SmartDelta

Automated Quality Assurance and Optimization in Incremental Industrial Software Systems Development

D4.5 - SmartDelta Quality Optimization and Recommendation Methodology

Submission date of deliverable: November 30, 2024

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	WP4 partners
Version number	V1.0
Work package	WP4
Work package leader	Akramul Azim
Dissemination level	Public
Description *(max 5 lines)*	This deliverable reports on the different planned activities of WP4 for quality improvement, optimization, and recommendation methodology in SmartDelta.

Executive Summary

This document discusses the quality optimization and recommendation methodology of SmartDelta. Three different tasks are performed which are: software quality trend analysis and prediction, similarity analysis and reuse recommendation, and change impact analysis and prediction. These tasks lead to three key innovations areas which are novel ML-based anomaly and threat detection methods, automatic code analysis and change impact analysis approaches, and similarity analysis approaches and recommendations. More than 20 tools were developed in this work package for software quality optimization and recommendations.

Project Description

Acronym and Full-length Title

20023	SmartDelta
Program Call	ITEA 3 Call 7
Full-length Title	Automated Quality Assurance and Optimization in Incremental Industrial Software Systems Development
Roadmap Challenge	Smart engineering

Description

Software-intensive industrial systems are typically not designed and built from scratch for each new customer and order, but rather as increments over an existing product or as a modified version tailored for the needs of a particular customer, market, or region. Similarly, for a single product and considering a continuous integration/continuous delivery approach with frequent builds and commits, a system gets built incrementally and iteratively resulting in many intermediate builds and versions. However, far too often it is observed that as a system is being built and incremented with new features, certain quality aspects of the system begin to deteriorate. Therefore, it is important to be able to accurately analyse and determine the quality implications of each change and increment to a system, particularly in a continuous engineering context. To address these challenges, SmartDelta builds automated solutions for quality assessment of product deltas in a continuous engineering environment by providing smart analytics from development artifacts (e.g., source code, log files, requirement specifications, etc.,) and system execution, offering insights into quality improvements or degradation of different product versions, and providing recommendations for next builds.

Table of Contents

List of Figures

List of Tables

Document Glossary

Acronym	Definition
AST	Abstract Syntax Tree
AI/ML	Artificial Intelligence / Machine Learning
CI/CD	Continuous Integration / Continuous Delivery
CIT	Combinatorial Interaction Testing
CPaaS	Communications Platform as a Service
DataOps	Data Operations
DevOps	Development (Dev) and Operations (Ops)
EFP	Extra-Functional Property
FinTech	Financial Technology
FM	Feature Modelling
FR	Functional Requirement
FODA	Feature-Oriented Domain Analysis
IoT	Internet of Things
MBT	Model-Based Testing
MLOps	Machine Learning Operations
NFP	Non-Functional Property
NFR	Non-Functional Requirement
NLP	Natural Language Processing
OEM	Original Equipment Manufacturer
OVM	Orthogonal Variability Modelling
PaaS	Platform as a Service
PLE	Product Line Engineering
QA	Quality Assurance
QIP	Quality Improvement Paradigm
RCS	Rich Communication Services
RL	Reinforcement Learning
SPLE	Software Product Line Engineering
UC	Use Case
UCaaS	Unified Communication as a Service

1. Introduction

SmartDelta aims to build automated solutions for quality assessment of product deltas in a continuous engineering environment by providing smart analytics from development artifacts (e.g., source code, log files, requirement specifications) and system execution, offering insights into quality improvements or degradation both in and of different product evolutions, and providing recommendations for next builds. SmartDelta will develop solutions for automated trend analysis and build recommendation with respect to quality characteristics using AI/ML for pattern recognition, optimization, and fault prediction techniques.

