SmartDelta

Automated Quality Assurance and Optimization in Incremental Industrial Software Systems Development

D5.6 – Software Quality Metrics Guideline

Submission date of deliverable: Mar 30, 2025

Edited by: SmartDelta Consortium

Project start date	Dec 1, 2021
Project duration	36 months
Project coordinator	Dr. Mehrdad Saadatmand, RISE Research Institutes of Sweden
Project number & call	20023 - ITEA 3 Call 7
Project website	https://itea4.org/project/smartdelta.html & https://smartdelta.org/

Contributing partners	SmartDelta Consortium
Version number	V1.0
Work package	WP5
Work package leader	Bilge Özdemir
Dissemination level	Public
Description	This document provides methods for visualizing and interpreting relevant tool outputs of WP5 using selected visualization types and explanations of each metric's importance and action recommendations based on their values. This guide enables end-users to understand the implications of the data presented and how it should influence their product deltas.

Executive Summary

The "D5.6 SmartDelta Metrics Guideline" document details the approach to defining, measuring, and visualizing key performance indicators within the SmartDelta project. It begins by outlining the purpose and importance of metrics, emphasizing their role in monitoring project progress and outcomes.

The document categorizes metrics into various types, distinguishing between quantitative and qualitative measures. It provides detailed methodologies for metric collection, analysis, and interpretation, ensuring uniformity across project contributors. Additionally, it highlights best practices for visualization, specifically through the use of Vaadin charts, to enhance the clarity and accessibility of complex data.

Further sections delve into case studies and practical applications of the proposed metrics framework, showcasing real-world examples from project partners. Recommendations for continuous improvement and adaptability of the metrics system are also included, ensuring long-term applicability and relevance.

By following this guideline, SmartDelta stakeholders can leverage a standardized approach to metric evaluation, fostering improved decision-making and operational efficiency across the project lifecycle.

Table of Contents

1. Introduction

Effective visualization of software quality metrics is essential for enabling stakeholders to interpret data, derive meaningful insights, and take informed actions. This guideline is designed to help different stakeholders—ranging from developers and project managers to decision-makers—identify the most suitable visualization techniques for their needs. By presenting metrics in a clear and actionable manner, we aim to enhance understanding and facilitate decision-making processes related to software deltas.

2. Cybersecurity Response Efficiency – Mean Time to Detect (MTTD) and Mean Time to Respond (MTTR)

2.1. Definition

- **MTTD (Mean Time to Detect):** Measures the time from log ingestion to security incident alerting, indicating how quickly potential threats are identified.
- **MTTR (Mean Time to Respond):** Measures the time from incident detection to response initiation, assessing the speed at which security teams take action after an alert.

2.2. Importance

Monitoring MTTD and MTTR is critical in cybersecurity to minimize the time threats remain undetected and unaddressed. A lower MTTD ensures faster threat identification, reducing potential damage, while a lower MTTR enhances the ability to contain and mitigate risks efficiently.

2.3. Actions

- **If MTTD is high**, enhance log processing, optimize detection rules, and leverage AI-based anomaly detection to reduce alerting delays.
- **If MTTR is high**, streamline response workflows, automate remediation actions, and improve SOC analyst coordination to initiate faster resolutions.
- Conduct periodic reviews of detection and response times to identify bottlenecks and enhance security posture.

2.4. Sources

- Security Information and Event Management (SIEM) systems (e.g., IBM QRadar)
- Security Orchestration, Automation, and Response (SOAR) platforms
- Incident response management tools
- SOC incident logs and reports

2.5. Grouping Variables

I. **Severity Levels** (Critical, High, Medium, Low)
II. **Incident Type** (Phishing, Malware, Ransomware, Insider Threats)
III. **Response Teams** (SOC Analysts, IT, Incident Response Teams)
IV. **Time Periods** (Hourly, Daily, Weekly, Monthly)

2.6. Correlated Metrics

- **False Positive Rate:** High false positives can delay real incident detection, increasing MTTD.
- **Incident Volume:** A surge in incidents may lead to higher MTTR due to resource constraints.

2.7. Visualization

- **Line Graphs** to track trends in MTTD and MTTR over time.
- **Bar Charts** to compare MTTD/MTTR across incident categories or severity levels.

3. Automated Test Generation Efficiency

3.1. Definition

This metric evaluates the efficiency of automated test generation tools and methodologies, specifically their ability to generate test cases for industrial control software, such as PLC programs. Efficiency is measured by the time required to generate test cases that satisfy specific coverage criteria using PyLC, SEAFOX and TIGER+.

3.2. Importance

Efficiency in automated test generation is critical for industrial projects, particularly in safety-critical domains. This metric provides insights into the usefulness of an automated testing tools.

3.3. Actions

If automated test generation efficiency is low, potential actions include:

- Optimizing tool configurations: Adjust parameters, such as search heuristics or timeout settings, to improve performance.
- Parallelizing test generation: Implement parallel processing to reduce time for large test suites.
- Evaluating alternative tools or methods: Consider other test design techniques if efficiency remains an issue.

3.4. Sources

Sources for data include:

- Test generation logs: Capture timestamps and events.
- Code repositories: Store generated test cases for analysis.
- Automation tools: Tools like PyLC and SEAFOX that log metrics.

3.5. Grouping Variables

Grouping variables for analysis include:

I. **Project phases:** Measure efficiency across different development stages.
II. **Software modules:** Track efficiency for various system components.
III. **Coverage criteria:** Analyse by type (e.g., decision coverage vs. MC/DC).

3.6. Correlated Metrics

This metric correlates with:

- Test case execution time: Measures the average time to execute each test case.

3.7. Visualization

- Bar charts to display test generation time across projects.
- Line graphs to show trends in efficiency improvements over time.

4. Code Quality Score

4.1. Definition

The code quality score of a project refers to the overall rating of the written source code, which is comprised of multiple factors like adherence to best-practice coding guidelines, number of duplicated code lines or use of deprecated methods.

4.2. Importance

Indicates the overall health of the software project. A low code quality score may suggest a high number of bugs, poor performance, or inadequate requirements.

4.3. Actions

Implement coding guidelines, conduct code reviews, and enhance requirements analysis to increase the code quality.

4.4. Sources

- Code repository
- Issue tracking system
- TWT’s CSSA

4.5. Grouping Variables

I. Product

- **Definition**: The product that multiple code repositories contribute to.
- **Importance**: Presents the ability to get an overview of the code quality of the whole product.
- **Examples:**
  - Aggregated code quality score for each product

4.6. Correlated Metrics

- **Bad Code**

  Correlation: Larger size of bad code leads to lower code quality.
  Action: Identify bad code segments and improve them.

