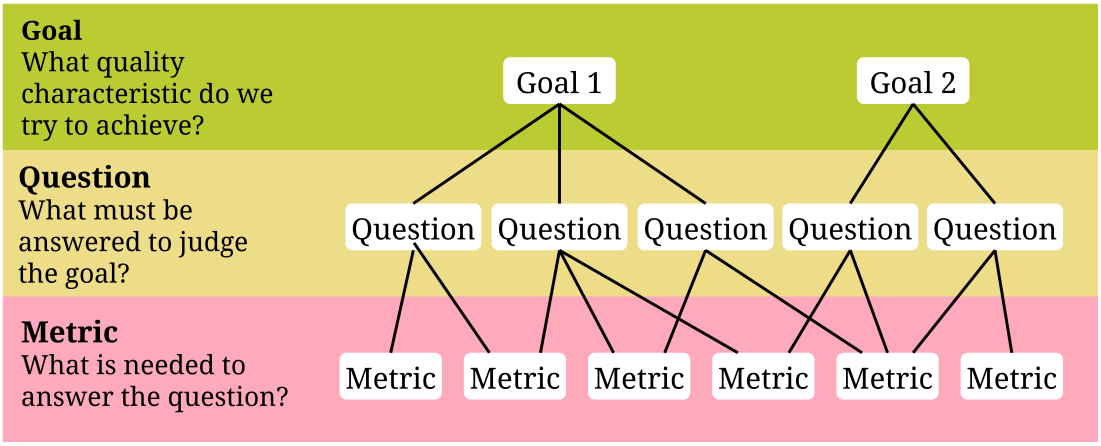

Figure 2.2: The Goal–Question–Metric (GQM) paradigm illustrating the relationship between goals, questions, and metrics. Adapted from Basili et al. (1994, Figure 1)

The structure of GQM is hierarchical and systematic (Basili et al., 1994). In Figure 2.2, the relationship between goals, questions, and metrics is shown. At the top level, broad goals concerning software quality are defined. These goals are then translated into questions, which focus attention on specific aspects of the goal that require clarification in order to decide if the goal is met. Finally, concrete metrics are defined to provide measurable data that can answer these questions. In this way, GQM builds a direct logical chain from abstract objectives to concrete metrics. GQM ensures that each metric has a clear purpose. This increases the likelihood that measurements will actually inform decision-making and process improvement.

Another advantage of GQM is its adaptability (Basili et al., 1994). While the paradigm offers a general methodology, the specific questions and metrics are always tailored to the project and its context. The same high-level goal may lead to different questions and metrics in different environments.

In subsection 3.6.2 *Mapping Quality Characteristics to Questions* of chapter 3 *Requirements*, the GQM approach will be applied to systematically derive measurable software quality metrics tailored to *MECOIS*, the object of measurement in this thesis.

2.4 Related Work

Research on code review automation shows that lightweight and automatic feedback can positively influence PR outcomes. Wessel et al. (2022) study the adoption of review bots that post automated comments or statuses in PR threads. They report measurable changes in review dynamics, including higher merge rates and fewer abandoned PRs.

Balachandran (2013) describe a review bot that injects static-analysis feedback into code review. They introduce *Review Bot*, a tool that runs automatic checks and posts comments directly in the code review. In a study, almost all of the Review Bot's automatic findings were considered correct and led to actual code changes. The tool also suggests suitable reviewers to fix an issue. It does this by looking at the change history of the code, for example, who has edited or reviewed those specific lines before. Those people are likely familiar with that part of the system, so they are good candidates to review the new change. Overall, combining automatic checks with code reviews saves effort and improves the quality of reviews.

Staron et al. (2014) explore how software companies can use dashboards to monitor quality in agile environments. Traditional management approaches often fail to provide timely insight into product quality. In case studies, they show how dashboards can visualize key indicators. They identify several success factors, including the use of standardized metrics and clear visualizations. Their conclusion is that effective dashboards can improve transparency and also support decision-making.

Lopez et al. (2021) introduce *QaSD*, a ***Q**uality-**a**ware **S**trategic **D**ashboard* for agile settings. QaSD lets users define strategic indicators and connect them to metrics. The system integrates techniques like Bayesian networks for qualitative assessment and predictive analytics for forecasting.

All in all, the related works show that automated QA feedback has the potential to improve software development.

2.5 Key Takeaways and Implications

The literature and standards reviewed in this chapter point to several lessons. First, there is no single best metric: effective metrics depends on their context. The GQM paradigm makes this explicit by starting from stakeholder goals and deriving questions before choosing any metrics. Second, the SQuaRE family provides a structure for developing a QA measurement system. ISO/IEC 25010 describes nine quality characteristics into which metrics fall. Related Work showed that quality dashboards and review bots offer good guidance to monitor a product's quality.

These insights are meaningful for this project. Quality characteristics will be derived before concrete metrics are selected in subsection 3.6.1 *Quality Scope*. Stakeholder-based goals and questions based on GQM will be developed as requirements in subsection 3.6.2 *Mapping Quality Characteristics to Questions*. Afterward, metrics based on them will be implemented in section 5.2 *Metric Collection* of chapter 5 *Design and Implementation*.

3 Requirements

This chapter defines the requirements for the QA feedback system within the *MECOIS* environment. It specifies what the system must achieve to deliver automated quality feedback through two components: the QA dashboard and the PR comment bot. The requirements cover functional and non-functional qualities, as well as an underlying measurement approach. Using the ISO/IEC 25010 quality model and the Goal-Question-Metric (GQM) method, measurable quality goals and corresponding questions are derived for the quality characteristics *maintainability*, *performance efficiency*, *security*, and *functional suitability*. The chapter concludes with a prioritization of requirements based on the MoSCoW method.

3.1 Project context

This thesis was conducted in the context of the *MECOIS* project, which aims to measure and benchmark the productivity of software development teams. *MECOIS* is still in development and is currently being peer-reviewed.

Within this environment, students contribute to the ongoing development. The frequently changing nature of student contributions presents a challenge for maintaining consistent software quality. Reviewers must also assess code from developers with varying levels of experience, which increases the need for helpful and objective feedback.

To address this challenge, this thesis introduces a QA feedback system composed of two complementary components: a dynamic QA dashboard and an advisory QA PR comment bot. Together, these components provide automated feedback on code quality to developers and reviewers.

3.2 Stakeholders and User Stories

3.2.1 Stakeholders

Identifying stakeholders and their needs is essential for defining meaningful requirements for a QA feedback system. The following section identifies the stakeholders of *MECOIS* and expresses their needs as user stories. These practical motivations guide the requirements defined later in this chapter.

The primary stakeholders of this work are the contributors and maintainers involved in the *MECOIS* development environment.

- **Student Developers** contribute code to *MECOIS* and related components as part of research and course projects. They benefit from immediate, structured feedback on code quality, which helps them identify issues early and learn good development practices.
- **Code Reviewers and Teaching Assistants** evaluate PR submitted by student developers. They rely on quality indicators to make review decisions more efficient.
- ***MECOIS* Core Development Team** maintain and evolve the *MECOIS* platform and supporting infrastructure. They ensure the quality is aligned with the project's research goals.

3.2.2 User Stories

The following user stories summarize the needs identified for these stakeholders:

- **As a developer**, I want to receive clear, automated feedback on my PRs so that I can understand and fix quality issues before the review process.
- **As a reviewer**, I want to see summarized quality metrics on changed code so that I can focus my attention on the most relevant parts of a PR.
- **As a member of the *MECOIS* core development team**, I want QA tools to run automatically in GitHub Actions and provide consistent feedback. As a long-term *MECOIS* project maintainer I want that the project maintains a reliable baseline of software quality across all contributions.

These stories capture the different perspectives of users interacting with the QA feedback system and form the foundation for all requirements defined in the following sections.

3.3 Required Solution Components

The QA feedback system is required to be implemented as two components that support each other:

- A dynamic QA dashboard that provides an overview of the current quality state of the *MECOIS* codebase and
- A QA PR comment bot that delivers feedback directly within GitHub PRs

Both components should use a shared quality model and measurement approach defined later in section 3.6. Together, they form an integrated QA feedback system that should provide automated insights throughout the *MECOIS* development process. The goal of the system is not to enforce hard rules but to create an advisory quality feedback loop that supports developers and reviewers in maintaining and improving code quality.

The **QA dashboard** should provide an overview of the current quality state of the *MECOIS* codebase. It should aggregate quality metrics from Continuous Integration (CI) builds, organizing them according to quality characteristics. The dashboard should present metrics in a visually structured format, enabling users to easily see the system's quality. It should update automatically, ensuring that feedback remains current without manual intervention.

The **QA PR comment bot** complements the dashboard by delivering feedback directly within GitHub PRs. It should analyze code changes submitted in a PR and compute the relevant metrics for the affected files or modules. Based on predefined quality metrics, the bot should post a structured comment summarizing the results. The bot should operate in a non-blocking manner: its feedback is informational and does not prevent merging. Users can manually trigger the bot through a special comment command.

Through this automated feedback loop, the dashboard provides a perspective on overall software quality, while the PR comment bot enables quality feedback during the review process. Together, they form a complementary system that support continuous quality improvement through metrics.

The following two sections define the functional and non-functional requirements derived from this solution concept. These requirements formalize what the QA dashboard and the QA PR comment bot must accomplish within the *MECOIS* development workflow.

3.4 Functional Requirements (FR)

The following functional requirements define the intended behavior of the QA feedback system within the *MECOIS* development environment.

3.4.1 FR of Dynamic QA Dashboard

The dashboard shall **integrate with the GitHub Actions CI pipeline** to automatically collect measurement data after each build on the main branch. This ensures that quality information is continuously updated.

The dashboard shall use ***MECOIS*-specific and meaningful metrics** that reflect the project's goals and quality characteristics. All metrics shall be based on standardized definitions derived from the SQuaRE framework to ensure consistency and comparability.

The dashboard shall **visualize collected metrics in a clear and structured way**. Results shall be optimized for developers and reviewers. It shall highlight metrics results through intuitive colors or symbols and format all output using GitHub-compatible Markdown.

The dashboard shall be **easily extensible** to accommodate new metrics or analysis tools. The dashboard's structure shall allow the addition of further measurement scripts or visual components without requiring significant modifications to the codebase.

3.4.2 FR of QA PR comment bot

The QA PR comment bot shall **operate within GitHub PRs and provide feedback on code changes**. It shall be fully integrated with the GitHub API and use repository data to analyze the modifications introduced in a PR.

The bot shall be **manually triggered** by a user command. This approach enables developers and reviewers to request quality feedback on demand, rather than having it executed automatically for every PR.

The bot shall **analyze code changes** and compute relevant *MECOIS*-specific metrics for the affected files or modules. It shall compare these results with the current baseline from the `main` branch to determine whether the changes positively or negatively affect software quality.

The bot shall generate a **structured advisory comment** summarizing its findings. The comment shall include key metrics, pass or warning indicators, and brief explanations formatted in GitHub-native Markdown.

The bot shall provide **clear and concise visual feedback** suitable for developers. Information shall be presented in a way that can be quickly interpreted during code review without requiring prior knowledge of the metrics.

The bot shall **automatically update** its comment when new commits are pushed to the same PR. This ensures that the feedback always reflects the latest state of the proposed code.

The bot shall operate in a **non-blocking mode**, offering advisory information without preventing merges. Its goal is to raise awareness of the quality, not to enforce automated quality gates.

The bot shall be **extensible and configurable**, allowing the addition of new metrics or rule sets in the future. Its design shall enable *MECOIS* maintainers to evolve the feedback logic as the project's quality goals expand.

3.5 Non-Functional Requirements

The non-functional requirements define the quality attributes that ensure the QA feedback system remains efficient within the *MECOIS* environment.

The system shall deliver feedback with **high performance** to support fast development workflows. Dashboard updates shall occur automatically immediately after each CI build, and the PR comment bot shall complete its analysis and post results within a few minutes of being triggered.

The system shall ensure **reliability and reproducibility** of results across builds and runs. All metrics shall be computed deterministically and guarantee comparable outcomes.