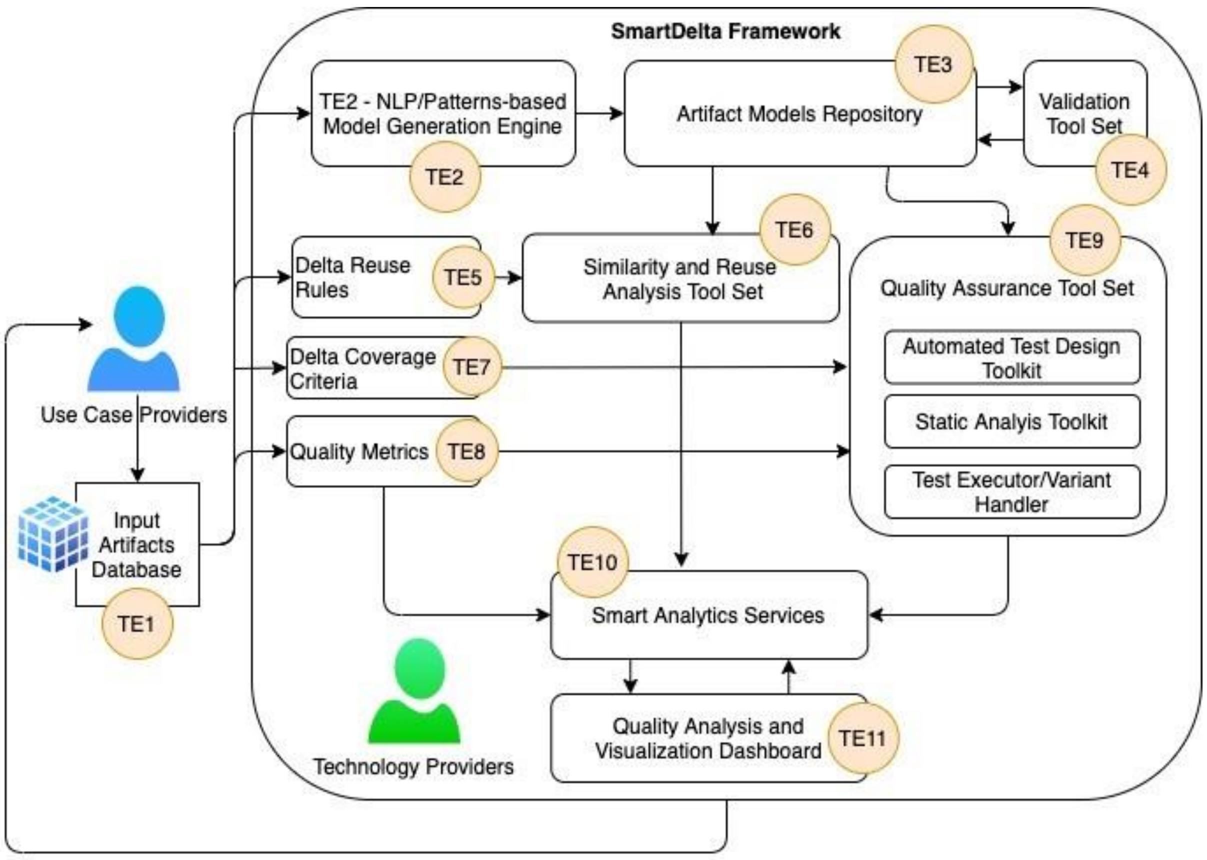

SmartDelta Framework
TE2 - NLP/Patterns-based Model Generation Engine
TE2
Artifact Models Repository
TE3
Validation Tool Set
TE4
Delta Reuse Rules
TE5
Similarity and Reuse Analysis Tool Set
TE6
TE9
Quality Assurance Tool Set
Automated Test Design Toolkit
Static Analyis Toolkit
Test Executor/Variant Handler
Use Case Providers
Delta Coverage Criteria
TE7
Quality Metrics
TE8
Input Artifacts Database
TE1
TE10
Smart Analytics Services
Quality Analysis and Visualization Dashboard
TE11
Technology Providers

Figure 1 : WP4 related entities: TE10 and TE11, will be analyzed through – Automated CI/CD Feedback Loops

Figure 1 shows WP4 related entities Smart Analysis Services (TE10) and Quality Analysis and Visualization Dashboard (TE11), smart analytics services and quality analysis and visualization dashboard. To achieve the goals of work package 4, it utilizes advanced AI and ML techniques to enable automated trend analysis, pattern recognition, fault prediction, and build recommendations. Key contributions include novel methods for anomaly detection and cybersecurity threat prioritization, which enhance operational stability and security in complex systems like micro-service architectures and telemetric environments. Additionally, automated tools for code analysis streamline maintenance, manage technical debt, and improve software reliability. Furthermore, our work advances similarity analysis and reuse recommendations, employing ML-driven techniques to identify reusable components and optimize software evolution. Moreover, tools for change impact analysis predict and evaluate the effects of changes on system quality, ensuring robust, scalable solutions for continuous integration and delivery. These contributions set a high standard for quality assurance and optimization in industrial software systems.

1.1. Project Context

Software is a dynamic entity that exhibits various quality characteristics as it undergoes updates over time. These enhancements not only contribute to its own development but also influence the operational environments in which it operates. Approaching software development with this awareness regarding quality can be a key factor for the long-term success of a company in the highly competitive market of industrial software-intensive products today. Far too often it is observed that as a system is being built and incremented with new features, certain quality aspects of the system begin to deteriorate. Therefore, it is important to be able to accurately analyse and determine the implications of each change and increment to a system, particularly in a continuous engineering context. This is, however, a complicated problem to solve because: i) most quality attributes are inter-dependent and cannot be addressed in isolation, for instance, adding more security features to a system can degrade its overall performance and also impact its energy consumption; ii) over time, companies end up having many different product versions and builds (including internal versions), tailored and customized for different customers, markets and regions, but each having different quality characteristics to analyse and test; iii) while at the same time, the size and complexity of the systems are also rapidly growing; iv) making the problem even more challenging under constant pressures to reduce development cost and time-to-market to be able to stay ahead of the competition.