4.7. Visualization

- **Bar Charts**

Use Case:

- Comparing the code quality scores of different products or projects.

Advantages:

- Easy to compare values across products/projects.
- Clearly shows trends and differences.

Example:

- **X-Axis:** Product or project identifier.
- **Y-Axis:** Code quality score.

Additional Elements: Different colours or stacked bars to represent quality levels.

- **Line Charts**

Use Case:

- Tracking the trend of the code quality score over time.
- Monitoring the progress of code quality enhancements efforts.

Advantages:

- Shows trends and changes over time.
- Easy to identify peaks and troughs.

Example:

- **X-Axis:** Time periods (days, weeks, months).
- **Y-Axis:** Code quality score.
- **Additional Elements:** Separate lines for each product/project.

- **Tables**

Use Case:

- Detailed representation of scores.
- Including multiple attributes such as product/project identifier, score level, last reported date.

Advantages:

- Comprehensive and detailed.
- Allows for sorting and filtering.

Example:

- Columns for Product/Project ID, Score, Score Level, Last reported Date.

5. Number of Artefacts Analysed per Product

5.1. Definition

The "number of artefacts analyzed per product” refers to the total count of artefacts like code repositories that have been analysed for their code quality. The count is aggregated for each product.

5.2. Importance

Provides an impression of the relative size of the products, their size evolution and whether the analysis is running at all.

5.3. Actions

Verify that all artefacts of a product are being analysed for code quality.

5.4. Sources

- Code Repositories

5.5. Grouping Variables

I. **Artefact type**
   - **Definition**: The artefact types that were analysed.
   - **Importance**: Overview of the share of each artefact type in the analysis.
   - **Examples**:
     - Source code, configuration files, compiled code, Docker images

5.6. Correlated Metrics

None.

5.7. Visualization

- **Bar Charts**

Use Case:

- Comparing the number of artefacts analysed across different products.

Advantages:

- Easy to compare values across categories.
- Clearly shows trends and differences.

Example:

- X-Axis: Product
- Y-Axis: Number of artefacts analysed.
- Additional Elements: Different colours or stacked bars to represent artefact types.

- **Line Charts**

Use Case

- Tracking the trend of the number of artefacts analysed over time.
- Monitoring the growth of a product.

Advantages:

- Shows trends and changes over time.
- Easy to identify peaks and troughs.

Example:

- X-Axis: Time periods (days, weeks, months).
- Y-Axis: Number of artefacts analysed.
- Additional Elements: Separate lines for each product.

- **Tables**

Use Case:

- Detailed representation of artefact data.
- Including multiple attributes such as artefact type, source, last analysis timestamp.

Advantages:

- Comprehensive and detailed.
- Allows for sorting and filtering.

Example:

- Columns for Product, Number of artefacts analysed, artefact types and counts, artefact sources, last analysis timestamps.

6. Number of Bugs

The "number of bugs" in a project refers to the total count of identified defects, issues, or errors within the software. A bug is any unintended behaviour or problem that causes the software to deviate from its expected performance, functionality, or user experience.

6.1. Definition

Indicates the overall health of the software project. A high number of bugs may suggest poor code quality, lack of testing, or inadequate requirements.

6.2. Importance

Implement rigorous testing, conduct code reviews, and enhance requirements analysis to reduce the number of bugs.

6.3. Actions

- Project Management Tools
- Version Control Systems

6.4. Sources

- Project Management Tools
- Version Control Systems

6.5. Grouping Variables

I. **Issue Criticality**

- **Definition**: The level of importance or urgency associated with resolving the issue.

- **Importance**: High criticality issues need immediate attention to prevent major disruptions.
- **Examples**:
  - **Severity Levels**: Critical, Major, Minor, Trivial.
  - **Priority Levels**: High, Medium, Low.

II. Issue Lead Times

- **Definition**: The time taken to resolve an issue from the moment it is reported to the moment it is resolved.
- **Importance**: Long lead times can indicate inefficiencies in the resolution process.
- **Examples**:
  - **Lead Time Categories**: Less than 1 day, 1-3 days, 3-7 days, more than 7 days.
  - **Mean Time to Resolve (MTTR)**: Average time taken to resolve issues.

III. Components of Issues

- **Definition**: The specific parts or modules of the software system where the issues are located.
- **Importance**: Identifying problematic components helps focus efforts on
- improving specific parts of the software.
- **Examples**:
  - **Software Modules**: User Interface, Database, API, Authentication, Payment System.
  - **Subsystems**: Frontend, Backend, Middleware.

IV. Assigned Developer/Team

- **Definition**: The individual developer or team responsible for resolving the issue.
- **Importance**: Identifies workload distribution and potential bottlenecks within teams.
- **Examples**:
  - **Individual Developers**: Developer A, Developer B.
  - **Teams**: Frontend Team, Backend Team, QA Team.

V. Issue Source

- **Definition**: The origin or detection point of the issue.
- **Importance**: Understanding the source of issues helps improve detection and reporting processes.
- **Examples**:
  - **Detection Methods**: Manual Testing, Automated Testing, User Feedback, Code Review.
  - **Reported By**: QA Team, End Users, Developers.

6.6. Correlated Metrics

- **Code Complexity (Cyclomatic Complexity)**
  Correlation: Higher code complexity often leads to more bugs.
  Action: Simplify complex code and use static analysis tools to identify high-complexity areas.

- **Code Coverage**
  Correlation: Lower code coverage in testing can result in more undetected bugs or higher bug count with high coverage can mean low quality tests.
  Action: Increase unit and integration tests to improve code coverage or improve tests.

- **Team Velocity**
  Correlation: Higher team velocity might lead to rushed work and more bugs.
  Action: Balance speed with quality, ensuring adequate testing and code reviews.

6.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing the number of bugs across different time periods or components.
  - Visualising the distribution of bugs by severity or priority.

  Advantages:
  - Easy to compare values across categories.
  - Clearly shows trends and differences.

  Example:
  - **X-Axis:** Time periods (weeks, months) or components (modules, subsystems).
  - **Y-Axis:** Number of bugs.
  - **Additional Elements:** Different colours or stacked bars to represent severity levels.

- **Line Charts**

Use Case:
  - Tracking the trend of the number of bugs over time.
  - Monitoring the progress of bug resolution efforts.