The system shall maintain **high usability** for developers and reviewers. The QA dashboard and QA PR comment shall present results in a clear, concise, and visually intuitive way.

The system shall be integrated seamlessly into the **GitHub-based** CI workflow used in *MECOIS*. All visualizations shall be rendered in GitHub-native Markdown to maintain consistency and avoid dependency on external services.

The system shall support maintainability through **modular design**. Each component shall be isolated to facilitate replacement when necessary.

The system shall maintain **availability and stability**. It shall recover from failed builds or unavailable analyzers without causing interruptions.

3.6 Measurement Approach

Finding fitting metrics that are helpful to developers and reviewers is a requirement that needs to be specified further. This section is dedicated to the development of quality metrics a QA feedback system for *MECOIS* should have.

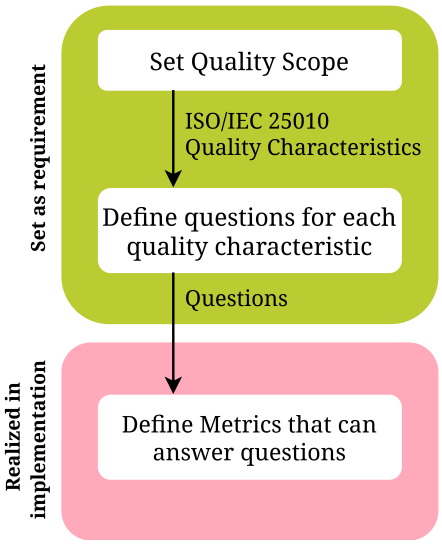

Figure 3.1: Overview of the measurement approach to derive metrics from abstract quality characteristics

The method integrates principles from international software quality standards and establishes a goal-driven quality model. The structure of how the requirements for suiting metrics are developed is visualized in Figure 3.1.

The ISO/IEC 25010 quality model defines the primary quality characteristics that serve as the conceptual basis for all metrics.

The GQM framework is used to organize the reasoning behind each measurement. Goals represent the intended quality outcomes, questions express what needs to be known to assess those outcomes, and metrics specify the type of information that can provide answers.

For each quality characteristic that is set to be part of the quality scope, questions are defined that need to be answered in order to assess if the quality characteristic is met. Based on these questions that serve as requirements, concrete metrics will be implemented in section 5.2 *Metric Collection*.

3.6.1 Quality Scope

This section defines the scope of the quality characteristics considered in this thesis. The ISO/IEC 25010 product quality model specifies nine main characteristics as described in section 2.2 *Standardized Quality Characteristics*: *functional suitability*, *performance efficiency*, *compatibility*, *interaction capability*, *reliability*, *security*, *maintainability*, *flexibility*, and *safety*. However, not all of these dimensions can be meaningfully assessed within the scope of this thesis and given the current state of the *MECOIS* project.

The quality scope was set to four characteristics: *maintainability*, *performance efficiency*, *security*, and *functional suitability*.

Maintainability is a key concern because *MECOIS* is developed collaboratively and evolves continuously. It reflects how easily the software can be maintained, modified, extended, and understood by contributors. In a context where many developers work on shared code, maintainability determines how efficiently knowledge can be transferred and how reliably quality can be sustained over time. This characteristic can also be examined effectively through static analysis and structural indicators, which integrate naturally with the CI pipeline.

Security is part of the scope because it remains an essential property of any modern software system, regardless of its development stage. Although *MECOIS* is still evolving, maintaining awareness of potential security issues helps reviewers to identify early warning signs. Monitoring this quality characteristic also promotes responsible development practices among contributors.

Functional suitability is included because it ensures that the implemented functionality behaves as intended and fulfills its defined purpose. For reviewers, this characteristic provides confidence that student contributions preserve the expected behavior of the system and that new functionality integrates correctly into the existing codebase.

Performance efficiency is considered relevant because it reflects the ability of *MECOIS* to perform its tasks effectively in relation to resource use. This characteristic indicates how well the system supports efficient development and testing processes.

The remaining ISO/IEC 25010 characteristics *compatibility, interaction capability, reliability, flexibility* and *safety* are not included in the current scope. These aspects often depend on system maturity, user interaction, operational data, or domain-specific safety requirements that are not yet applicable. Although these characteristics are important for a comprehensive product evaluation, they do not yet align with current objectives. These dimensions are interesting for future work but remain outside the scope of this thesis.

3.6.2 Mapping Quality Characteristics to Questions

The Goal-Question-Metric (GQM) approach provides a structured way to link quality characteristics to metrics. It was described in detail in section 2.3. In this work, GQM is applied to express how the selected ISO/IEC 25010 quality characteristics *maintainability*, *performance efficiency*, *security*, and *functional suitability* are realized with quality metrics. Each goal describes the intended quality outcome, while the related questions indicate what reviewers and developers must understand to judge whether that goal is achieved.

Maintainability

Maintainability serves as an indicator of how well a project can sustain continuous change without accumulating unnecessary complexity. It reflects the readability of the code, as well as the ease of testing and debugging. From a pedagogical perspective, maintainability feedback also supports learning by helping contributors understand how their implementation decisions influence the broader system. By observing maintainability, reviewers can detect hotspots and guide contributors toward more sustainable practices. Based on these considerations, the following questions have to be answered in order to judge the maintainability of *MECOIS*:

Questions:

- Is the documentation density adequate?
- Is the number of logical statements appropriate?
- How large is the code size?
- Is the number and scope of lint disables within agreed limits?
- Are new suppressions justified?
- How maintainable is the code?
- Is the share of low-grade files within acceptable limits?
- Where should refactoring be focused?
- How maintainable is the code in general?
- Which functions are too complex and should be refactored?
- Is the overall complexity distribution acceptable in critical areas?
- Are the most important parts of the system covered by tests?

Security Questions

Security represents a fundamental aspect of software quality. Although *MECOIS* is not publicly deployed, maintaining secure development practices helps prevent vulnerabilities from spreading into future releases. Student developers often rely on external packages and frameworks, and without oversight, insecure dependencies or configurations can accumulate. For reviewers, having visibility into security-related risks is helpful. Including security in the measurement approach has practical and educational functions. It protects the integrity of the project while fostering students' awareness of responsible dependency management. To guide the assessment of security as a quality goal, these questions are defined:

Questions:

- Is the direct dependency footprint justified?
- Are new dependencies justified and compliant?
- Is the dependency set minimal?

Functional Suitability Questions

Functional suitability ensures that the system continues to fulfill its intended purpose as new functionality is added. It describes how well the software's behavior matches its specified requirements and whether modifications preserve correctness. For a collaborative academic project, where students frequently integrate changes, functional suitability is essential. Reliable functional evidence, such as automated test outcomes, gives reviewers confidence that changes will not break existing features. To evaluate this functional suitability, the following guiding questions are applied:

Questions:

- Do code changes preserve the system's behavior?
- Are previously correct functionalities preserved after integration?
- Are automated tests sufficiently comprehensive to verify new or changed functionality?
- Does the system's observed behavior align with its intended requirements?

Performance efficiency Questions

Performance efficiency addresses how effectively a system performs its operations in relation to the resources consumed. When test execution or build pipelines become slow, developers are less likely to run them frequently, delaying quality assurance and integration. Tracking this characteristic provides insights into whether code or configuration changes degrade responsiveness and whether optimizations are needed to sustain a scalable and efficient workflow. The following questions guide the evaluation of performance efficiency:

Questions:

- Is the feedback loop timely and reliable?
- Are there regressions in responsiveness or resource consumption after changes?
- How stable are execution times across repeated builds?

In summary, the GQM model provides a structured foundation for linking the chosen quality characteristics to concrete evaluation questions that guide the selection of suitable metrics. The defined goals and questions establish what aspects of software quality must be observed, but do not yet specify how these observations are technically implemented. The concrete indicators and measurement mechanisms derived from these questions are introduced later in section 5.2 *Metric Collection* of chapter 5 *Design and Implementation*.

3.7 Summary and Prioritization

This chapter defined the requirements for an automated QA feedback system within the *MECOIS* development environment. It introduced the project context, the primary stakeholders, their respective needs, and translated these into functional and non-functional requirements. The required system consists of two main components: the QA dashboard, which visualizes quality information over time, and the QA PR comment bot, which provides advisory feedback directly in GitHub PRs. Both components share a standardized measurement approach grounded in the International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) 25010 quality model and structured through the Goal-Question-Metric (GQM) method. The selected quality characteristics *maintainability*, *performance efficiency*, *security*, and *functional suitability* represent the most relevant dimensions of quality for a collaborative software project.

The stated requirements are prioritized using the *MoSCoW* method (**M**ust-have, **S**hould-have, **C**ould-have, and **W**on't-have).

MoSCoW Prioritization

Must-have

These requirements are essential for the success of this project and represent the minimum viable functionality of the QA feedback system. They include:

- A working QA dashboard that visualizes metrics dynamically in Markdown.
- A fully functional PR comment bot that can analyze PRs, generate structured comments, and integrate with GitHub Actions.
- The use of metrics that specifically fit *MECOIS*.

Should-have

These requirements significantly enhance the system's usefulness but are not strictly necessary for initial functionality. They include:

- A shared and extensible measurement foundation that allows both components to use the same metrics.
- Metrics that directly support review decisions, such as metrics of maintainability and functional suitability.
- Full integration into GitHub by using GitHub-based CI workflow and GitHub-supported Markdown files.

Could-have

These features would provide additional analytical depth or user comfort if time and resources permit. They include:

- Performance testing.
- Historical comparisons of metric evolution across releases.
- Advanced, extensive visualization dashboards beyond the prototype level.

Won't-have (for now)

These requirements are out of scope for this thesis, but may be valuable for future iterations of *MECOIS*. They include:

- Support for additional quality characteristics such as compatibility, flexibility, or safety.
- Automated enforcement mechanisms or blocking checks within CI pipelines.
- Configurability for selecting or weighting metrics per project or branch.

In Figure 3.2, the *MoSCoW* prioritization is visualized as a checklist. A filled-in version of the same checklist will be presented in chapter 6 *Evaluation*.

The requirements established a clear understanding of the system's scope and priorities. The central objective is to deliver an operational QA feedback system that integrates into the *MECOIS* CI workflow and provides meaningful insights.

The next chapter will describe how these requirements are realized in the QA feedback system's architecture.

Figure 3.2: *MoSCoW* checklist of the most important requirements an implementation of a *MECOIS*-specific QA feedback system should have.

4 Architecture

The architecture of the QA feedback system defines how the requirements identified in chapter 3 are realized. It describes how the system's components interact to collect and present quality metrics within the *MECOIS* development environment. Because the solution consists of two main parts, *the QA dashboard and the QA PR comment bot*, the architecture focuses on how both of their components operate independently and which components are shared between them. This chapter also explains architectural decisions that had to be taken and how they support extensibility.

The architecture is described from two complementary perspectives:

1. The **System Context View** illustrates the environment in which the QA feedback system operates as a broad overview of all participating actors.
2. The **Component View** breaks down the system into building blocks, showing data flow and communication for both the QA dashboard and the QA PR comment bot.

Together, the views illustrate how the feedback system is structured to enable consistent quality evaluation and future extension.

4.1 System Context

The QA feedback system operates within the *MECOIS* development environment, interacting with developers, the Git-based repository platform, and the CI platform. The system context diagram in Figure 4.1 shows how developers interact with the Git platform, how the CI system processes these events, and how results are returned as feedback through the QA dashboard and the QA PR comment bot.

Developers push commits to the `main` repository and open PRs for review. During a review, a developer or reviewer can request PR-specific analysis by adding a trigger phrase as a PR comment.