To address the above challenges, SmartDelta will develop solutions for large-scale automated quality assurance and optimization in incremental development of industrial software-intensive systems. Towards this goal, SmartDelta will develop a set of tools and approaches as part of the SmartDelta framework in the following directions:

- Automated analysis solutions (e.g., based on AI/ML, model extraction, and pattern recognition) to identify and extract quality improvement or degradation trends from and across a set of existing/previous product versions and development artifacts.
- Techniques to identify the features, design decisions, and development artifacts causing quality degradation and deviation in a system.
- Static and dynamic verification and validation solutions, using techniques such as static code analysis, model-based test generation, test prioritization and selection, and mutation testing, to assess and ensure desired quality attributes of a system.
- Novel techniques for automated reuse analysis and design recommendation for next builds optimizing with respect to specific quality attributes such as performance.
- A set of innovative visualization solutions to illustrate software quality attributes, and their evolution and trend analysis results over different builds and versions.

Considering the relevance and importance of the project topic for a wide range of industries offering software-intensive products, the project has attracted and brought together various partners from different sectors and market domains with complementary expertise, knowledge, and technologies to develop the proposed solutions and verify their technology readiness levels. In particular, the consortium consists of a well-balanced mix of partners from Sweden, Germany, Canada, Turkey, Spain, and Austria, including industrial use-cases from the railway, telecommunication, logistics and mobility, FinTech and banking, cybersecurity, and enterprise software domains.

In this work package WP4, one of the technical activities is **Service Specification and Implementation**. In this activity, services for optimization as well as ML-based solutions for predicting quality trends, pattern recognition, and automated recommendations are specified and implemented. The challenges in this activity are to collect quality-related data, to analyze data and observe quality trends, to analyze trends and predict future trends, to recommend quality measures aligned with the observed trends, and to provide recommendations for next and updated builds to improve quality. Trends shall include development as well as operation (DevOps; feedback of operational data into development), intermediate products as well as releases (CI/CD), and functional as well as extra-functional quality characteristics.

Another technical activity of this work package, which is also related to WP5, is **Tool Set and Service Dashboard Integration.** The main goal of this activity is to provide a turnkey framework for rapid quality assessment of product deltas through advanced automated tool sets. The challenge is software integration and connectivity of different tools, developed by different groups, with various interfaces and intermediate representations of artifacts. The resulting integrated dashboard shall be applicable and useful for different end-users, such as industrial system providers, developers, testers, and infrastructure operators, and shall provide a uniform view on quality, the evolution of quality, pinpointing to potential problems and issues, and making recommendations for improvement.

1.2. Tasks

The following table outlines the different tasks of this work package.

Table 1: SmartDelta WP4 tasks

Task	Task Name	Task Description
T4.1	Software quality trend analysis and prediction	Task 4.1 will develop automated solutions to extract software quality trends in terms of degradation or improvement of different quality characteristics across different versions and builds
T4.2	Similarity analysis and reuse recommendation	In task T4.2, automated solutions to perform similarity analysis with the purpose of identifying similar software artifacts across a range of product versions are developed. Moreover, based on the results of the similarity analysis, reuse recommendations will also be provided for i) selecting design

Task	Task Name	Task Description
		artifacts and component that can be reused for the next builds to achieve desired levels of quality characteristics in the system, as well as ii) test cases that can be reused across different product versions.
T4.3	Change impact analysis and prediction	This task will build solutions to automatically determine how a change in the software has affected its quality characteristics and provide predictions on possible impacts of the upcoming changes.

1.3. Use Cases

The use cases and summary in the SmartDelta project are as in Table 2.

Table 2: SmartDelta use cases

Use Case ID	Country	Partner	Domain	Topic
UC1	Sweden	Alstom	Railway	Quality in agile model-based system and product line engineering
UC2	Germany	AKKA	eMobility	Charging communication controller software for electrical vehicle
UC3	Canada	eCAMION	eMobility	High quality and cybersecure software in deployable energy hubs
UC4	Turkey	NetRD	Telecommunication	AI based fault and performance analysis in cloud communication services
UC5	Turkey	Kuveyt Türk	Banking and Finance	Continuous improvement of code quality, security and performance in core banking software
UC6	Germany	Software AG	Enterprise Software	Continuous security and quality improvement in enterprise software
UC7	Austria	c.c.com	Logistics and Personal mobility	Continuous quality monitoring & improvement in automated traffic detection software
UC8	Canada	GlassHouse	Cybersecurity	Continuous improvement of cybersecurity solutions

Use Case ID	Country	Partner	Domain	Topic
UC9	Spain	Izertis and UC3M	Digital and IT	Semantic Matchmaking
UC10	Finland	Vaadin	Software development platform	Continuous quality, security and performance improvement in software development platform
UC11	Turkey	Arcelik	Home Appliances	Measure code quality and performance in employees’ single point of solution: connecta

1.4. Functional and Non-Functional Requirements

Business requirements define the scope of the solution, what a company needs and its objectives, while functional requirements deal with how the company will achieve it.