  Advantages:
  - Shows trends and changes over time.
  - Easy to identify peaks and troughs.

  Example:
  - **X-Axis:** Time periods (days, weeks, months).
  - **Y-Axis:** Number of bugs.
  - **Additional Elements:** Separate lines for different severity levels or components.

- **Pie Charts**
  Use Case:
  - Showing the proportion of bugs by severity, priority, or type.
  - Visualising the distribution of bugs across different components.

  Advantages:
  - Provides a clear visual representation of proportions.
  - Easy to understand at a glance.

  Example:
  - Slices for different severities (critical, major, minor) or components (UI, backend).

- **Tables**
  Use Case:
  - Detailed representation of bug data.
  - Including multiple attributes such as bug ID, description, severity, status, and assigned developer.

  Advantages:
  - Comprehensive and detailed.
  - Allows for sorting and filtering.

  Example:
  - Columns for Bug ID, Description, Severity, Assigned Developer, Date Reported.

- **Heat Maps**
  Use Case:
  - Highlighting areas with high concentrations of bugs.
  - Visualising the density of bugs across different components or time periods.

  Advantages:
  - Quickly identifies hotspots.
  - Effective for large datasets.

  Example:
  - Rows and Columns: Components and severity levels.
  - Colours: Gradient representing the number of bugs.

- **Scatter Plots**
  Use Case:
  - Correlating the number of bugs with other metrics such as code complexity, code churn, or team velocity.

  Advantages:
  - Shows relationships between two variables.
  - Useful for identifying patterns and correlations.

  Example:
  - X-Axis: Code complexity or another correlated metric.
  - Y-Axis: Number of bugs.
  - Points: Individual data points, possibly color-coded by severity.

7. Number of Frequency of Degrading Code Commits

7.1. Definition

The "number of degrading code commits" in a project refers to the total count of code commits that were identified to affect the product quality detrimentally based on the analysis of previous code and defects. The “frequency of degrading code commits” refers to the distribution of degrading code commits over a specific time period.

7.2. Importance

Indicates the health of a software project as well as the quality of the work put into its development.

7.3. Actions

Implement rigorous testing, conduct code reviews, and ensure that the development is not rushed.

7.4. Sources

- Code repository
- UIBK’s SoHist

7.5. Grouping Variables

I. **Product**

- **Definition**: The product that multiple code repositories contribute to.

- **Importance**: Presents the ability to get an overview of the number of degrading code commits of the whole product.
- **Examples**: Aggregated number of degrading code commits for each product.

7.6. Correlated Metrics

I. **Bad code**
- **Correlation:** More degrading commits lead to an increase of bad code sections.
- **Action:** Implement rigorous testing, conduct code reviews, and ensure that the development is not rushed.

II. **Test percentage advised/Test scenarios absent**
- **Correlation:** A low number of tests and low code coverage leads to bad code remaining unidentified before committing.
- **Action:** Increase unit and integration tests to improve code coverage or improve tests.

7.7. Visualization

- **Bar Charts**
  Use Case:
    - Comparing the number of degrading commits across different products.

  Advantages:
    - Easy to compare values across products.
    - Clearly shows trends and differences.

  Example:
    - **X-Axis:** Product.
    - **Y-Axis:** Number of degrading code commits.

- **Line Charts**
  Use Case:
    - Tracking the trend and frequency of degrading code commits over time.
    - Monitoring the progress of code quality improvement efforts.

  Advantages:
    - Shows trends and changes over time.
    - Easy to identify peaks and troughs.

  Example:
    - **X-Axis:** Time periods (days, weeks, months).
    - **Y-Axis:** Number of degrading code commits.
    - **Additional Elements:** Separate lines for different products.

- **Tables**
  Use Case:
    - Detailed representation of commit data.
    - Including multiple attributes such as commit ID, description and assigned developer.

  Advantages:
    - Comprehensive and detailed.
    - Allows for sorting and filtering.

  Example:
    - Columns for Commit ID, Description, Assigned Developer, Date.

- **Scatter Plots**
  Use Case:
  - Correlating the number of degrading code commits with other metrics such as bad code or test scenarios absent.

  Advantages:
  - Shows relationships between two variables.
  - Useful for identifying patterns and correlations.

  Example:
  - **X-Axis:** Bad code or another correlated metric.
  - **Y-Axis:** Number of degrading code commits.
  - **Points:** Individual data points, possibly color-coded or scaled by size.

8. Number of Issues Similar to Existing Code

8.1. Definition

The "number of issues similar to existing code" for a product refers to the total count of issues (features or defects) that have not yet been implemented and were identified as being similar to issues from other analysed products. The analysis shows unimplemented issues where the description is similar to those of already implemented issues of other products.

8.2. Importance

Indicates the number of issues, for which existing code could be re-used.

8.3. Actions

Use issue similarity suggestions to identify similar issues across products to re-use code fixes.

8.4. Sources

- Issue tracking system
- IFAK's Issue Analysis Tool

8.5. Grouping Variables

None.

8.6. Correlated Metrics

None.

8.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing the number of issues similar to existing code across different products.

  Advantages:
  - Easy to compare values across products.
  - Clearly shows trends and differences.

  Example:
  - **X-Axis:** Product
  - **Y-Axis:** Number of issues similar to existing code.

- **Line Charts**
  Use Case:
  - Tracking the trend of the number of issues similar to existing code over time.

  Advantages:
  - Shows trends and changes over time.
  - Easy to identify peaks and troughs.

  Example:
  - **X-Axis:** Time periods (days, weeks, months).
  - **Y-Axis:** Number of issues similar to existing code.
  - **Additional Elements:** Separate lines for each product.

- **Tables**
  Use Case:
  - Detailed representation of issue data.
  - Including multiple attributes such as issue ID, description, similarity score, product.

  Advantages:
  - Comprehensive and detailed.
  - Allows for sorting and filtering.

  Example:
  - Columns for Issue ID, Description, Similarity Score, Product ID, Date Reported.

9. Number of Similar Issues with Different Code

9.1. Definition

The "number of similar issues with different code" for a product refers to the total count of issues (features or defects) that are already implemented and have been identified as being similar to issues from other analysed products but have been implemented in different ways. The analysis shows issues where the description is similar, but the code is different.

9.2. Importance

Indicates the number of feature or fix implementations, that were done multiple times across different products from scratch but could have been re-used.

9.3. Actions

Use issue similarity suggestions to identify similar issues across products to re-use code fixes.