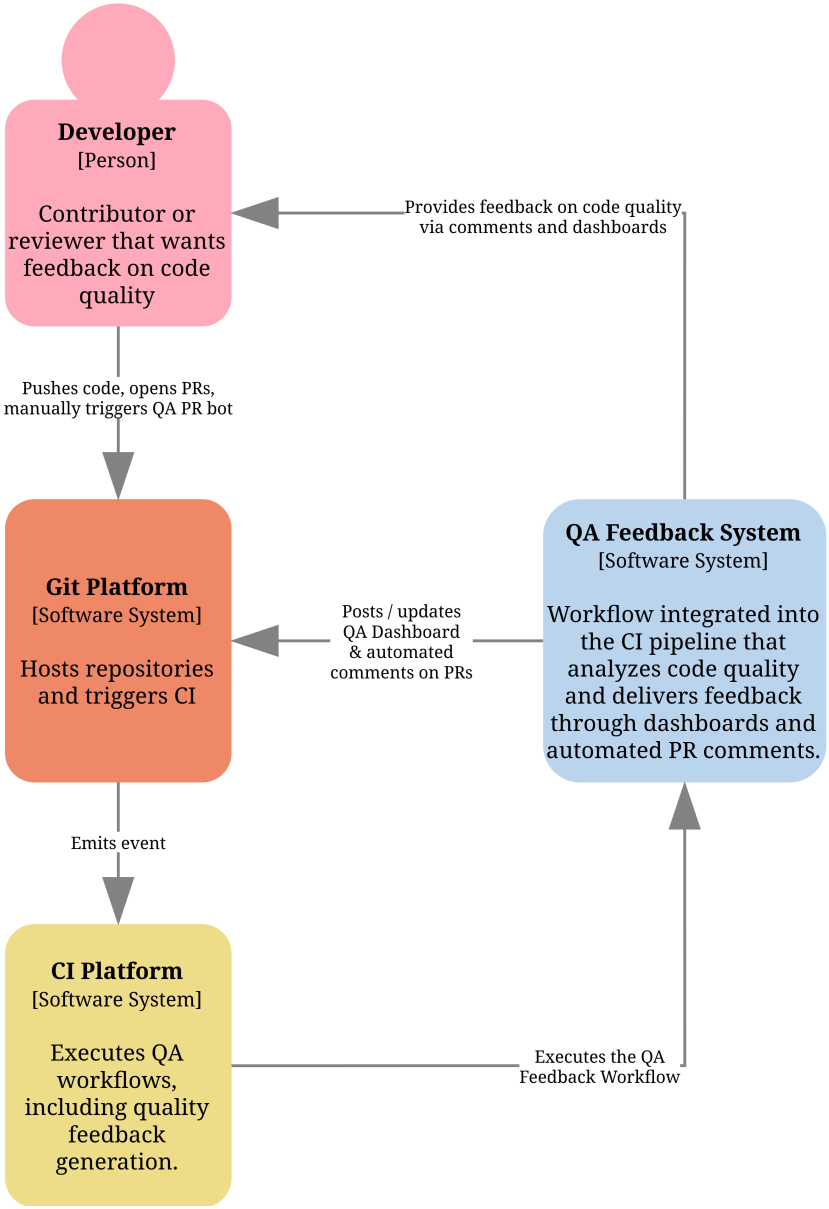

Figure 4.1: System context diagram of the QA Feedback System

The Git repository platform receives these pushes and comments. It then emits two kinds of events that trigger the CI platform (GitHub Actions):

- A push to the `main` branch starts the standard CI pipeline that updates the dashboard.
- A PR comment containing the trigger phrase starts a workflow that runs the PR comment bot.

The QA dashboard reflects the current quality state of the `main` repository and is updated automatically after successful pushes to `main`. The PR comment bot posts an advisory comment directly into the PR only when the trigger phrase is used, highlighting the quality impact of the proposed changes.

Developers submit or review code, the CI platform performs analysis, and the QA feedback system visualizes the results directly within GitHub.

4.2 Component View

Architecturally, the QA feedback system is decomposed into six components, both for the QA dashboard and the QA PR comment bot.

- The **Configuration** component provides rules for which metrics to collect and where and how they should be displayed. In the case of the PR comment bot, where quality change is in focus, the configuration specifies how the metrics should be compared to the baseline to derive the final metrics.
- The **Metric Collectors** component describes all modules that collect metrics. They collect metrics from external sources or compute them.
- The **Orchestrator** component calls the Metric Collectors modules and returns all metrics as specified in the configuration.
- The **Template Data Builder** component renders metrics into a visualization type specified in the configuration.
- The **Renderer** component fills a template with metrics rendered from the Template Data Builder to produce the final Markdown.
- The **Workflow** component runs all important steps and is executed by the CI platform.

In addition to the component of the QA feedback system itself, the system interacts with three external components:

- **Developers** who interact with the system by performing actions that may trigger the execution of the QA feedback.
- The **Git Platform** that hosts the repository for which the QA feedback is provided. The Git Platform also emits the exact events that trigger the QA feedback system. The Git Platform also hosts the QA Dashboard and its metrics, which serve as a baseline for the QA PR comment bot.
- The **CI platform** executes the QA workflows.

The following two sections show the component view as seen from the QA dashboard and the QA PR comment bot.

4.2.1 Component View of QA Dashboard

This section will describe how all components work together when the QA dashboard is generated. All components and their interaction are visualized in the component diagram in Figure 4.2.

When a **developer** pushes to `main`, the **CI platform** is triggered through the **Git platform**. The **CI platform** then executes the **Dashboard Workflow**, which performs the following steps:

1. It requests metrics from the **Baseline Orchestrator**. This **Orchestrator** asks the **configuration** which metrics to collect and where the corresponding **Metric Collectors** are to collect them. Finally, the **Baseline Orchestrator** returns all metrics that are needed for the dashboard in a structured format.
2. From the **Dashboard Renderer**, it requests a rendered dashboard containing the metrics it just obtained. The **Dashboard Renderer** first sends the metrics to the **Template Data Builder**, which renders the metrics separately in a visualization type specified in the **configuration**. This could be a list, a badge, a diagram, or another visualization style. The **Dashboard Renderer** then fills a dashboard template with the rendered metrics and returns it to the **Dashboard Workflow**.
3. It then sends the metrics and the rendered dashboard to the **Git Platform**, where an orphan branch is used as storage. On this branch, the **developer** can view the dashboard and get feedback on the current code quality.

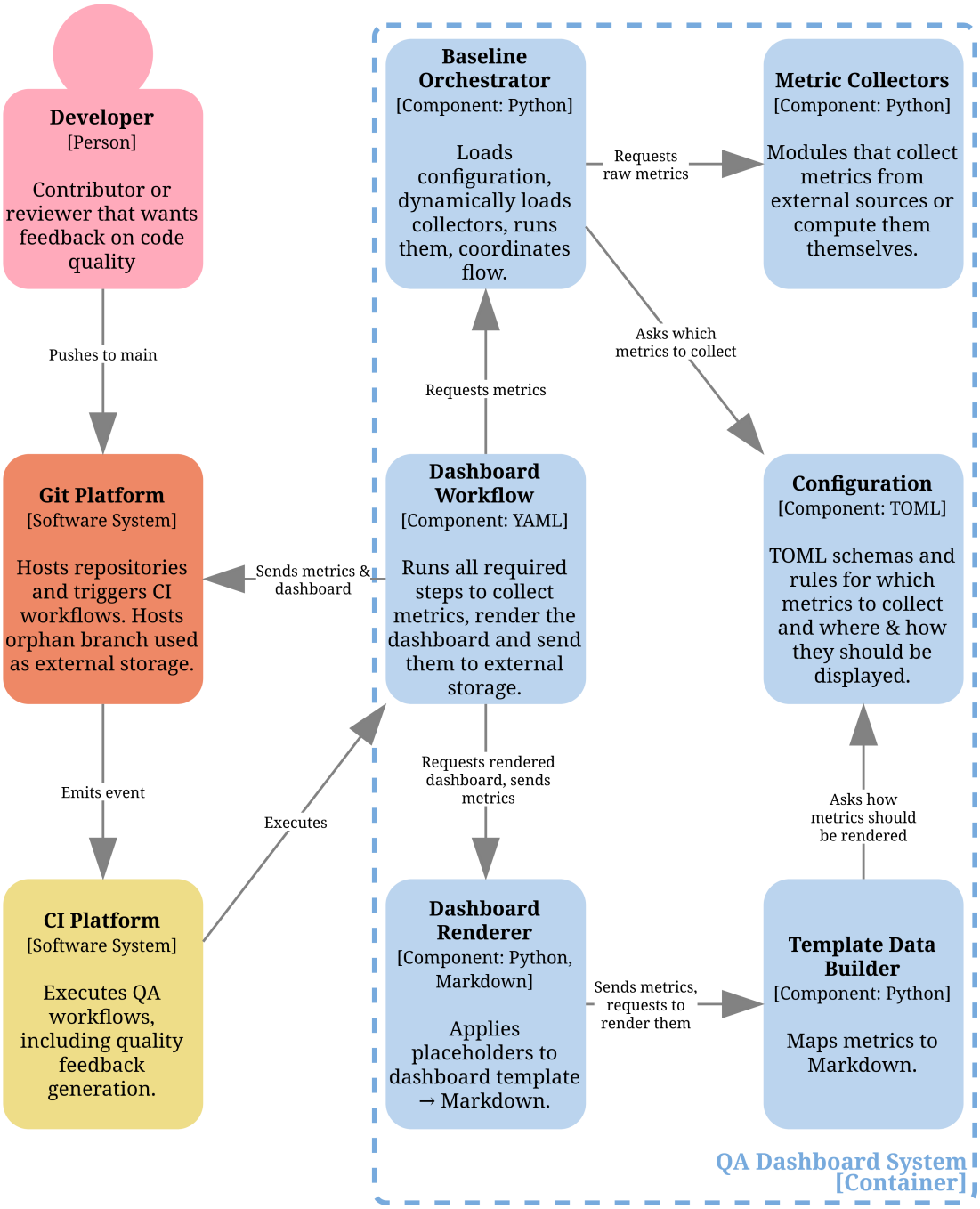

Figure 4.2: Component diagram of the QA dashboard system.

4.2.2 QA PR Comment View

The components for creating the QA PR comment work similarly to those in the QA dashboard case above. But here, two additional steps are required: the metrics must be compared to a baseline first, and the resulting Markdown must be posted as a comment in a PR. The baseline values against which the code in the PR is compared are the metrics from a QA dashboard generation run. In the following, these metrics will be referred to as the baseline. This also explains why at least one prior run of the QA dashboard workflow is required for the QA PR comment bot to work.

All components and their interaction are visualized in the component diagram in Figure 4.3.

When a **developer** comments a specific trigger word in a PR, the **CI platform** is triggered through the **Git platform**. The **CI platform** then executes the **PR Comment Workflow**, which performs the following steps:

1. It requests metrics that capture the current quality of the `main` branch from the **Git Platform**. These metrics were stored during the creation of the QA Dashboard in a separate orphan branch.
2. It then requests the generation of metrics for the PR code and their comparison with the baseline metrics from the **Comparison Orchestrator**. This **Orchestrator** asks the **configuration** which metrics to collect and where the corresponding **Metric Collectors** are to collect them. It also asks which comparison method should be used to compare these two sets of metrics. Finally, the **Comparison Orchestrator** returns all metrics needed for the PR comment in a structured format.
3. From the **Comment Renderer & Publisher**, it requests a rendered comment containing the metrics it just obtained. The **Comment Renderer & Publisher** first sends the metrics to the **Template Data Builder**, which renders the metrics separately in a visualization type specified in the **configuration**. This could be a list, a badge, a diagram, or another visualization style. The **Comment Renderer & Publisher** then fills a comment template with the rendered metrics and publishes or updates it in the PR from where the **workflow** was triggered.