Functional requirements (FRs) help to understand why the application exists in the first place. In other words, what business problem does it solve? More to the point, what is it originally designed to do? When you analyse an application with a focus on how the application goes about solving their business problem, you will end up analysing its functional requirements. Functional requirements are the things that the application absolutely must do. In this project, all business and user requirements are handled as functional requirements.

Non-functional requirements (NFRs) are requirements that may not necessarily need to be met for the application to function (i.e., functional correctness), but define the quality of services and functionalities that a system is expected to provide. NFRs define system attributes such as security, reliability, performance, maintainability, scalability, and usability. They serve as constraints or restrictions on the design of the system across the different backlogs and subsystems.

2. Background and Literature Review

2.1. Software quality trend analysis and prediction

Software quality metrics are integral to understanding and enhancing a wide range of project-related aspects of the development lifecycle if we want to manage all the elements effectively and assess the product's quality before release.

Managers use metrics as valuable tools to enhance their understanding of the production process. While metrics alone may not directly improve development, they serve as a powerful means to illustrate the current state of the project by offering insightful statistics for each process step. This information equips managers with the necessary insights to identify potential issues and implement effective solutions, ultimately contributing to the overall improvement of the project.

Software development is a complicated and multifaceted process. It involves many different tasks and activities. We will evaluate some of the most important metrics to help you assess the success of the software development life cycle.

According to the IEEE, software quality metrics are [1]:

(1) A quantitative assessment of the extent to which a specific quality attribute is present in each item.
(2) A function that takes software data as inputs and produces a single numerical result that may be used to represent how much a specific quality feature is present in the software

2.2. Importance of Software Quality Metrics

It would be wiser to start with their objective before delving into the IT world and all its code quality criteria. Why is it necessary to use these technologies at all? Let's examine the significant benefits of software metrics in more detail [2]:

Productivity: Fast data processing is an application's most valuable feature. The better, the less time it needs to do the job. Some indicators aid in boosting and monitoring the project's productivity and resolving pressing problems.

Creating decisions: These indicators can be helpful when determining how decisions were influenced. Project leaders can organize goals and priorities while avoiding rash decisions. It enables them to meet the objectives of software quality assurance, optimize the project, and make informed concessions.

Sorting data: Metrics can be used in complex projects to lessen misconceptions and ambiguities. You can obtain unbiased information by using the software organization.

Priorities: Managers will no longer struggle to track, recognize, or order the project's problems without measurements. All levels of a company can communicate with them.

Progress control: Is the project finished on time? How is everything going? Controlling the work's progress and the outcome is crucial, and you should always have the answers. These metrics display the software product's status, quality, and modifications.

Management approach: Some hazards require direct estimation, management, and prioritization. Metrics assist in managing such problems and preventing further expensive remedies. In addition, they help with management tactics, identify faults, and fix technical aspects of the project.

There are different categories of metrics. Some example categories of metrics are:

1. Process metrics
2. Product metrics
3. Project metrics
4. Production metrics
5. Security response metrics
6. Traditional metrics
7. Object-oriented metrics

Below, a detailed discussion of each of the above metrics is listed:

2.2.1. Process Metrics

Process metrics [3] make the Software Development Life Cycle (SDLC) more efficient. Process metrics measure various aspects of software development. One good example of this metric is the duration of time that the process of software creation tasks.

2.2.2. Product Metrics

Product metrics are software product measures at any phase of its development, from requirements to the installed system. Product metrics define the product's attributes, such as size, code complexity, design aspect, performance, and quality level.

Line of Code: This simple metric is used to calculate the software size, including any line of program text, excluding comments or blank lines. By utilizing this, one can measure the productivity of programmers.

Token Count: A software can be considered a collection of either operators or operands (also known as a token). A token can be used as a metric.

Function Count: Software can be better interpreted as a collection of a larger unit called a function or module. Modules can be compiled independently. For example, if the software requires n modules. We can say that the module size should be about fifty to sixty code lines. Hence the software is about n x 60 lines of code.

McCabe's Cyclomatic Metric: McCabe presented a software program as a set of a connected directed graphs consisting of nodes and arcs. The nodes, parts of the code, do not have any branches, while arcs represent the control flow when the program runs. The complexity of software can be associated with the topological density of a graph.

Stetter's Program Complexity Measure: Stetter's metric looks into the data flow and the program's control flow, which may be viewed as a sequence of declarations and statements.

2.2.3. Project Metrics

The software team adjusts project workflow and technical operations using software project metrics. Project metrics prevent delays in the development timeline, reduce potential risks, and continuously evaluate the quality of the final result. Every project should evaluate its resources, deliverables, and outcomes (effectiveness of deliverables).

For example, in a software development project aimed at creating a new e-commerce website, project metrics are used to track key indicators such as website load times, conversion rates, and customer feedback. By analyzing these metrics, the team can identify and address performance bottlenecks, improve user experience, and optimize the site's overall quality and effectiveness.

2.2.4. Production Metrics

This metric estimates the developers' productivity, speed, and quantity of completed work. We can check production metrics by using the number of active days, failure and debugging times, productivity, task scopes, and other factors.