9.4. Sources

- Code repository
- Issue tracking system
- IFAK’s Issue Analysis Tool
- TWT’s CSSA

9.5. Grouping Variables

None.

9.6. Correlated Metrics

I. **Number of similar issues with similar code**

- **Correlation:** New similar issues can only be categorized in one of both metrics, so both will increase with the number of issues increasing.
- **Action:** Use issue similarity suggestions to identify similar issues across products to re-use code fixes.

9.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing the number of similar issues with different code across different products.

  Advantages:
  - Easy to compare values across products.
  - Clearly shows trends and differences.

  Example:
  - **X-Axis:** Product
  - **Y-Axis:** Number of similar issues with different code.

- **Line Charts**
  Use Case:
  - Tracking the trend of the number of similar issues with different code over time.

  Advantages:
  - Shows trends and changes over time.
  - Easy to identify peaks and troughs.

  Example:
  - **X-Axis:** Time periods (days, weeks, months).
  - **Y-Axis:** Number of similar issues with different code.
  - **Additional Elements:** Separate lines for each product.

- **Tables**
  Use Case:
  - Detailed representation of issue data.
  - Including multiple attributes such as issue ID, description, similarity score, product.

  Advantages:
  - Comprehensive and detailed.
  - Allows for sorting and filtering.

  Example:
  - Columns for Issue ID, Description, Similarity Score, Product ID, Date Reported.

10. Number of Similar Issues with Similar Code

10.1. Definition

The "number of similar issues with similar code" for a product refers to the total count of issues (features or defects) that have been identified as being similar to issues from other analysed products and have been implemented in similar ways. The analysis shows issues where the description and the code are similar.

10.2. Importance

Indicates the number of feature or fix implementations, that were done multiple times across different products in similar ways and may have been re-used.

10.3. Actions

Use issue similarity suggestions to identify similar issues across products to re-use code fixes.

10.4. Sources

- Code repository
- Issue tracking system
- IFAK’s Issue Analysis Tool
- TWT’s CSSA

10.5. Grouping Variables

None.

10.6. Correlated Metrics

I. **Number of similar issues with different code**

- **Correlation:** New similar issues can only be categorized in one of both metrics, so both will increase with the number of issues increasing.
- **Action:** Use issue similarity suggestions to identify similar issues across products to re-use code fixes.

10.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing the number of similar issues with similar code across different products.

  Advantages:
  - Easy to compare values across products.
  - Clearly shows trends and differences.

  Example:
  - **X-Axis:** Product
  - **Y-Axis:** Number of similar issues with similar code.

- **Line Charts**
  Use Case:
  - Tracking the trend of the number of similar issues with similar code over time.

  Advantages:
  - Shows trends and changes over time.
  - Easy to identify peaks and troughs.

  Example:
  - **X-Axis:** Time periods (days, weeks, months).
  - **Y-Axis:** Number of similar issues with similar code.
  - **Additional Elements:** Separate lines for each product.

- **Tables**

Use Case:
- Detailed representation of issue data.
- Including multiple attributes such as issue ID, description, similarity score, product.

Advantages:
- Comprehensive and detailed.
- Allows for sorting and filtering.

Example:
- Columns for Issue ID, Description, Similarity Score, Product ID, Date Reported.

11. Observable Decision Coverage

11.1. Definition

Observable Decision Coverage measures how decisions in industrial control software (e.g., FBD programs developed under IEC 61131-3 standards) are tested for all possible outcomes. Specifically, this metric evaluates observable Boolean outcomes of each decision without going into the internal logic. Observable Decision Coverage delta is an extension of this metric, representing changes in coverage levels over time, capturing improvements or declines in decision coverage. Monitoring the delta helps ensure that modifications, refinements, or additions to software do not introduce untested decision points.

11.2. Importance

Test coverage is essential for quality assurance, particularly in safety-critical control software applications. High observable decision coverage with positive delta trends contributes to overall software quality and reduces risks associated with undetected faults in critical decision-making logic.

11.3. Actions

To respond to observable decision coverage findings, particularly delta trends:

- Expand Combinatorial Techniques: For complex decisions, consider t-way combinatorial testing (e.g., from pairwise to 3-way) to ensure higher coverage.
- Manual Augmentation: For cases where test generation tools fall short, create

11.4. Sources

Data for Observable Decision Coverage and its delta can be collected from:

- CODESYS IDE: For developing, running, and exporting PLC program test results.
- SEAFOX: Generates test cases and input data, for coverage analysis.
- CfUnit: Integrated testing tool in CODESYS for executing tests and logging.

11.5. Grouping Variables

Grouping for Observable decision coverage and its delta can provide insights:

I. Test Generation Method: Group by technique (e.g., Pairwise, Base Choice, Random) to assess coverage levels and trends across combinatorial strategies.
II. Project Milestone or Release: Compare coverage at different development stages to monitor delta trends and ensure ongoing quality.

11.6. Correlated Metrics

Observable decision coverage and delta correlate with other metrics:

- Defect density
- Test generation efficiency

11.7. Visualization

Visualizations for observable decision coverage and delta:

- Bar charts: show coverage levels highlighting areas with lower coverage and negative deltas.
- Scatter plots: use scatter plots to show relationships between observable decision coverage and correlated metrics like defect density.

12. Requirements Smell Score

12.1. Definition

The Requirements smell score is a composite metric that quantifies the presence of multiple quality issues ("smells") in natural language requirements. These smells include vagueness, optionality, subjectivity, complexity, readability, and referenceability, among others. The score is calculated by aggregating individual metrics that detect these smells, resulting in a single score that reflects the overall quality and clarity of the requirements document. The score can range from low (indicating minimal issues) to high (indicating many issues), helping users identify requirements that may require further refinement.

12.2. Importance

The Requirements smell score is a critical metric for ensuring the quality of requirements in software projects. It helps teams identify areas where requirements may be unclear, ambiguous, or overly complex, which can lead to misinterpretations and rework. By tracking this metric, project teams gain insights into their requirements documentation, enabling them to address quality issues early in the development process.

12.3. Actions

Based on the requirements smell score, teams can take the following actions:

- Refinement: requirements with high scores may benefit from clarification, simplification, or rephrasing to reduce ambiguity.
- Training: reams may consider training stakeholders on writing high-quality, more unambiguous requirements.
- Verification: requirements with high smell scores can undergo additional reviews or stakeholder consultations to ensure they are well-understood.
- Documentation Standards: develop or refine standards for writing requirements and reduce the occurrence of smells.