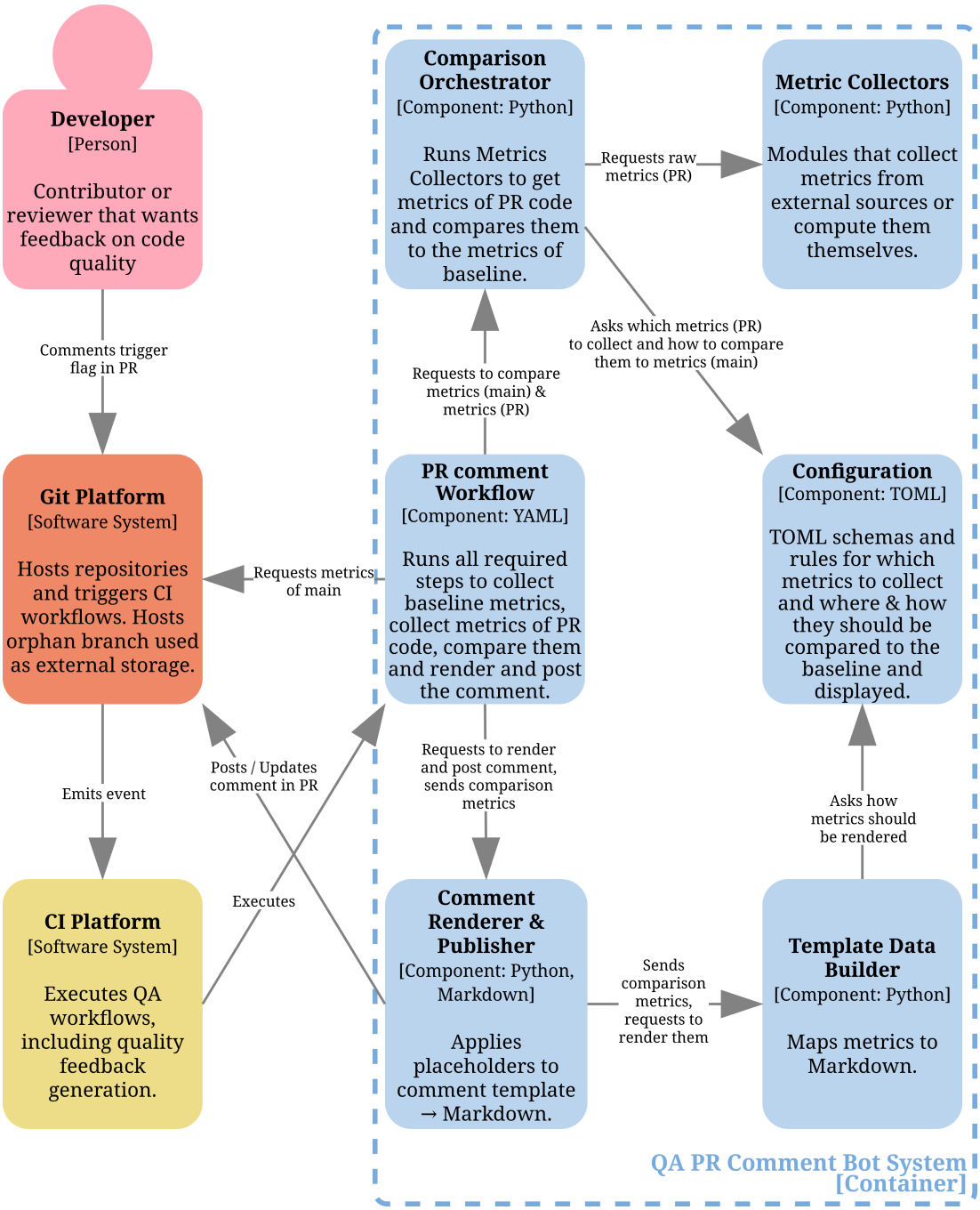

Figure 4.3: Component diagram of the QA PR comment bot system.

In summary, the QA feedback system is built from six main components: configuration, metric collectors, orchestrator, template data builder, renderer, and workflow. Each has a clear role and interacts with the others through the CI and Git platforms. The configuration defines what to measure, the collectors gather data, and the orchestrator controls the process. The template data builder prepares the results for display, the renderer produces the final Markdown output, and the workflow executes everything in the CI pipeline. Together, these components form a simple structure for providing automated quality feedback. The following section explains all the key architectural decisions and trade-offs behind this design.

4.3 Architectural Decisions and Trade-offs

This section summarizes the key architectural decisions behind the QA dashboard and QA PR comment bot and discusses the trade-offs that shaped them.

Early in the project, publishing the dashboard directly on the `main` branch looked like a good solution. While attractive for visibility, it proved inefficient in practice. Every push to `main` triggers a dashboard refresh, which in turn produces a commit that all contributors then have to pull, even though the change was only the generated Markdown file rather than source code. To avoid this problem, the dashboard is published to a dedicated `stats` branch instead, keeping generated artifacts separate from the development history. The trade-off is a slight reduction in discoverability, which is overcome by linking to the dashboard from project documentation and PR comments.

As storage for both the rendered dashboard and the baseline metrics, the Git orphan branch `stats` was chosen rather than external services. Keeping everything in GitHub provides built-in versioning and access control. The downside is that working with an orphan branch can be slightly inconvenient, as GitHub continues to display the difference in commits between the `main` branch and the orphan branch. Since for this system all artifacts are small text files, the simplicity and traceability outweigh the minor inconvenience.

The dashboard workflow runs on pushes to `main` because the dashboard is intended to represent the quality state of the codebase. Generating per-branch dashboards would be incompatible with the choice of an orphan branch. In the orphan branch, the dashboard is set as `README.md`. Several dashboards for different branches would be less straightforward. The QA PR comment workflow listens to PR events but is triggered by an explicit flag in the PR body. This gives reviewers control over when a quality snapshot is worthwhile.

The PR comment depends on a prior dashboard run because it must compare the PR's metrics to a baseline that reflects the latest main. Without that baseline file in the `stats` branch, any difference would be undefined. This dependency introduces a one-time initialization requirement of the dashboard. Fixing the comparison to the `main` branch also has the consequence that deltas may shift if the `main` branch advances while a PR is open. This was not seen as a problem for this implementation but might be relevant in the future.

With the architectural structure now established, the next chapter presents the system's design and implementation.

5 Design and Implementation

This chapter translates the architecture into concrete design choices and their code realization. It is organized to mirror the component structure introduced earlier in section 4.2 *Component View* of chapter 4 *Architecture*.

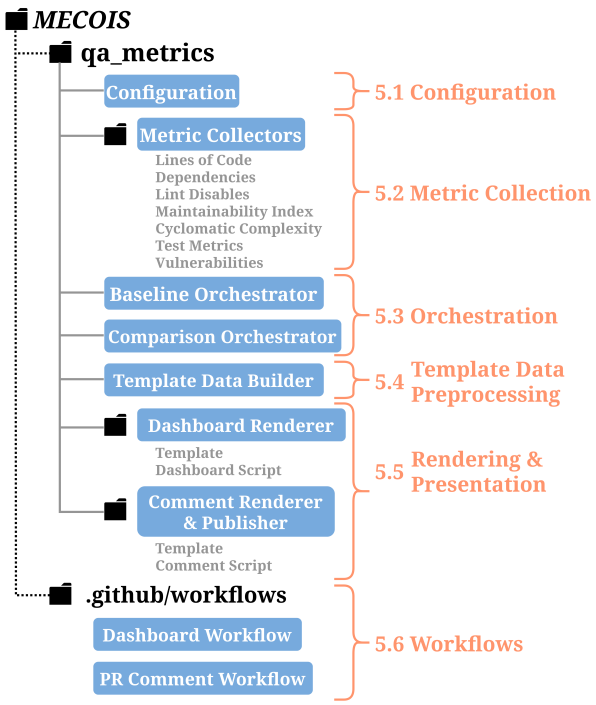

Figure 5.1: Mapping of architectural components to their implementation sections and corresponding folders, showing how each component from the architecture is realized within the code structure.

Each section focuses on the implementation details of a component or a pair of components when the implementation of the QA dashboard and the QA PR comment bot differs. Figure 5.1 shows how the components and their respective implementations are structured in the project folder and to which sections they are mapped in this chapter. In section 5.1 *Configuration* the configuration is shown. This component defines which metrics are collected, how they are visualized, and how they are compared to the baseline for the PR comment generation. The *Metric Collectors* gather raw data from the repository, providing the quantitative foundation for all metrics. The Metric Collectors and specifically their underlying metrics are presented in section 5.2 *Metric Collection*. Each metric is explained as part of the quality characteristics and GQM approach.

The *Orchestrator* coordinates these Metrics collectors: in the dashboard, the *Baseline Orchestrator* compiles the current state of quality metrics, while in the comment bot, the *Comparison Orchestrator* compares PR results to the stored baseline. They are presented in section 5.3 *Orchestration*. The *Template Data Builder* prepares the collected metrics for display by rendering them in a configured visualization type, such as lists, badges, or diagrams. This component is presented in section 5.4 *Template Data Preprocessing*. The *Renderer* then generates the final Markdown output: the *Dashboard Renderer* produces the static quality overview, and the *Comment Renderer & Publisher* creates and posts the PR comment. This component also includes the Markdown templates themselves. The renderer modules and the templates, along with their rendered counterparts, are presented in section 5.5 *Rendering and Presentation*. Finally, the two *workflows* are presented in section 5.6 *Workflows*.

5.1 Configuration

The design is anchored in an explicit contract. `metrics_config.toml` is the single source of truth that declares what is measured and how results are rendered and compared. It is read by the QA dashboard's and QA PR comment bot's orchestrators, as well as the rendering pipeline, so a change here propagates through metric collection and presentation without requiring code edits elsewhere.

The configuration is made up of the following parts:

1. Declaring collectors for the dashboard

Entries under `[[load-dashboard-metrics]]` list the concrete metric functions to run when producing the baseline `metrics.json`. Each entry specifies:

- `path` (module file), `function` (callable to execute), and optional `args`
- `output-structure`, which names the keys that will be written into `metrics.json`. It can be a single name (e.g., `vulnerabilities`) or a tuple (e.g., `(loc, lloc, sloc, commentl, blankl)`). At runtime, the orchestrator imports the function and validates that the arity and order of the returned tuple match the names listed here.

This section, therefore, defines the baseline data model: the exact keys that downstream components (renderers and comparisons) rely on.

2. Mapping baseline values to dashboard widgets

The `[[dashboard-metric]]` section tells the renderer how each baseline key should appear in the dashboard template:

- `id` references a key that must exist in `metrics.json`
- `display-type` chooses one of three render modes:
  - `badge` (compact "key–value" with optional `suffix` and `colors` rule)
  - `list` (bulleted lines)
  - `other` (delegate to a custom renderer)
- For `badge`, `display-name` sets the left label text, `suffix` appends a unit (e.g., `s`, `%`), and `colors` is either a literal color or a reference to a rule defined in `[[colorrule.*]]`.
- For `other`, `render-path`, `render-function`, and optional `render-args` declare a function that receives the metric value (plus parsed arguments) and returns Markdown.

This section thus defines a presentation contract: every `id` maps to a placeholder in `dashboard-template.md`, and the renderer is told exactly how to turn raw values into Markdown fragments.

3. Declaring collectors for PR comparison

`[[load-comment-metrics]]` mirrors the dashboard collector list but is evaluated on the PR's code to build the *new* values used to compare the PR's code to the baseline. Keys emitted here are suffixed with *_new* (e.g., `loc_new`, `mi_overview_new`) to separate them from baseline keys. As with the dashboard, the tuple in `output-structure` must match the function's return shape.

4. Defining comparison logic and PR comment rendering

The `[[comment-metric]]` section instructs the comparison orchestrator how to compute and present each item in `delta.json` (the output of the orchestrator):

- `id` names the derived metric that will be written to `delta.json` and used as a placeholder in `comment-template.md`.
- `comparison-type` selects one of:
  - `num_diff` → compute a signed numeric difference between `old-value` and `new-value` (rounded, with `+`/`-`)
  - `list_diff` → elements present in the new list but not in the old
  - `only_new` → pass through a new value unchanged (used when the baseline's value is not relevant, e.g., test results)
  - `other` → call a custom comparator given by `comparison-path` and `comparison-function` (with optional `comparison-args`)

- A second set of fields (`display-type`, and either badge attributes or `render-*` for other) tells the renderer how to turn that derived value into Markdown for the PR comment. The configuration also uses `display-type = "none"` for the internal intermediates (`tests_failed_new`, `failed_tests_new`) that are needed as arguments but should not render directly and be displayed.