Active days: During this time, developers write and iterate over code. It does not include any additional minor tasks, like planning. Finding hidden costs is made simpler by these metrics.

Failure and repair time: Errors and defects cannot be completely eliminated when making a new product. As a result, all you can do is monitor the amount of time the engineers spend searching for a solution.

Productivity: Although measuring this aspect is difficult, each developer's code volume can serve as a guide.

Task scopes: An annual maximum of this amount of code can be produced by a developer. It may seem odd but knowing how many engineers a project will need is helpful.

Code turnover: Chaos in the code represents the proportion of the product's code that has been altered. B dividing the frequency of application failures (F) by the frequency of usage (U), the application crash rate (ACR) can be calculated = F/U.

2.2.5. Security Response Metrics

These metrics aim to ensure product safety, as the name suggests. When evaluating the quality of your software, you should pay attention to how well it handles security. Due to the increasing frequency of hacking attacks, this stage is crucial. It is essential to observe how quickly the project can either notify the IT manager of the issue or find a solution [9].

Endpoint incidents: the number of devices affected by a virus or other security threat over a given time frame.

Mean time to response (MTTR): the interval between discovering a security event and taking corrective action.

Mean Time to Detect (MTTD): It measures the time interval between the occurrence of a security event and its successful detection. A shorter MTTD indicates a more efficient and timely identification of security incidents, enabling quicker response and mitigation actions.

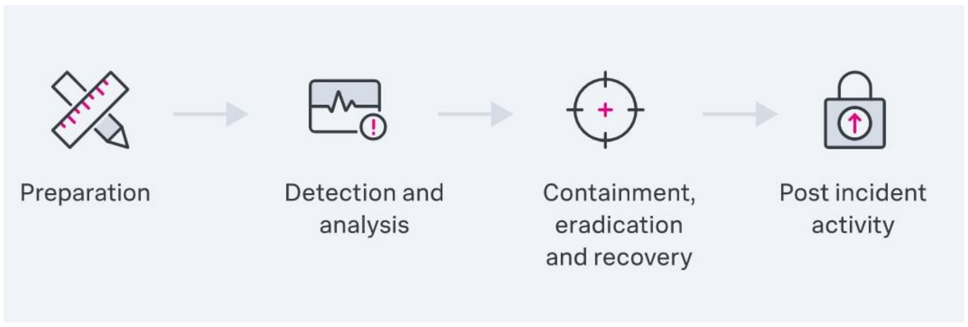


Figure 2 : Incident response Lifecycle

2.2.6. Defect and version control metrics

The primary determinant of how well-made a piece of software is its defect count. It contains:

- Stages of the flaws' emergence
- How many defect reports were there
- The number of defects per code line (density)
- The number of defects per code line (thickness)

Version control system (GitHub) requests might demonstrate project complexity, pull request involvement, and team communication. The following indicators are part of quality control for software development:

- Pull requests that failed the testing process
- Breaking pull requests for the build
- The number of requests that were merged and rejected
- The number of comments on pull requests
- They shouldn't be excessively large or little. However, these indicators rise the more complex the software gets.

From the above analysis, we have created a metrics reporting template as follows:

2.3. Metrics Reporting Template

- **Product metrics:** Measures certain characteristics of the software, such as size, complexity, design features, performance, and quality level.

- **Process metrics:** These metrics can be used to expand the development and maintenance activities of the software.
- **Project metrics:** These metrics define the project characteristics and execution.
- **Production metrics:** These types of metrics measure the amount of work completed and determine the efficiency of software development teams.
- **Security response metrics:** They are used to check how security, operations, and development teams are countering to security issues for each application supported. Applications that have low-security metrics may have underlying quality issues.
- **Defect and Version control system Metrics:** The primary determinant of how well-made a piece of software is its defect count.
- **Traditional metrics:** Such as Line of code (LOC), Cyclometric complexity, etc.

2.4. Software Quality Analysis Prediction

The quality of software plays a pivotal role in the successful deployment of software products. However, software developers face significant challenges in predicting a product's quality before it is tested in real-world scenarios. As highlighted in the literature, research in software quality prediction remains relatively limited [4].
Machine learning approaches have emerged as effective solutions for forecasting software quality. These approaches not only enhance the accuracy of quality predictions but also aim to minimize the developer’s workload by providing early warnings during the software development lifecycle. Early detection of potential quality issues can significantly reduce downstream costs and improve the overall efficiency of the development process.