12.4. Sources

Requirements smell score can be gathered from:

- Requirements Documents: primary data sources such as software requirement specifications, functional requirements documents, or user stories.

- Tools: Tools like NALABS or other Natural Language Processing (NLP) systems that analyse text and detect defined smell metrics.
- Document Repositories: Requirements stored in repositories like GitHub can be processed to extract relevant smell metrics.

12.5. Grouping Variables

The Requirements smell score can be grouped or analysed according to several variables for insights:

I. By requirement type: group deltas by functional, non-functional, or security requirements to see where issues may arise over time.
II. By component: track deltas across different system components to pinpoint areas where requirements quality is improving or deteriorating.
III. By project milestone or phase: deltas can be monitored at each project milestone to assess the impact of recent changes, adjustments, or corrective actions.

12.6. Correlated Metrics

Deltas in the requirements smell score can be further contextualized by correlating with other metrics:

- **Complexity:** Rising complexity metrics may correlate with increases in smell scores, suggesting areas where requirements could benefit from simplification.
- **Defect density:** tracking changes in defect density alongside smell score deltas may reveal a direct link between requirements quality and defect occurrences.
- **Readability:** As readability improves or declines, corresponding shifts in the requirements smell score may signal changes in complexity or structure.

12.7. Visualization

Visualization is essential for tracking and interpreting the requirements smell score and its deltas:

- **Heatmaps:** Score and delta variations across different sections or components.
- **Time series line charts:** Display smell score deltas over time, illustrating trends and impacts of improvement efforts.
- **Box plots with deltas:** Highlight score distributions and changes over specified time periods.

13. Size of Bad Code

13.1. Definition

The "size of bad code" in a project refers to the total size of bad code segments (e.g. in code lines) identified in the overall source code base as affecting the product quality detrimentally.

13.2. Importance

Indicates the health of the software project. A large amount of bad code may lead to bugs, poor performance or the non-fulfilment of requirements.

13.3. Actions

Implement rigorous testing, conduct code reviews, and reduce technical debt to reduce the size of bad code.

13.4. Sources

- Code repository
- UIBK's SoHist

13.5. Grouping Variables

I. **Product**

- **Definition**: The product that multiple code repositories contribute to.
- **Importance**: Presents the ability to get an overview of size of bad code for the whole product.
- **Examples**: Aggregated size of bad code for each product

13.6. Correlated Metrics

I. **Test percentage advised/Test scenarios absent**

- **Correlation:** A low number of test scenarios may lead to more bad code remaining unidentified.
- **Action:** Increase test coverage to identify bad code segments.

II. **Number of degrading code commits**

- **Correlation:** A high number of degrading code commits often leads to the increase of bad code sections.
- **Action:** Increase testing

13.7. Visualization

- **Bar Charts**
  Use Case:
    - Comparing the size of bad code across different products.

  Advantages:
    - Easy to compare values across categories.
    - Clearly shows trends and differences.

  Example:
    - **X-Axis:** Product.
    - **Y-Axis:** Size of bad code.
    - **Additional Elements:** Different colours or stacked bars to represent types of bad code.

- **Line Charts**
  Use Case:
    - Tracking the trend of the size of bad code over time.
    - Monitoring the progress of code quality improvement efforts.

  Advantages:
    - Shows trends and changes over time.
    - Easy to identify peaks and troughs.

  Example:
    - **X-Axis:** Time periods (days, weeks, months).
    - **Y-Axis:** Size of bad code.
    - **Additional Elements:** Separate lines for different products.

- **Pie Charts**

Use Case:
- Showing the proportion of bad code relative to total code size.
- Visualising the distribution of bad code across different products.

Advantages:
- Provides a clear visual representation of proportions.
- Easy to understand at a glance.

Example:
- One pie chart per product with one slice for bad code and one for the remaining code.

- **Scatter Plots**

  Use Case:
  - Correlating the size of bad code with other metrics such as test scenarios absent or number of degrading code commits.

  Advantages:
  - Shows relationships between two variables.
  - Useful for identifying patterns and correlations.

  Example:
  - **X-Axis:** Test scenarios absent or another correlated metric.
  - **Y-Axis:** Size of bad code.
  - **Points:** Individual data points, possibly color-coded or scaled by size.

14. Test Effort

14.1. Definition

Testing effort can be measured at different levels of software development granularity, namely commits, pull requests, and releases. Considering software development as a continuous process, one can provide a prediction of test effort for subsequent commits, pull requests, or releases based on all the previous changes at any of these levels. This could enable a project manager to predict test effort for any of the three granularity levels. By analysing different types of granularities, we can get a more accurate picture of the amount of testing effort required. This is because different types of changes (e.g., commits, pull requests) can have different degrees of impact on the software and therefore require different levels of testing effort.

14.2. Importance

Test effort estimation is a crucial phase when estimating the time and cost of a software development project. Accurate test effort estimation is crucial for project success and minimizing costs in posterior fixes in the code’s maintenance phase of SDLC. In recent years, software effort estimation has become a challenging problem for the software industry, attracting research interest from academia and industry professionals. Part of the interest comes from the fact that lack of software testing impacts the quality of the
software overall.

14.3. Actions

Test effort can be estimated by analysing metrics from the development code at various code change granularities. The highest accuracy in classifying test effort was achieved for releases among the three used granularities (i.e., commits, pull requests, and releases). By using test effort in terms of CLOC, managers can be assisted in resource planning, budget management, and quality assurance. This information can help them make informed decisions related to testing.

Assuming that the estimation is valid, managers can allocate the right resources at the right time, avoiding delays and unnecessary costs. Also, inadequate testing could impact the general quality of the product. This can lead to quality issues that could be expensive to fix later. Accurate estimation ensures that sufficient time is dedicated to thorough testing early on, potentially leading to better quality.

14.4. Sources

We use three sources of data, namely:

Designite Java: This tool is used to analyse the quality of Java-based repositories to identify architecture, design, and implementation issues that could influence maintainability. Moreover, it computes various object-oriented metrics. Designite generates an analysis summary specific to the current version of the project. To evaluate each commit merged into the main repository branch, we extract feature values from this analysis summary.

JavaNCSS: This is an open-source command line tool that analyses Java-based repositories. It calculates noncommenting source statements, the number of packages, classes, functions, cyclomatic complexity, and many other Metrics.

GitHub API: This API is used to interact with GitHub repositories and get information about the commits, pull requests, and tags. This tool helps to detect the change of lines of code in each granularity.