5. Color rules as a reusable style system

Blocks under `[[colorrule.*]]` define named palettes that the renderer can apply to badges. The builder supports two patterns:

- `type = "solid"` → always use the given `color` (e.g., `normal_badge` is set to always be blue),
- `type = "negzeropos"` → choose among `neg`, `zero`, `pos` by inspecting the sign of a computed value (e.g., `posneg` colors improvements green and regressions red, while `posneg_inverse` flips that for metrics where a lower value is better).
- By referencing a rule name in `colors`, the dashboard and comment badges remain consistent without hard-coding color choices.

6. Example usage

Putting these sections together in an example: a dashboard entry for **LOC** declares where to collect it and how to display it as badges; the orchestrator writes `loc`, `lloc`, `sloc`, `commentl`, and `blankl` into `metrics.json`; the template builder reads those keys, applies the `normal_badge` rule and `suffix`, and fills the placeholders in `dashboard-template.md`. For PRs, the corresponding `_new` values are collected, the `num_diff` rules compute signed deltas (e.g., `23`), and the comment renderer formats them as badges using `posneg` color logic. More complex metrics, such as vulnerabilities or the maintainability index, usually rely on `other` hooks to render tables and difference lists via helper modules.

Because `metrics_config.toml` names every function, key, and renderer, the system is configuration-driven: adding a metric or a new rendered section requires no orchestrator changes, only a collector module (with optional helper functions) plus declarations here.

5.2 Metric Collection

This section shows the measurement approach by specifying each metric collector used in the QA feedback system. For the collectors, their resulting metrics are mapped to the thesis requirements via quality characteristics and the GQM goals

and questions they address. This approach was introduced in section 3.6 *Measurement Approach* of chapter 3 *Requirements*.

5.2.1 Lines of Code

The LOC metrics show the size and structural composition of the codebase. It serves the quality characteristic *Maintainability* and addresses the GQM concerns: *Is documentation density adequate? Is the number of logical statements appropriate? How large is the code size?*

The collector reports totals with the sub-counts (Lines of Code, Logical lines of code, source lines of code, comment lines, blank lines). The Baseline keys are `loc`, `lloc`, `sloc`, `commentl`, and `blankl` in `metrics.json`. Runs of the QA PR comment bot pipeline emit the corresponding `_new` keys, which are turned into signed deltas `(*_diff)` for review. In both the dashboard and PR comment, these metrics appear as badges with a collapsible details block. A summary of this Metric Collector can be found in Table 5.1.

Metric Collector	Quality Characteristic	Question (GQM)	Metrics	Dashboard Perspective (Baseline)	Comment Perspective
`get_loc.py`	Maintainability	Is documentation density adequate?	**Lines of Code** Total lines	Absolute count	Change vs baseline
			Logical lines of code Logical code statements	Absolute count	Change vs baseline
		Is the number of logical statements appropriate?	**Source lines of code** Non-blank, non-comment code lines (physical)	Absolute count	Change vs baseline
		How large is the code size?	**Comment lines** Documentation-only lines	Absolute count	Change vs baseline
			Blank lines	Absolute count	Change vs baseline

Table 5.1: Overview of *Lines of Code (LOC)* metrics, related quality characteristics, GQM questions, and perspectives in dashboard and PR comment contexts.

LOC-family metrics provide an immediate orientation signal for reviewers: they quantify how much code was touched and whether code growth is accompanied by documentation. This directly supports Maintainability by making the magnitude of changes and documentation density visible.

The Metric collector executes `radon raw . -s` in the repository root and parses the `** Total **` summary block. Regular expressions then extract integer totals for lines of Code, logical lines of code, source lines of code, comment lines, and blank lines.

Rising LOC is not inherently negative, but ongoing growth without a corresponding increase in comments or tests can signal accumulating debt. LOC is not a useful metric in isolation. It should be interpreted alongside Maintainability Index (MI), Cyclomatic Complexity (CC), and test results.

5.2.2 Lint Disables

This metric shows globally suppressed lint checks defined in the repository's configuration, highlighting potential blind spots in static analysis. It supports Maintainability and addresses the GQM questions: *Is the number and scope of lint disables within agreed limits? Are new suppressions justified?*

An overview of this metric is provided in Table 5.2.

Metric Collector	Quality Characteristic	Question (GQM)	Metrics	Dashboard Perspective	Comment Perspective
`get_lint_disables.py`	Maintainability	Is the number and scope of lint disables within agreed limits? Are new suppressions justified?	**Lint Disables** Globally suppressed lint rules	Current list	Change vs baseline (new disables)

Table 5.2: Overview of the *lint disables* metric, related quality characteristics, GQM questions, and perspectives in dashboard and PR comment contexts.

The Metric collector opens `pyproject.toml`, scans for the start of a disable block, and collects double-quoted entries until the closing bracket. A regex extracts the rule code and an optional trailing comment, which is preserved to provide reviewer context. The function returns a list of strings such as `<RULE_CODE> <optional comment>`.

The baseline key is `all_lint_disables` in `metrics.json`. Runs of the QA PR comment bot pipeline produce `all_lint_disables_new`, with `all_lint_disables_diff` computed via `list_diff` to show only newly added suppressions.

5.2.3 Maintainability Index (file-level)

The Maintainability Index (MI) metric identifies file-level maintainability hotspots and regressions. It serves the Maintainability characteristic and answers the GQM questions: *How maintainable is the code? Is the share of low-grade files within acceptable limits? Where should refactoring be focused?*

The collector analyzes Python source files and returns a mapping of path → rank. Baseline values are stored under `mi_overview` (dict) in `metrics.json`. For PRs, `_new` values are compared to the baseline to produce `mi_worse`, `mi_better`, and `mi_new_section` via helper functions. The dashboard renders an overview section. For the PR comments, the metric renders to focused lists of regressions, improvements, and newly added files. In Table 5.3, the metric is summarized.

Metric Collector	Quality Characteristic	Question (GQM)	Metrics	Dashboard Perspective	Comment Perspective
`get_mi.py`	Maintainability	How maintainable is the code? Is the share of low-grade files within acceptable limits? Where should refactoring be focused?	**Maintainability Index** File-level grade (A–C)	Grade distribution	Change vs baseline (worse/better files) & Grade distribution (new files)

Table 5.3: Overview of the *Maintainability Index (MI)* metric, related quality characteristics, GQM questions, and perspectives in dashboard and PR comment contexts.

File-level maintainability aligns with reviewer needs to spot hotspots quickly and to prioritize refactoring where it will have the most impact.

The collector invokes `radon mi . -j` at the repository root and parses the resulting `JSON`. For each file path, it extracts the rank field and builds a dictionary `{ path: rank }`. The MI grade is ranked from **A** *(Very high maintainability)* to **B** *(Medium maintainability)* to **C** *(Extremely low maintainability)* as seen in Table 5.4.

Rank	Meaning
A	Very high
B	Medium
C	Extremely low

Table 5.4: Maintainability Index (MI) ranks and corresponding risk levels, adapted from the Radon documentation.

MI is a heuristic that combines structural properties. However, it can be noisy for very small or very large files and does not directly capture readability or domain complexity. Regressions (e.g., A→B or B→C) warrant attention, especially in files that are central to the system. MI should be interpreted alongside Cyclomatic Complexity (CC) and test evidence to avoid single-metric decisions.

5.2.4 Cyclomatic Complexity (function-level)

The Cyclomatic Complexity (CC) metric grades the control-flow complexity of individual functions and methods. It serves the quality characteristic Maintainability and answers the GQM questions: *How maintainable is the code? Which functions are too complex and should be refactored? Is overall complexity distribution acceptable in critical areas?* A summary of this metric is visualized in Table 5.5.

Metric Collector	Quality Characteristic	Question (GQM)	Metrics	Dashboard Perspective	Comment Perspective
`get_cc.py`	Maintainability	How maintainable is the code? Which functions are too complex and should be refactored? Is overall complexity distribution acceptable in critical areas?	**Cyclomatic Complexity** Control-flow complexity grade (A–F)	Grade distribution	Change vs baseline (worse/better files) & Grade distribution (new functions)

Table 5.5: Overview of the *Cyclomatic Complexity (CC)* metric, related quality characteristics, GQM questions, and perspectives in dashboard and PR comment contexts.

For PRs, `_new` values are compared to the baseline to produce `cc_worse`, `cc_better`, and `cc_new_section` via helper functions. The dashboard displays an overview section that shows the rank distribution for all functions. In PR comments, the metric is displayed as lists of worsened and improved functions, and a rank distribution for new functions. The collector invokes `radon cc . -j` and parses the `JSON` output. Entries of type function or method are retained and flattened into qualified names using the pattern `filename > [Classname.]functionname`.

The CC is ranked from **A** *(lowest complexity)* to **F** *(highest complexity)* as seen in Table 5.6.

A downgrade (e.g., B→D) signals increased potential defect risk where refactoring should be considered. CC should be considered alongside the Maintainability Index (MI) and test results.

Rank	Meaning
A	low - simple block
B	low - well structured and stable block
C	moderate - slightly complex block
D	more than moderate - more complex block
E	high - complex block, alarming
F	very high - error-prone, unstable block

Table 5.6: Cyclomatic Complexity (CC) ranks and corresponding risk levels, adapted from the Radon documentation.

5.2.5 Test Metrics

This metric collector collects the metrics *test pass/fail counts*, *total duration*, and *overall coverage*. These metrics support different quality characteristics and questions:

The `tests pass vs fail rate` supports the quality characteristic *Functional suitability* and answers the GQM question *Do code changes preserve behavior?*

The `code coverage` supports the quality characteristic *Maintainability* and answers the GQM question *Are the most important parts of the system covered by tests?*

The `test duration` supports the quality characteristic *Performance efficiency* and answers the GQM question *Is the feedback loop timely and reliable?*

A summary of these three metrics is presented in Table 5.7.

Metric Collector	Quality Characteristic	Question (GQM)	Metrics	Dashboard Perspective	Comment Perspective
`get_test_metrics.py`	Functional suitability	Do code changes preserve behavior?	**Tests – pass/fail**	Pass rate & failed tests	Pass rate & failed tests
	Maintainability	Are the most important parts of the system covered by tests?	**Code coverage**	Percentage	Change vs baseline (pp)
	Performance efficiency	Is the feedback loop timely and reliable?	**Test duration**	Absolute duration	Change vs baseline

Table 5.7: Overview of *test* metrics, related quality characteristics, GQM questions, and perspectives in dashboard and PR comment contexts.

The metric collector performs a whole-repository test run with coverage and returns `tests_passed`, `tests_failed`, `test_duration`, `code_coverage`, and `failed_tests`). Baseline values are written to `metrics.json`.

PR comment bot runs produce the corresponding `_new` values; deltas include `test_duration_diff` (with inverse coloring, where a lower score is better) and `test_coverage_diff`. The dashboard renders a tests passed summary badge with optional locations of the failed tests, and badges for test duration and coverage. The PR comment renders the new passed tests summary and delta badges for duration and coverage.

Automated tests are the primary evidence for Functional Suitability. They check if the system continues to perform its intended functions as changes are integrated. Total duration is an indicator of Performance Efficiency within CI, affecting the feedback loop's responsiveness.

The collector runs the project's test suite with coverage enabled and requests machine-readable reports. Non-success exit codes that still indicate "tests ran but some failed" are treated as valid measurement outcomes.

5.2.6 Dependencies

This metric makes the third-party footprint explicit by listing installed Python packages. It supports the quality characteristic Security and addresses the GQM questions: *Is the direct dependency footprint justified? Is the dependency set minimal? Are new dependencies justified and compliant?* A summary of this metric is presented in Table 5.8.

Metric Collector	Quality Characteristic	Question (GQM)	Metrics	Dashboard Perspective (Baseline)	Comment Perspective
`get_deps.py`	Security	Is the direct dependency footprint justified? Is the dependency set minimal? Are new dependencies justified and compliant?	**Dependencies** Third-party packages and versions	Current set & count	Change vs baseline (added/updated/removed)

Table 5.8: Overview of the *dependencies* metric, related quality characteristics, GQM questions, and perspectives in dashboard and PR comment contexts.