To achieve this, machine learning methods can be broadly categorize into **supervised** and **unsupervised** learning techniques, each addressing different aspects of quality analysis:

- **Supervised Learning Techniques:**
  These methods leverage labeled data to train models for predicting software quality attributes. By learning from historical data, supervised techniques can predict the likelihood of defects, identify problematic modules, and classify software components based on their quality. Examples include:
- **Neural Networks:** Used for pattern recognition and mapping relationships between inputs and outputs.
- **Bayesian Networks (BN):** Capture probabilistic dependencies between software metrics and quality attributes.
- **Decision Tree Techniques:** Provide interpretable models to identify quality issues based on logical conditions.
- **Unsupervised Learning Techniques:**
  Unsupervised methods are particularly useful when labeled data is scarce or unavailable. These techniques identify hidden patterns or anomalies in software metrics that could indicate potential quality issues. Examples include:
- **Fuzzy Logic:** Handles uncertainty in software metrics and offers insights into vague or imprecise quality data.
- **Genetic Algorithms (GA):** Optimize software quality predictions by simulating evolutionary processes.
- **Case-Based Reasoning (CBR):** Uses historical cases to provide recommendations for similar quality concerns in new software.

- **Large Language Models (LLMs):**
Emerging techniques like LLMs (e.g., GPT models) are being explored for software quality analysis. These models excel in understanding natural language and structured code data, enabling tasks like defect prediction, code review automation, and generation of quality improvement suggestions.

2.4.1. Neural network-based software quality prediction models

Neural networks are versatile models that can be used for various software quality prediction tasks. They can analyze large and complex datasets, making them suitable for tasks like defect prediction and code quality assessment. Neural networks can capture intricate patterns and relationships in the data, providing accurate predictions when trained on ample quality-related data.

For example, Karunanithi et al. [6] propose a neural network model for reliability prediction. The failure history is used as the basis for the model's internal failure prediction model, which adapts to the software model's complexity. Network training is required for this. Using the software's error history to alter the strength of neural connections in this process.

2.4.2. Bayesian network for predicting software quality models

The Bayesian Network method for predicting software quality relies on activity-based quality models, simplifying complex concepts into precise definitions based on relevant facts [7]. To create a Bayesian Network (BN) for software quality assessment and prediction, a four-step process is followed:

(1) **Development of Activities:** The first step involves defining activities based on goals and the associated indicators used to measure these activities.
(2) **Identification of New Criteria:** Using the quality model, new criteria connected to the activities are identified. This step helps in refining the quality assessment.
(3) **Incorporation of Quantitative Data:** Quantitative data related to software quality is integrated into the BN by adding additional nodes for each fact and its associated activity node. This allows for breaking down complex quality models into tangible definitions, enhancing the model's ability to access and predict quality.
(4) **Accessing and Predicting Quality:** The resulting BN is used to access and predict software quality by searching for operations that enable it to forecast quality effectively.

One specific application of BN in software quality prediction is for Extreme Programming (XP) process success/failure prediction, which is particularly relevant in iterative software development approaches like XP [13]. In XP, traditional software requirement specification documents are replaced with user stories, and each user story is developed during a single iteration. However, XP projects often face challenges in predicting software quality accurately. The mathematical model proposed by Abouelela and Benedicenti for XP procedures based on BNs has several noteworthy characteristics:

- **Succession of Releases:** The model accounts for the iterative nature of XP projects by considering the succession of releases.

- **Completion Time Prediction:** It accurately predicts the expected completion time, a critical factor in determining project success or failure.
- **Development Velocity Tracking:** The model enhances development velocity by tracking it daily in terms of user story points. This prediction can be made during the planning phase, significantly before the actual development begins.
- **Defect Rate Calculation:** To forecast process quality, the model calculates the defect rate for each release, providing insights into software quality. Results from two distinct case studies demonstrate the model's ability to forecast project completion time effectively.

Bayesian Networks offer a structured approach to predict and assess software quality by breaking down complex models into manageable components. Their application in predicting software quality in iterative development processes like Extreme Programming has demonstrated the ability to improve project planning, predict completion times accurately, and enhance development velocity.

2.4.3. Models for using Genetic Algorithm to forecast software quality Identifying defective modules

Genetic Algorithms (GAs) emerge as a valuable tool for not only locating faults but also identifying their root causes and, notably, predicting defects. GAs operates as problem-solving algorithms inspired by genetic principles. The GA approach, as proposed by Puri and Singh [12], is particularly relevant for fault discovery in open-source software and encompasses several key steps. Firstly, it involves gathering raw data from the source code of an open-source software system. Subsequently, the gathered data undergoes evaluation utilizing a metrics suite, which includes various metrics such as coupling between objects (CBO), lack of cohesion (LCOM), and others. Following this, the relevant metrics for fault prediction are refined through data filtering. The GA process is then applied to work with the reduced data or attributes, leveraging genetic principles to explore potential fault factors. Finally, confusion metrics are employed to evaluate the model's predictive performance comprehensively. This comprehensive approach enables software developers and analysts to effectively harness Genetic Algorithms to identify and rectify faults within open-source software systems, ultimately contributing to enhanced software quality and reliability.

2.4.4. Fuzzy logic for software quality prediction Models:

To evaluate the quality of software, numerous models for software quality assessment have been put forth by multiple authors, each considering a different software metric. However, these models are always missing two crucial elements that, from the implementer's perspective, could increase the transparency of the quality model. These quality elements are imprecise professional linguistic understanding and precise numerical quantitative knowledge from the historical dataset. Ahmed and Al-Jamimi et al. [15] develop a fuzzy-based transparent model for software quality assessment using both these knowledge forms. The main focus of the model is maintainability prediction. Here, the authors use the Mamdani fuzzy inference model that performs satisfactorily better than any other machine learning technique.