14.5. Grouping Variables

We extract variables about all the commits, irrespective of whether they were merged in the main branch or not. We also extract variables for the commits that have been merged into the main branch. To get a unified dataset for each granularity, we include only the commits intersecting with all three data sources in the final dataset.

Variable set 1 contains the following information: Changes lines of code in each file, pull request number corresponding to each commit, and release tags and the associated commit. It also contains the names of the files changed in each commit, which we use to segregate the feature values of the test and development code based on the search string “test”. Only the feature values of the development code were used for training. The feature values of the test code are our target labels.

Variable set 2 contains non-commenting source statements, Cyclomatic Complexity, Javadoc comments etc., while variable set 3 contains architectural smells, design smells, and compute metrics. Feature sets 2 and 3 are exclusively used for training.

We return three datasets, one for each granularity, commits, pull requests and releases. Since the data from each of the three sources is extracted based on commits, we combined data from the commits corresponding to each pull request and release to prepare the final merged datasets for these specific granularities.

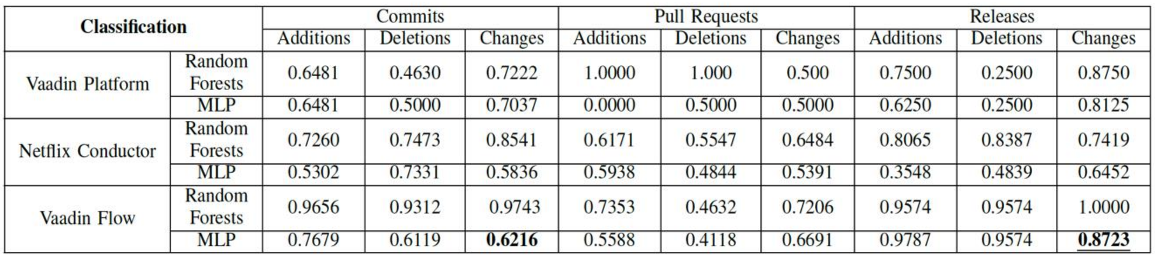
14.6. Correlated Metrics

Test effort seems to be correlated with change of lines of code metrics from the development code at various code change granularities. The highest accuracy in classifying test effort was achieved for releases among the three used granularities (i.e., commits, pull requests, and releases).

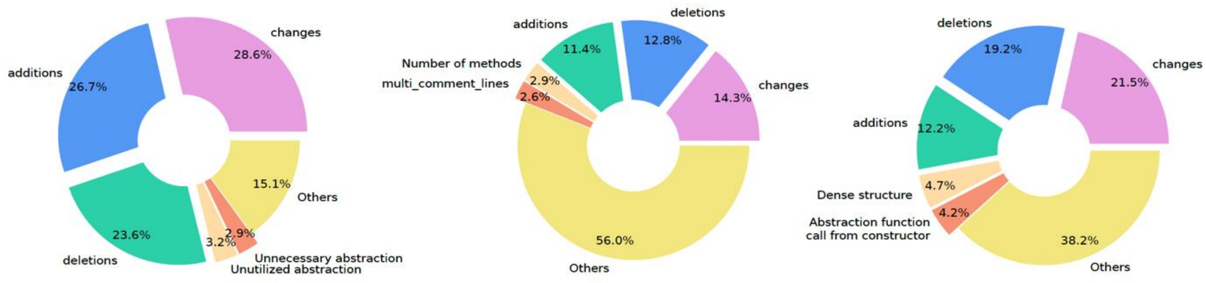
On an implementation level, test effort of a function is tied to the types of the inputs and outputs, as different types have a different impact on the number of equivalence classes or abstract domains.

14.7. Visualization

Example of visualizations for test effort estimation is shown below:



Classification		Commits			Pull Requests			Releases		
		Additions	Deletions	Changes	Additions	Deletions	Changes	Additions	Deletions	Changes
Vaadin Platform	Random Forests	0.6481	0.4630	0.7222	1.0000	1.000	0.500	0.7500	0.2500	0.8750
	MLP	0.6481	0.5000	0.7037	0.0000	0.5000	0.5000	0.6250	0.2500	0.8125
Netflix Conductor	Random Forests	0.7260	0.7473	0.8541	0.6171	0.5547	0.6484	0.8065	0.8387	0.7419
	MLP	0.5302	0.7331	0.5836	0.5938	0.4844	0.5391	0.3548	0.4839	0.6452
Vaadin Flow	Random Forests	0.9656	0.9312	0.9743	0.7353	0.4632	0.7206	0.9574	0.9574	1.0000
	MLP	0.7679	0.6119	**0.6216**	0.5588	0.4118	0.6691	0.9787	0.9574	**0.8723**



When visualizing metric data related to test effort estimation, tables and pie charts provide two distinct but complementary ways to present this information.
The tabular format allows a clear comparison of metrics (additions, deletions, changes) across different classification tasks and machine learning models:
- Group by Platform/Method: Organize the table so that each row corresponds to a platform and the method used (Random Forests, MLP).
- Metric Columns: Keep separate columns for each metric (Additions, Deletions, Changes) across different categories (Commits, Pull Requests, Releases).
- Highlight Key Values: Use bold or shading to highlight key values, such as the highest accuracy in each row, to make it easier for readers to identify top-performing methods.

Example Structure:

Platform	**Method**	**Commits Additions**	**Commits Deletions**	**Commits Changes**	**Pull Req. Additions**	**Pull Req. Deletions**	**Pull Req. Changes**	**Releases Additions**	**Releases Deletions**	**Releases Changes**
Platform 1	Random Forests	0.6481	0.4630	0.7222	1.0000	1.0000	0.5000	0.2500	0.7500	0.8750
Platform 2	Random Forests	0.7260	0.7473	0.8541	0.6171	0.5547	0.6484	0.8065	0.8387	0.7419
Platform 3	Random Forests	0.9656	0.9312	0.9743	0.7353	0.4632	0.7206	0.9574	0.9574	1.0000

Pie charts are particularly useful for visualizing the distribution of feature importances or classifications across different categories, as shown in the feature importance diagrams.

15. Test Percentage Advised/Test Scenarios Absent

15.1. Definition

The "test percentage advised" in a project refers to the ratio of test cases implemented in comparison to the number of test cases that are suggested to be implemented. Expressed as a formular: (n° of implemented test cases / n° of suggested test cases) * 100. “Test scenarios absent” refers to the total difference between implemented and suggested test cases, resulting in the number of test cases not yet implemented.