The collector inspects the repository environment (as seen by `pip freeze`), normalizes package names, and returns a dictionary of `name → version` together with the total number of packages. Baseline keys are `dependencies` (dict) and `dependencies_len` (int) in `metrics.json`. For PRs, `_new` keys are produced and compared to the baseline to render new and updated dependency lists in the PR comment. The dashboard displays the complete list in a collapsible format along with a count badge.

5.2.7 Vulnerabilities

Making vulnerability information visible aligns directly with the Security scope and answers the GQM question *Are there unresolved High/Critical vulnerabilities?* A summary of this metric is presented in Table 5.9.

Metric Collector	Quality Characteristic	Question (GQM)	Metrics	Dashboard Perspective	Comment Perspective
`get_vulns.py`	Security	Are there unresolved High/Critical vulnerabilities?	**Vulnerabilities**	Current findings	—

Table 5.9: Overview of the *vulnerabilities* metric, related quality characteristics, GQM questions, and perspectives in dashboard and PR comment contexts.

The collector audits the project's declared dependencies and requests a machine-readable vulnerability report with the tool *pip-audit*. The resulting `JSON` is filtered to include only entries with findings, producing a dictionary of `{package, version, vulnerabilities}` records.

5.2.8 Overview of used metrics

Together, the chosen collectors provide an overview of quality across the project's most relevant quality characteristics as seen in Table 5.10. **Maintainability** is covered by *LOC measures*, *lint disables*, *MI*, *CC*, and *code coverage*. **Security** is addressed through the *dependencies* inventory and a *vulnerabilities* audit. **Functional suitability** is covered by the *test pass rate*. **Performance efficiency** is addressed by *test duration*.

Some quality characteristics are more extensively covered than others.

How and if the chosen metrics satisfy the set quality scope is addressed in chapter 6 *Evaluation*.

Maintainability	Security	Functional Suitability	Performance Efficiency
Lines-of-code Measures	**Dependencies**	**Tests – pass/fail**	**Test duration**
Lint Disables	**Vulnerabilities**		
Maintainability Index			
Cyclomatic Complexity			
Code coverage			

Table 5.10: Mapping of all implemented metrics to their corresponding quality characteristics.

5.3 Orchestration

Orchestration in this system serves as the glue between configuration and metric collectors. The Baseline (Dashboard) Orchestrator (`create_json_all_metrics.py`) produces the baseline snapshot `metrics.json`. The PR Comparison Orchestrator (`create_json_compare_metrics.py`) produces the reviewer-focused `delta.json`. Both orchestrators are executed by their respective workflows.

The configuration declares what to run and how results map to metric IDs. The orchestrators execute the referenced functions and assemble the outputs into dictionaries that the rendering components rely on.

5.3.1 Baseline Orchestrator

The baseline orchestrator (`create_json_all_metrics.py`) executes the collector set declared in `[[load-dashboard-metrics]]` and outputs the baseline snapshot `metrics.json`. This artifact is the single source of truth for rendering the dashboard and the baseline for later comparisons during PR evaluation.

The orchestrator is entirely driven by `metrics_config.toml`. At runtime, the program loads the `TOML` configuration and iterates the `load-dashboard-metrics` list. For each entry, it loads the callable, executes it (with or without arguments), and validates that the returned value's arity matches the output structure. Results are accumulated into a dictionary under the exact IDs defined by the configuration. This dictionary, in `JSON` format, is used as `metrics.json`. Values may be numbers, strings, lists, or dictionaries, depending on the collector.

Adding a new metric does not require changes to the orchestrator itself.

The QA Dashboard GitHub Actions workflow invokes this script to generate the baseline. The resulting `metrics.json` is committed to the `branch` alongside the rendered dashboard, where it serves as the baseline later used by the PR comparison orchestrator.

5.3.2 PR comparison orchestrator

The PR comparison orchestrator recomputes the configured metrics on the PR code, compares them to the baseline, and outputs `delta.json` used by the PR comment generator. Its purpose is to translate raw measurements into change metrics that highlight what improved, worsened, or was newly introduced.

The orchestrator accepts a path to the baseline `metrics.json` (checked out from the `stats` branch) and is driven by `metrics_config.toml`. Two configuration blocks govern its behavior:

1. `[[load-comment-metrics]]` declares which collectors to rerun on the PR code (with `path`, `function`, optional `args`, and `output-structure`).
2. `[[comment-metric]]` specifies how to compare and present results. Four comparison types are supported: `num_diff` (signed numeric deltas), `list_diff` (new items only), `only_new` (forward the raw new value), and `other` (a custom comparator loaded dynamically via `comparison-path` and `comparison-function`).

The output is a dictionary keyed by comment metric IDs. Values are comparison results, not raw measurements: badges expect signed numbers, lists contain newly introduced elements, and `other` entries hold pre-renderable structures for more complex sections.

Adding a new comparison requires only configuration: declare or reuse a collector under `[[load-comment-metrics]]`, then define a `[[comment-metric]]` with `num_diff`, `list_diff`, `only_new`, or a custom `other` comparator (providing `comparison-path` and `comparison-function`). The same dynamic loading mechanism used for collectors is reused for comparators and renderers, so the orchestrator code remains unchanged.

The QA PR Comment GitHub Actions workflow invokes this orchestrator after verifying that a baseline exists in `stats` because `metrics.json` is required for this orchestrator to run. The resulting `delta.json` is then used by the comment generator, which renders or updates the PR comment with badges, lists, and sections based on the comparison results.

5.4 Template Data Preprocessing

The Template Data Builder acts as the contract interpreter between the metrics configuration and the presentation layer. It reads `metrics_config.toml` to decide how each metric should appear: either as a badge, a list, or an *other* section. It then returns the Markdown fragments keyed by each metric's ID that appears in the Markdown templates. Simple forms (e.g., badges and compact bulleted lists) are synthesized directly. Complex sections (MI, CC, test metrics, vulnerability tables, dependencies list) are delegated to Render Helpers that are invoked through helper functions defined in the configuration (`render-path` and `render-function`). This module prepares exactly the placeholders expected by the templates.

The builder consumes two inputs: a JSONPath pointing to either `metrics.json` (baseline) or `delta.json` (PR deltas), and a `metric_type` flag indicating whether to render dashboard or comment metrics.

From `metrics_config.toml`, it uses the entries of `[[dashboard-metric]]` or `[[comment-metric]]` (with `id`, `display-type`, and per-type fields) and the `[[colorrule.*]]` section for color logic. Each configured ID must exist in the input JSON.

Badges are rendered from a `display-name`, optional `suffix`, and `colors`. If `colors` is defined by a rule in the configuration, it is applied. If it refers to another metric, that metric's value is used as the color. Otherwise, the value is treated as a literal color. All badges are rendered with the tool **shields.io**[1]. It allows to get a link to an image of a badge with the desired text and color. These badge images can be easily used in Github Markdown. The badge text is sanitized (hyphens normalized to em dashes, percent signs, and spaces made URL-safe).

Lists are formatted as compact bullet points with controlled line breaks. Other sections that are neither a badge nor a list are produced via dynamic hooks. The builder loads the specified render function and resolves any declared arguments using `parse_args`, which can pass literal strings or substitute values by referencing other metric IDs. This enables renderers for MI/CC summaries, test result overviews, vulnerability tables, and dependency diffs without hard-coding presentation logic.

Adding new visual elements requires no changes to this module. One has to define the element in `metrics_config.toml` with an `id` and `display-type`. For rich sections, implement a helper and reference it via `render-path` and `render-function`. Optional color rules or suffixes can be introduced or reused as needed.

5.5 Rendering and Presentation

Rendering converts raw values and comparison results into Markdown. This layer transforms `metrics.json` (baseline) and `delta.json` (PR deltas) into Markdown. It is made up of two core modules and their templates:

- the Dashboard Renderer (`generate_dashboard.py` with `dashboard-template.md`)
- the PR Comment Renderer & Publisher (`generate_comment.py` with `comment-template.md`)

After the Template Data Builder reads either `metrics.json` or `delta.json` and converts the raw values into a placeholder map, one of these steps is applied:

- the Dashboard Renderer fills `dashboard-template.md` to produce Markdown committed to the `stats` branch

[1] https://shields.io/

- the PR Comment Renderer & Publisher fills `comment-template.md` to create or update a PR comment

The resulting outputs are a `README.md` file in `stats` that serves as the live dashboard, and a PR comment tailored for review.

5.5.1 Dashboard Renderer

This module produces the Markdown for the QA Dashboard. It is invoked by the QA Dashboard GitHub Actions workflow and expects as input the JSON created by the baseline orchestrator (`create_json_all_metrics.py`).

The program receives a single argument, the path to `metrics.json`.

It combines `dashboard-template.md`, fills it with the elements from `generate_template_data.py`, and outputs the fully rendered Markdown document. In the workflow, this output is captured as `dashboard.md` and later renamed to `README.md` in the `stats` branch.

The renderer executes these steps:

1. Build the placeholder map: It calls the Template Data Builder `generate_template_data` with `metric_type="dashboard-metric"`. This step interprets `metrics_config.toml` to convert raw values into badges, lists, and rich sections (via render helpers).
2. Apply the template: It loads `dashboard-template.md` and uses Jinja2 to substitute the placeholders with the prepared fragments.
3. Print the result: The rendered Markdown is written to stdout.

All decisions about what appears and how it should be formatted (e.g., colors, suffixes, complex sections such as MI/CC/test summaries) are delegated to the *configuration* and the *Template Data Builder*. Jinja2 is used to compose the final document from the placeholder values and the template.

Adding new dashboard content does not require changes to this script. New metrics or sections are introduced by registering collectors and display rules in `metrics_config.toml`, providing optional render helpers for *other* sections, and placing new placeholders in `dashboard-template.md`.

5.5.2 PR Comment Renderer & Publisher

This module renders and publishes the QA comment on a PR. It is executed by the QA PR Comment GitHub Actions workflow and consumes `delta.json` produced by the comparison orchestrator. Its purpose is to present a compact Markdown comment that updates in place as the PR evolves.

The program expects three arguments: the GitHub repository name, the PR number, and the path to `delta.json`. It posts (or edits) a PR comment through the GitHub API.

The process is as follows:

1. Prepare placeholders: The Template Data Builder is invoked with `metric_type="comment-metric"`, transforming `delta.json` into a placeholder map (badges, lists, and other sections)
2. Apply the template: The module loads `comment-template.md` and renders it with Jinja2 using the prepared placeholders
3. Publish/update: Using the MECOISAURUS App, it authenticates, locates the target PR, searches for a prior comment containing the hidden identifier, and either edits it or creates a new one.

The module contains no metric computation or comparison logic. All presentation decisions are delegated to configuration and the Template Data Builder. Jinja2 is used to compose the final Markdown from placeholders.

New metrics in the PR comment require no changes to this module. Extend `metrics_config.toml` with additional `[[comment-metric]]` entries (including render helpers for `other` sections as needed), then add corresponding placeholders in `comment-template.md`.

5.5.3 The Markdown Templates

The templates (`dashboard-template.md` and `comment-template.md`) are Markdown skeletons whose placeholders correspond with metric IDs in the configuration. They define the presentation layer as plain Markdown with Jinja-style placeholders that the template data builder fills. `dashboard-template.md` organizes the overview of metrics for the `main` branch: badges and lists are placed where the corresponding `id`s appear, and complexer sections (`mi_overview`, `cc_overview`, `vulnerabilities`) are inserted as pre-rendered Markdown. Collapsible `<details>` blocks keep dense data compact.