2.4.5. Software quality estimation using Case-based reasoning (CBR)

Case-Based Reasoning (CBR) in software quality estimation relies on past cases or experiences stored in a repository. When a new quality estimation task arises, CBR retrieves relevant past cases, assesses their similarity to the current problem, adapts their insights, and generates an estimation based on historical knowledge. This approach benefits from human experts who can guide the relevance and adaptation process, making it a valuable tool for predicting and improving software quality by drawing from real-world experiences. For example, Rashid et al. [11] employ CBR to evaluate software quality, with human experts performing the estimation task. This method considers the resemblance between previous projects established by the primary quality characteristics.

2.4.6. Decision tree algorithm for software quality Classification

Decision tree techniques, including random forests and gradient boosting, offer interpretability in software quality analysis. These models are capable of identifying the most crucial factors contributing to software quality issues. Decision trees are often employed for feature selection, allowing developers to focus on critical aspects of quality improvement. Their transparency makes them a valuable choice when understanding and explaining the factors impacting software quality is essential.

For example, Using the SPRINT decision tree algorithm, Najafabadi , Khoshgoftaar and Seiya [10] explicitly provide a thorough investigation on calibrating classification trees that help estimate software quality. Additionally, this strategy effectively overcomes the memory constraints that prevent a faster and more scalable analysis for several other classification algorithms. As an extension of the decision tree algorithm CART, it investigates a novel approach to tree pruning based on the minimum description length (MDL) approach. SPRINT's modified CART algorithm and MDL principle enable it to provide precise quality predictions. Large telecommunication systems are used for the case study implementation, and defect data from four different system versions are collected along with pertinent software metrics. According to the authors' observations, SPRINT can produce classification trees that are more evenly distributed and stable than those produced by the CART classification algorithm.

2.4.7. Large Language Model (LLM)

Large Language Models (LLMs), exemplified by GPT-3 and its successors, operate through a two-step process: pre-training and fine-tuning. During pre-training, LLMs learn language patterns, grammar, and common linguistic structures by predicting the next word in a sentence, using vast amounts of text data. In the fine-tuning phase, they specialize in specific tasks, adapting to the domain in question. When it comes to software quality prediction, fine-tuning would involve training the model on software-related datasets encompassing code repositories, bug reports, user feedback, and documentation.

LLMs bring distinctive advantages to the field of software quality prediction. Their natural language understanding capabilities enable them to analyze textual data effectively, identifying patterns, sentiments, and potential quality issues within unstructured text. They can also assist in generating documentation by summarizing code changes, explaining intricate algorithms, or even automatically producing user-friendly documentation for software products. Additionally, LLMs excel in predictive analytics, capable of forecasting potential quality issues by examining historical data, recognizing trends in code changes, and anticipating potential bugs or performance bottlenecks.

Contrasting traditional machine learning models, LLMs operate differently. They require extensive pre-training on large text corpora, whereas traditional models rely on structured datasets with explicitly labelled features. LLMs are highly adaptable, requiring minimal task-specific fine-tuning, while traditional models necessitate more feature engineering and specialized tuning for each task. However, it's important to note that LLMs are less interpretable than traditional models due to their complex neural network architecture.

To incorporate LLMs into existing software quality prediction systems, one typically starts with fine-tuning the model on a specific software quality prediction task, utilizing relevant datasets from the software development domain. Integration can be achieved through the development of an API or interface that enables seamless interaction between the LLM and the existing prediction system. Regular updates and continuous fine-tuning are essential to ensure the LLM adapts to evolving software development trends and maintains its effectiveness.

Evaluating the performance of an LLM-based software quality prediction system involves a combination of traditional machine learning metrics and domain-specific measures. Metrics such as accuracy, precision, recall, and F1 score provide insights into the model's overall predictive performance. However, it's equally important to introduce domain-specific metrics like bug detection rate, code review efficiency improvement, and user satisfaction to assess the model's real-world impact in the software development context. A/B testing can further compare LLM-based predictions with traditional methods to determine the model's efficacy in practical scenarios.

Incorporating LLMs into software quality prediction systems has the potential to enhance accuracy, adaptability, and insights, ultimately contributing to more effective software development processes and improved user satisfaction. Careful fine-tuning, evaluation, and ongoing monitoring are critical to ensure that LLMs deliver their promised benefits in the specific context of software quality assessment.

LLMs are transforming CI/CD pipelines by enabling smarter, automated workflows in software testing and quality assurance. These models enhance pipeline efficiency by analyzing code changes, predicting potential defects, and generating automated test cases. LLMs can assist in identifying patterns from historical build failures, suggesting fixes, and optimizing test case prioritization. By integrating LLMs into CI/CD workflows, organizations can ensure faster feedback loops, reduce manual intervention, and improve the overall reliability of software deployments, making them invaluable in modern, fast-paced development environments.