15.2. Importance

Indicates the test coverage of a software project. A low-test percentage and high number of test scenarios absent indicate a low test coverage that may lead to unidentified bugs, missing edge cases or unused code.

15.3. Actions

Refer to the test case suggestions to add missing test cases where they are useful. Add test cases for new features while developing the feature.

15.4. Sources

- Code repository
- UIBK’s SoHist

15.5. Grouping Variables

None.

15.6. Correlated Metrics

I. **Bad Code**

- **Correlation:** More test cases might lead to the identification of bad code sections.
- **Action:** Add missing test cases for uncovered code sections and use static analysis tools to identify bad code blocks.

15.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing the number/percentage of test cases between products.

  Advantages:
  - Easy to compare values across products.
  - Clearly shows trends and differences.

  Example:
  - **X-Axis:** Product.
  - **Y-Axis:** Test percentage advised/Test cases absent.

- **Line Charts**
  Use Case:
  - Tracking the trend of tests over time.
  - Monitoring the progress of test coverage improvement efforts.

Advantages:
  - Shows trends and changes over time.
  - Easy to identify peaks and troughs.

Example:
  - **X-Axis:** Time periods (days, weeks, months).
  - **Y-Axis:** Test percentage advised/Test cases absent.
  - **Additional Elements:** Separate lines for different products.

- **Pie Charts**

  Use Case:
  - Showing the proportion of implemented and suggested (not implemented) tests.
  - Visualising the distribution of test across different products.

  Advantages:
  - Provides a clear visual representation of proportions.
  - Easy to understand at a glance.

  Example:
  - One pie chart per product with one slice for the number of implemented test cases and one for the number of test cases absent.

- **Scatter Plots**

  Use Case:
  - Correlating the number of test cases absent with other metrics such as size of bad code.

  Advantages:
  - Shows relationships between two variables.
  - Useful for identifying patterns and correlations.

  Example:
  - **X-Axis:** Bad Code (possibly correlated metric).
  - **Y-Axis:** Test cases absent.
  - **Points:** Individual data points per product, possibly color-coded or scaled by size.

16. Defect Effort

16.1. Definition

"Defect Effort" refers to the software development effort spent on fixing bugs. This effort is measured based on the total number of changed lines of code (added, modified, and deleted lines) associated with defect resolution. It does not include Feature Effort (new feature development) or Enhancement Effort (refactoring and optimizations).

16.2. Importance

Tracking Defect Effort helps teams understand how much of their development capacity is dedicated to resolving software defects. A high Defect Effort percentage may indicate underlying issues with software quality, requiring improved testing and development practices.

A balanced Defect Effort ensures that defects are addressed without overwhelming feature development and enhancement efforts.

16.3. Actions

Identify trends in defect resolution effort to assess software quality over time.
Improve test coverage and quality assurance to minimize future defect resolution work.

Allocate sufficient resources to defect resolution without negatively impacting new feature development.

16.4. Sources

- Code Repository
- Issue Tracking System

16.5. Grouping Variables

I. **Product**

- **Definition:** The product or system that multiple repositories contribute to.
- **Importance:** Allows for a holistic view of Feature Effort across an entire product.
- **Examples:** Aggregated Defect Effort for a software suite with multiple services.

II. **Development Team**

- **Definition:** The team responsible for resolving defects.
- **Importance:** Helps analyze team efficiency and identify workload distribution.
- **Examples:** Comparison of Defect Effort across backend and frontend teams.

16.6. Correlated Metrics

I. **Feature Effort**

- **Correlation:** A high Feature Effort with increasing Defect Effort may indicate poor initial implementation quality.
- **Action:** Improve design and code reviews before new feature implementations.

II. **Enhancement Effort**

- **Correlation:** A low Enhancement Effort may indicate that code quality is not being actively improved, leading to technical debt.
- **Action:** Allocate time for codebase improvements to prevent recurring defects.

III. **Code Churn Rate**

- **Correlation:** A high churn rate may indicate unstable development practices leading to defects.
- **Action:** Implement stricter validation processes and improve developer collaboration.

16.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing Defect Effort across different teams or products.

  Advantages:
  - Easily compares values across categories.
  - Highlights discrepancies in Defect Effort allocation.

  Example:
  - X-Axis: Teams or products.

  - Y-Axis: Defect Effort (in total changed lines of code).
  - **Additional Elements:** Different colours to distinguish types of defects (e.g., critical vs. minor bugs).

- **Line Charts**
  Use Case:
  - Tracking Defect Effort trends over time.

  Advantages:
  - Shows fluctuations in Defect Effort across different time periods.
  - Helps detect patterns related to software stability.

  Example:
  - X-Axis: Time periods (weeks, sprints, months).
  - Y-Axis: Defect Effort
  - **Additional Elements:** Separate lines for different teams or projects.

- **Pie Charts**
  Use Case:
  - Showing Defect Effort as a proportion of total development effort.

  Advantages:
  - Provides a clear view of effort distribution.
  - Helps assess the balance between defect resolution, feature development, and enhancements.

  Example:
  - One pie chart per product, with slices for Feature Effort, Defect Effort, and Enhancement Effort.

- **Scatter Plots**
  Use Case:
  - Correlating Defect Effort with other metrics, such as feature effort or code churn rate.

  Advantages:
  - Helps identify trade-offs between feature development and software quality.
  - Useful for detecting inefficiencies in the development process.

  Example:
  - X-Axis: Code churn rate or another correlated metric.
  - Y-Axis: Defect Effort
  - **Additional Elements:** Individual data points representing different teams or sprints, color-coded by project phase.

17. Feature Effort

17.1. Definition

"Feature Effort" refers to the software development effort spent on developing new features or making improvements to a project. This effort does not include effort spent on fixing defects (Defect Effort) or improving code quality through refactoring and optimizations (Enhancement Effort). Feature Effort is measured based on the total number of changed lines of code (added, modified, and deleted lines) rather than time spent.

17.2. Importance

Tracking Feature Effort helps teams understand how much of their development capacity is dedicated to delivering new functionality. A well-balanced allocation ensures continuous innovation while maintaining code quality and stability.
A high Feature Effort percentage indicates a strong focus on expanding functionality, while a lower percentage may indicate that teams are overwhelmed with bug fixes or technical debt reduction.

17.3. Actions

- Ensure a balanced distribution between Feature Effort, Defect Effort, and Enhancement Effort to maintain software quality.
- Monitor trends over time to identify whether teams are spending excessive effort on non-feature work.
- Use Feature Effort data to align development priorities with business goals and ensure that engineering resources are effectively utilized.