`comment-template.md` mirrors the same structure but for deltas: placeholders reference comparison IDs. It combines badges and lists and separates the *old files/functions* sections for MI/CC to visualize regressions and improvements.

```
# MECOIS Quality Assurance Dashboard

## 🧩 Maintainability Metrics

#### 📏 Lines of Code (LOC)

<details>
<summary>
{{ loc }}
</summary>

    {{ lloc }} \
    {{ sloc }} \
    {{ commentl }} \
    {{ blankl }}
</details>

#### 🚫 Lint Disables

<details>
<summary> {{ lint_disables_len }}
</summary>

{{ lint_disables }}
</details>

#### 🧮 Maintainability Index
{{ mi_overview }}

#### ⚙️ Cyclomatic Complexity
{{ cc_overview }}

## 📊 Test Metrics

{{ tests_passed }}
    {{ test_duration }} \
    {{ test_coverage }}

## 🛡️ Security Metrics

#### 🔗 Dependencies

<details>
<summary> {{ dependencies_len }}
</summary>

{{ dependencies }}
</details>

#### ⚠️ Vulnerabilities

{{ vulnerabilities }}
```

Listing 5.1: `dashboard-template.md`

```
## MECOIS Quality Assurance Check

<sup>_Auto-generated by [`Mecoisaurus`](https://github.com/apps/
      mecoisaurus)_</sup>

### 🧩 Maintainability Metrics

#### 📏 Lines of Code (LOC)

<details>
<summary>
{{ loc_diff }}
</summary>

    {{ lloc_diff }} \
    {{ sloc_diff }} \
    {{ commentl_diff }} \
    {{ blankl_diff }}
</details>

#### 🚫 Lint Disables

{{ lint_disables_diff }}

#### 🧮 Maintainability Index
##### Old files
{{ mi_worse }}
{{ mi_better }}
{{ mi_new_section }}

#### ⚙️ Cyclomatic Complexity
##### Old functions
{{ cc_worse }}
{{ cc_better }}
{{ cc_new_section }}

### 📊 Test Metrics

{{ tests_passed_new }}
    {{ test_duration_diff }} \
    {{ test_coverage_diff }}

### 🛡️ Security Metrics

#### 🔗 Dependencies

{{ dependencies_new_diff }}
{{ dependencies_updated_diff }}

---

<sub>_Updated automatically on PR update. View full report in [
      QA Dashboard](../blob/stats/README.md)_</sub>
```

Listing 5.2: `comment-template.md`

5.5.4 A look at the rendered Markdown

Figure 5.2 shows the final rendered Markdown for the QA Dashboard. In this view, all collapsible sections are hidden to make the view more compact. The collapsible sections contain either other badges or lists.

Figure 5.3 shows the final rendered Markdown for the QA PR comment. To provide an extensive overview, the metrics for this figure were chosen to display optional sections and colors. In this view, all collapsible sections are hidden to make the view more compact. The collapsible sections contain either other badges or lists.

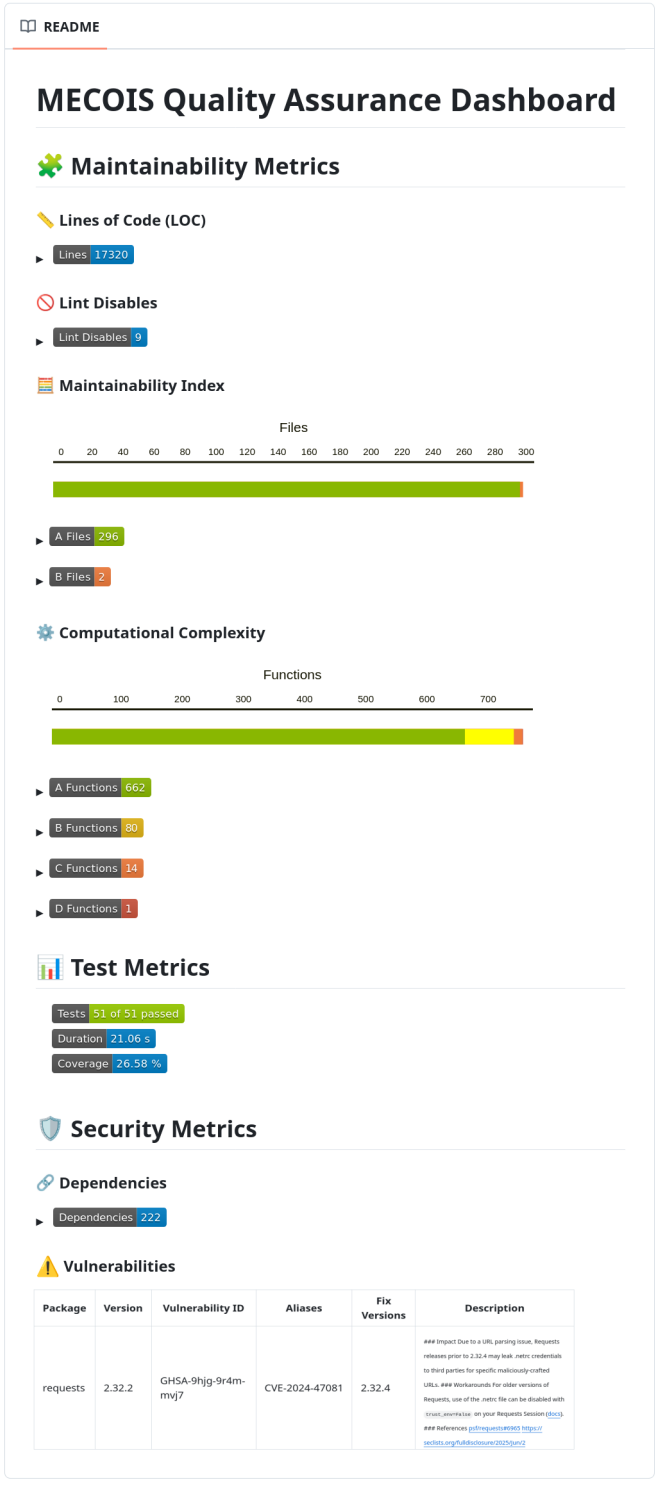

Figure 5.2: A look at the final QA Dashboard. (The dashboard is the `README.md` file of the storage branch `stats`.)

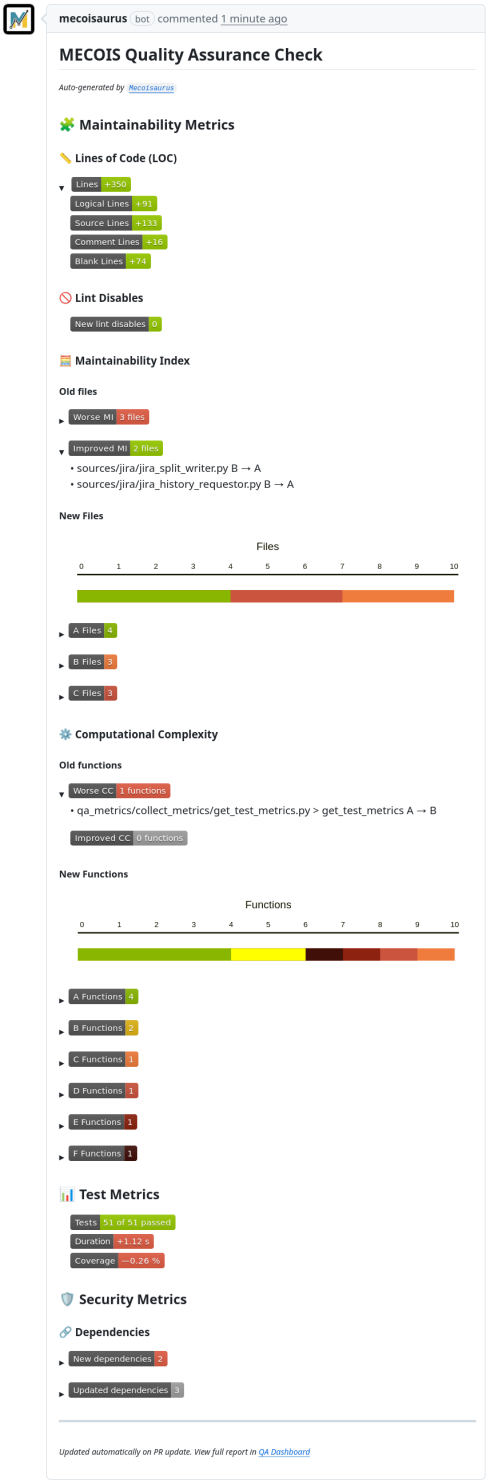

Figure 5.3: A look at the final QA PR comment. (The comment is posted in a PR where the flag *–generate-qa-comment* is set.)

5.6 Workflows

Two GitHub Actions workflows are responsible for the whole QA PR system:

- `qa-metrics-dashboard.yml` builds and publishes the baseline dashboard.
- `qa-metrics-comment.yml` recomputes metrics on PRs, computes differences, and posts a review comment.

The key artifacts that move through these pipelines are: `metrics.json` (the baseline snapshot) and `dashboard.md/README` committed to the `stats` branch for the dashboard, and `delta.json` for the PR comment.

5.6.1 Dashboard Workflow

The workflow is named and labeled for runs at the top of the file [**Dashboard L1–L2**]. It triggers on every push to the `main` branch [**Dashboard L3–L6**], ensuring that the baseline always reflects the current state of the codebase.

`Job get_metrics`: produce artifacts. This job installs the project dependencies and the QA-dashboard–specific dependencies [**Dashboard L17–L21**]. The baseline orchestrator is then run to collect metrics and write `metrics.json` [**Dashboard L23–L24**]. Immediately after, the dashboard is rendered from that JSON to `dashboard.md` [**Dashboard L26–L27**]. Both files are uploaded as a single artifact bundle (`dashboard_files`) for use in the publishing job [**Dashboard L29–L35**].

`Job push_metrics`: publish to `stats`. This job depends on successful completion of `get_metrics` via `needs: get_metrics` [**Dashboard L38**]. It again checks out the repository with full history [**Dashboard L56–L60**] and then switches to the `stats` branch, creating it as an orphan branch if it does not yet exist [**Dashboard L41–L54**]. The job downloads the artifact bundle [**Dashboard L56–L59**], renames `dashboard.md` to `README.md` [**Dashboard L61–L62**], and commits/pushes both `README.md` and `metrics.json` [**Dashboard L64–L70**].

Publishing to a dedicated stats branch provides repository-native storage without external infrastructure. New metrics or visual sections can be added by updating the configuration and templates. This workflow remains unchanged.

```yaml
name: QA Dashboard
run-name: QA Dashboard Update
on:
  push:
    branches:
      - main
jobs:
  get_metrics:
    name: Run metrics
    runs-on: ubuntu-latest
    steps:
        - name: Check out repository code
          uses: actions/checkout@v4
          with:
            fetch-depth: 0

        - name: Install project libraries
          run: pip3 install -r
          ↪  requirements.txt

        - name: Install qa dashboard
        ↪  specific libraries
          run: pip3 install -r
          ↪  qa_metrics/requirements.txt

        - name: Collect all Metrics
          run: python qa_metrics/create_jso
          ↪  n_all_metrics.py >
          ↪  metrics.json

        - name: Generate Dashboard
          run: python -m qa_metrics.dashboa
          ↪  rd.generate_dashboard
          ↪  metrics.json > dashboard.md

        - name: Upload metrics and dashboard
          uses: actions/upload-artifact@v4
          with:
            name: dashboard_files
            path: |
              metrics.json
              dashboard.md

  push_metrics:
    name: Push metrics to stats branch
    needs: get_metrics
    runs-on: ubuntu-latest
    steps:
        - name: Check out repository code
          uses: actions/checkout@v4
          with:
            fetch-depth: 0

        - name: Switch to stats branch
        ↪  (create if missing)
          run: |
            git fetch origin
            if git ls-remote --exit-code
            ↪  --heads origin stats; then
              git checkout stats
            else
              git checkout --orphan stats
              git rm -rf .
            fi

        - name: Download dashboard files
          uses: actions/download-artifact@v4
          with:
            name: dashboard_files

        - name: Rename dashboard.md to
        ↪  README.md
          run: mv dashboard.md README.md

        - name: Commit and Push Report
        ↪  directory
          run: |
            git config --global user.name
            ↪  "github-actions"
            git config --global user.email
            ↪  "github-actions@github.com"
            git add README.md metrics.json
            git commit -m "Metrics Test" ||
            ↪  exit 0
            git push origin stats
```

Listing 5.3: The Dashboard Workflow `qa-metrics-dashboard.yml`

5.6.2 PR Comment Workflow

The workflow responds to PR events, but its execution is gated by explicit flags in the PR body: its two jobs run only when the body contains *–generate-qa-comment* [**L12, L28**].