17.4. Sources

- A. Code Repository
- B. Issue Tracking System

17.5. Grouping Variables

I. **Product**

- **Definition:** The product or system that multiple repositories contribute to.
- **Importance:** Allows for a holistic view of Feature Effort across an entire product.
- **Examples:** Aggregated Feature Effort for a software suite with multiple services.

II. **Development Team**

- **Definition:** The team responsible for implementing features.
- **Importance:** Helps analyze team efficiency and identify workload distribution.
- **Examples:** Comparison of Feature Effort across backend and frontend teams.

17.6. Correlated Metrics

I. **Defect Effort**

- **Correlation:** A high Defect Effort may indicate that new features introduce significant bugs, leading to rework.
- **Action:** Improve testing and quality assurance to reduce post-release bug fixes.

II. **Enhancement Effort**

- **Correlation:** A low Enhancement Effort may indicate that code quality is not being actively improved, leading to technical debt.
- **Action:** Allocate dedicated time for refactoring and performance optimizations.

III. **Code Churn Rate**

- **Correlation:** A high code churn rate (frequent rewrites of recent code) may indicate inefficiencies or unstable development.
- **Action:** Improve planning and design before implementing new features.

17.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing Feature Effort across different teams or products.

  Advantages:
  - Easily compares values across categories.
  - Highlights discrepancies in Feature Effort allocation.

  Example:
  - X-Axis: Teams or products.
  - Y-Axis: Feature Effort (in total changed lines of code).
  - **Additional Elements:** Different colours to distinguish feature types (e.g., UI improvements vs. backend features).

- **Line Charts**
  Use Case:
  - Tracking Feature Effort trends over time.

  Advantages:
  - Shows fluctuations in Feature Effort across different time periods.
  - Helps detect shifts in development focus.

  Example:
  - X-Axis: Time periods (weeks, sprints, months).
  - Y-Axis: Feature Effort
  - **Additional Elements:** Separate lines for different teams or projects.

- **Pie Charts**
  Use Case:
  - Showing Feature Effort as a proportion of total development effort.

  Advantages:
  - Provides a clear view of effort distribution.
  - Helps assess the balance between feature development, bug fixing, and enhancements.

  Example:
  - One pie chart per product, with slices for Feature Effort, Defect Effort, and Enhancement Effort.

- **Scatter Plots**
  Use Case:
  - Correlating Feature Effort with other metrics, such as defect effort or code churn rate.

  Advantages:
  - Helps identify trade-offs between feature development and software quality.
  - Useful for detecting inefficiencies in the development process.

  Example:
  - X-Axis: Code churn rate or another correlated metric.
  - Y-Axis: Feature Effort

- **Additional Elements:** Individual data points representing different teams or sprints, color-coded by project phase.

18. Review Percentage

18.1. Definition

"Review Percentage" refers to the proportion of code changes that go through a review process before being merged into the main codebase. It is derived from the pull request (PR) review system and indicates how rigorously code is reviewed before integration. This metric can be measured at both the project and module levels.

18.2. Importance

Review Percentage is a key indicator of code quality control and development discipline. A higher review percentage typically correlates with fewer defects and better code maintainability. Conversely, a low review percentage may indicate rushed development, leading to increased defect rates and technical debt.

18.3. Actions

- Encourage peer code reviews to improve code quality and knowledge sharing.
- Establish review policies ensuring all code undergoes review before merging.
- Monitor review trends to detect gaps in code quality control and developer engagement.

18.4. Sources

- Code Repository
- Issue Tracking System

18.5. Grouping Variables

I. **Project**

- **Definition:** The software project being measured.
- **Importance:** Allows assessment of review processes across an entire project.
- **Examples:** Review Percentage for a microservices-based application.

II. **Module**

- **Definition:** A specific component or module of a project.
- **Importance:** Helps identify areas where review practices may need improvement.
- **Examples:** Review Percentage for frontend vs. backend modules.

18.6. Correlated Metrics

I. **Defect Rate**

- **Correlation:** A low Review Percentage may correlate with a higher defect rate.
- **Action:** Increase review requirements to improve defect detection before deployment.

II. **Bug Count**

- **Correlation:** Projects or modules with lower review percentages may have a higher number of reported bugs.
- **Action:** Implement stricter review policies and quality control processes.

18.7. Visualization

- **Bar Charts**
  Use Case:
  - Comparing Review Percentage across different teams, projects, or modules.

  Advantages:
  - Clearly highlights differences in code review adoption.
  - Helps identify teams or projects that need stricter review processes.

  Example:
  - X-Axis: Teams, projects, or modules.
  - Y-Axis: Review Percentage.
  - **Additional Elements:** Different colors to indicate levels of review percentage.

- **Line Charts**
  Use Case:
  - Tracking Review Percentage trends over time.

  Advantages:
  - Shows the evolution of code review practices.
  - Helps detect patterns in review adherence.

  Example:
  - X-Axis: Time periods (weeks, sprints, months).
  - Y-Axis: Review Percentage.
  - **Additional Elements:** Separate lines for different teams, modules or projects.

- **Scatter Plots**
  Use Case:
  - Correlating Review Percentage with defect rate or bug count.

  Advantages:
  - Helps identify relationships between code review rigor and software quality.
  - Useful for evaluating the impact of review policies.

  Example:
  - X-Axis: Defect rate or bug count.
  - Y-Axis: Review Percentage.
  - **Additional Elements:** Individual data points representing different teams, projects, or modules.

19. Summary

The "D5.6 SmartDelta Metrics Guideline" document details the approach to defining, measuring, and visualizing key performance indicators within the SmartDelta project. It begins by outlining the purpose and importance of metrics, emphasizing their role in monitoring project progress and outcomes.

The document categorizes metrics into various types, distinguishing between quantitative and qualitative measures. It provides detailed methodologies for metric collection, analysis, and interpretation, ensuring uniformity across project contributors. Additionally, it highlights best practices for visualization, specifically through the use of Vaadin charts, to enhance the clarity and accessibility of complex data.

Further sections delve into case studies and practical applications of the proposed metrics framework, showcasing real-world examples from project partners. Recommendations for continuous improvement and adaptability of the metrics system are also included, ensuring long-term applicability and relevance.

By following this guideline, SmartDelta stakeholders can leverage a standardized approach to metric evaluation, fostering improved decision-making and operational efficiency across the project lifecycle.