`Job check_stats`: ensure baseline exists. The workflow first checks out the repository [**L1–L12**] and verifies that the `stats` branch exists on the remote. If it does not, the job fails with an instructional message directing maintainers to run the dashboard workflow first [**L15–L23**]. This is essential because the PR comparison requires the baseline `metrics.json` stored in `stats`.

`Job qa_comment`: compute deltas and publish comment. The publishing job requires `check_stats` to have succeeded [**L27**]. It checks out the PR's code with full history [**L38–L43**], installs the QA toolchain, and installs project dependencies [**L32–L47**]. It then fetches the baseline `metrics.json` directly from `origin/stats` into the workspace [**L49–L51**] and runs the comparison orchestrator to generate `delta.json` [**L53–L55**]. Finally, it invokes the PR comment renderer, passing the repository, PR number, and `delta.json`, with GitHub App credentials supplied via environment variables [**L57–L66**].

```yaml
name: QA metrics PR comment
on:
  pull_request:
    types: [opened, synchronize, reopened, edited]
permissions:
  contents: read
  pull-requests: write

jobs:
  check_stats:
    name: Check if stats branch exists
    if: contains(github.event.pull_request.body, '--generate-qa-preview')
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Ensure 'stats' branch exists
        run: |
          git fetch origin
          if ! git ls-remote --exit-code --heads origin stats; then
            echo "Before this workflow can be used, the workflow QA Metrics Dashboard has to run at least once.
            Please push to main first or run the QA Metrics Dashboard workflow manually."
            exit 1
          fi

  qa_comment:
    name: Post PR qa metrics comment
    needs: check_stats
    if: contains(github.event.pull_request.body, '--generate-qa-comment')
    runs-on: ubuntu-latest

    steps:
      - name: Checkout PR code
        uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Set up Python and uv
        uses: astral-sh/setup-uv@v6
        with:
          python-version: 3.11
          activate-environment: true

      - name: Install dependencies with uv
        run: uv pip install -r qa_metrics/requirements.txt

      - name: Install project libraries
        run: pip3 install -r requirements.txt

      - name: Get metrics.json from stats branch
        run: |
          git checkout origin/stats -- metrics.json

      - name: Collect Metrics of PR code and compare them to metrics of main branch
        run: |
          python qa_metrics/create_json_compare_metrics.py metrics.json > delta.json

      - name: Post QA Preview comment via Mecoisaurus App
        env:
          MECOISAURUS_APP_ID: ${{ secrets.MECOISAURUS_APP_ID }}
          MECOISAURUS_PRIVATE_KEY: ${{ secrets.MECOISAURUS_APP_PRIVATE_KEY }}
          MECOISAURUS_INSTALLATION_ID: ${{ secrets.MECOISAURUS_INSTALLATION_ID }}
        run: |
          python -m qa_metrics.pr_comment.generate_comment \
            ${{ github.repository }} \
            ${{ github.event.pull_request.number }} \
            delta.json
```

Listing 5.3: The PR Comment Workflow `qa-metrics-comment.yml`

5.7 Summary

This chapter described the design and implementation of the QA feedback system. It explained how the QA dashboard and QA PR comment bot use six core components: configuration, metric collectors, orchestrators, template builders, renderers, and CI workflows. Together, they collect metrics, create Markdown reports, and publish results on GitHub. The system was shown to be modular and easy to extend through configuration. The next chapter, Evaluation, checks these results against the defined requirements using the predefined MoSCoW checklist from chapter 3.

6 Evaluation

This chapter evaluates the implemented QA feedback system based on the MoSCoW checklist defined in chapter 3 *Requirements*. The goal is to verify which of the prioritized requirements were fulfilled and to assess how well the system meets the intended objectives of automated quality feedback within *MECOIS*. In Figure 6.1, a filled-in version of the MoSCoW checklist is visualized.

Must-have Requirements

All must-have requirements were successfully met.
The **QA Dashboard** dynamically visualizes all collected metrics in Markdown format. It is automatically generated by the CI pipeline and stored in a GitHub branch, making the current quality always accessible.
The **QA PR comment bot** was fully implemented and integrated into GitHub Actions. It reacts to trigger words in PRs, compares the current branch with the baseline, and posts structured feedback as a comment.
Finally, the system uses a **metric set tailored to MECOIS**. Metrics were chosen to reflect maintainability, security, performance efficiency, and functional suitability, ensuring relevance to the specific goals of the MECOIS project.
But it has to be said that the system covers the defined QA characteristics to different extents:
For *Maintainability*, all formulated GQM questions were answered. The metrics give a detailed picture of structural quality and code comprehensibility. This characteristic is the most strongly supported by the current implementation.
Security questions were also addressed through the detection of dependencies and vulnerabilities. The implemented collectors identify known issues at build time. However, once *MECOIS* includes active users, additional aspects, such as authentication, data handling, and permission checks, should be monitored to provide a more complete view.
For *Functional Suitability*, only part of the questions could be answered. Test pass/fail results give a basic indication of correctness, but the system does not yet verify whether previously correct functionality remains intact after integration. Future extensions could integrate behavioral or regression testing to capture these aspects better.

For *Performance Efficiency*, test duration provides a first signal of runtime behavior for the test suite, but this alone is not sufficient to evaluate performance regressions or resource consumption stability. Dedicated performance or load testing would be necessary to address this quality characteristic meaningfully.

Should-have Requirements

All should-have requirements were also implemented.
The system provides a **shared and extensible measurement foundation**, meaning both the dashboard and the PR comment bot use the same configuration and metric collector modules. This modular design ensures consistency and simplifies maintenance.
Metrics such as MI, CC, and code coverage were integrated to **support review decisions** and make code quality visible.
Finally, the complete solution is **fully integrated into GitHub**. Both workflows are defined as GitHub Actions, and all outputs use GitHub-compatible Markdown for direct visibility in the development workflow.

Could-have and Won't-have Requirements

The could-have requirements were not implemented due to scope limitations. Features such as **performance testing**, **historical trend visualization**, and **advanced dashboards** were considered for future extensions.
Similarly, the won't-have requirements (additional quality characteristics, automated enforcement, and configurable projects and branch) remain out of scope but represent directions for future work.

Summary

In summary, all **must-have** and **should-have** requirements were fully achieved, establishing a stable and easily extensible foundation for quality feedback in *MECOIS*. The evaluation of quality characteristics shows that *maintainability* is well supported and that *security* is partially addressed. The other quality characteristics, *functional suitability* and *performance efficiency*, could be enhanced in future work.

Figure 6.1: Evaluated *MoSCoW* checklist of the most important requirements an implementation of a *MECOIS*-specific QA feedback system should have. Compare to the requirements version in Figure 3.2.

7 Conclusions

This thesis presents a GitHub-native quality feedback system built around two artifacts: a QA dashboard that captures the current quality state of the main branch, and a QA PR comment that highlights how the proposed changes in a PR affect the project's quality. Both aim to keep feedback close to where work happens and to focus on metrics that matter in collaborative projects. Maintainability, functional suitability, performance efficiency, and security were chosen as the quality characteristics on which the metrics are based. Results are delivered as plain Markdown inside GitHub, avoiding extra services.

The implementation realized all must-have and should-have requirements defined in chapter 3 *Requirements*. The QA dashboard dynamically visualizes metrics for *maintainability*, *functional suitability*, *performance efficiency*, and *security*. The QA PR comment bot compares code changes against a baseline, highlighting improvements or regressions directly in GitHub. Both components rely on a shared measurement foundation that allows consistent data collection and extensibility.

The evaluation confirmed that the system effectively supports the intended quality goals, though with varying coverage across characteristics. *Maintainability* is fully supported and benefits from detailed metrics such as LOC, MI, CC, and lint disables. *Security* is addressed through dependency and vulnerability checks, but further expansion will be necessary once *MECOIS* includes active users. *Functional suitability* and *performance efficiency* are only partially covered. While basic test metrics and execution times are available, deeper behavioral and performance analyses would be required to assess these quality goals meaningfully.

Overall, the project demonstrates that automated QA feedback can be integrated into GitHub workflows with minimal overhead. The solution aligns with the goal of extensibility. By using configuration files and modular collectors, it can be easily adapted for other projects or extended with new metrics.

Several directions remain open for future development. The system could be extended with performance testing to better capture efficiency regressions. Historical trend visualization across commits or releases would add meaningful insight into project quality. Furthermore, a more sophisticated appearance could enhance the usability of the QA system. Finally, future iterations might explore automated enforcement mechanisms and additional quality characteristics like reliability or flexibility.

In conclusion, this work shows that even a simple QA feedback system can provide useful guidance without relying on complex platforms. By focusing on a limited set of metrics and presenting them clearly, developers can gain helpful insights on code quality. The QA dashboard and QA PR comment currently offer basic assistance, though there is room for improvement. With further refinements, the system could better support secure and maintainable code in collaborative projects.

References

Albrecht, A. J. (1979). Measuring application development productivity. *Proceedings of the IBM Applications Development joint SHARE/GUIDE Symposium*, 83–92.

Balachandran, V. (2013). Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. *Proceedings - International Conference on Software Engineering*, 931–940. https://doi.org/10.1109/ICSE.2013.6606642

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). The goal question metric approach. *Encyclopedia of Software Engineering*, 528–532.

Basili, V. R., & Rombach, H. D. (1988). The tame project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, *14*(6), 758–773. https://doi.org/10.1109/32.6156

Fenton, N. E., & Neil, M. (1999). Software metrics: Success, failures and new directions. *J. Syst. Softw.*, *47*(2–3), 149–157. https://doi.org/10.1016/S0164-1212(99)00035-7

Halstead, M. H. (1977). *Elements of software science*. Elsevier North-Holland.

Heričko, T., & Šumak, B. (2023). Exploring maintainability index variants for software maintainability measurement in object-oriented systems. *Applied Sciences*, *13*. https://doi.org/10.3390/app13052972

International Organization for Standardization & International Electrotechnical Commission. (2014). *ISO/IEC 25000:2014 systems and software engineering — systems and software quality requirements and evaluation (square) — guide to square*. https://www.iso.org/standard/64764.html

International Organization for Standardization & International Electrotechnical Commission. (2023). *ISO/IEC 25010:2023 systems and software engineering — systems and software quality requirements and evaluation (square) — product quality model*. https://www.iso.org/standard/78176.html

ISO 25000 Portal. (2024). *Iso/iec 25010*. Retrieved October 27, 2025, from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Jones, C. (2008). *Applied software measurement: Global analysis of productivity and quality* (3rd ed.). McGraw-Hill Professional.

Kafura, D. (2025). Reflections on McCabe's Cyclomatic Complexity. *IEEE Transactions on Software Engineering*, *51*(03), 700–705. https://doi.org/10.1109/TSE.2025.3534580

Lopez, L., Manzano, M., Gómez, C., Oriol, M., Farré, C., Franch, X., Martínez-Fernández, S., & Vollmer, A. M. (2021). Qasd: A quality-aware strategic dashboard for supporting decision makers in agile software development. *Science of Computer Programming*, *202*, 102568. https://doi.org/10.1016/j.scico.2020.102568

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Mills, E. E. (1988, December). Software metrics [Curriculum Module]. https://www.sei.cmu.edu/library/software-metrics/

Oman, P., & Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. *Proceedings Conference on Software Maintenance 1992*, 337–344. https://doi.org/10.1109/ICSM.1992.242525

Staron, M., Meding, W., Hansson, J., Höglund, C., Niesel, K., & Bergmann, V. (2014). Dashboards for continuous monitoring of quality for software product under development. *Relating System Quality and Software Architecture*, 209–229. https://doi.org/10.1016/B978-0-12-417009-4.00008-9

Voas, J., & Kuhn, D. (2017). What happened to software metrics? *Computer*, (50). https://doi.org/https://doi.org/10.1109/MC.2017.144

Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., & Gerosa, M. A. (2022). Quality gatekeepers: Investigating the effects of code review bots on pull request activities. *Empirical Software Engineering*, *27*. https://doi.org/10.1007/s10664-022-10130-9