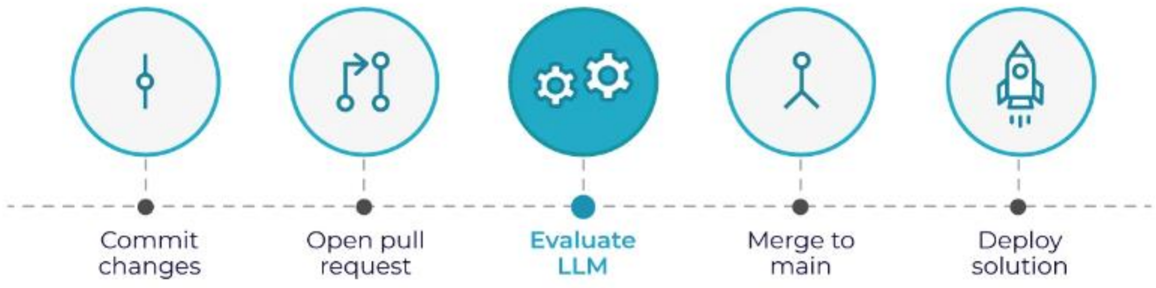


Figure 3 : CI/CD pipeline with LLM Integration

2.5. Similarity analysis and reuse recommendation

Similarity refers to the degree of closeness on relevant dimensions, features, and/or characteristics. Typically, for similarity analysis, different similarity measures are used that quantify the degrees of closeness between two elements. Similarity measures are used mainly in recommender systems for ranking potential recommendations to support various steps in the software development and maintenance lifecycle. As most of the measures are applied to quantify the degree of closeness on a scale of features and dimensions, often an intermediate representation is used for computing the similarity. Therefore, this section also summarizes the different representation that enable similarity analysis.

To apply similarity measures to textual input often the text is converted into representation vectors for better computation of similarity. The representation vectors are computed using different approaches ranging from lexical to embeddings. Approaches based on lexical features often use term frequency matrix-based extraction of the feature vectors. Seminal approaches include the bag of words and term frequency-inverse document frequency (tfidf). Bag of word-based vector extraction often uses the frequencies as values for the features. On the other hand, tfidf also considers inverse document frequencies of terms for enriching the feature vectors. Other approaches focus on representing textual input using embeddings, often extracted as weights of a representation learning-based neural network's layer. This is often done by mapping textual token to real numbers using the probability of relatedness of the words in the dataset. Seminal architectures for embeddings include the skip gram neural network [63], recurrent neural networks [64] and transformer-based neural networks [65].

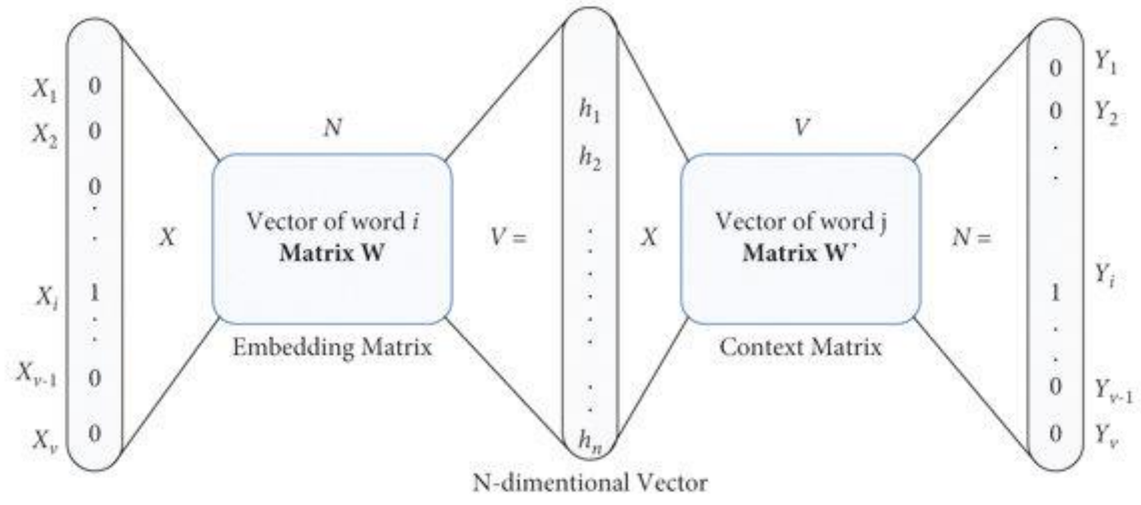


Figure 4: Word2vec embedding architecture with the skip-gram model[1].

Similarity metrics: Various similarity metrics could be applied to the representation vectors for quantification of the degree of closeness. The most seminal ones are summarized below.

Edit distance is a class of similarity metrics based on the dissimilarity between input [66]. Edit distance-based metrics could be applied to both raw text or its representation vector and works based on quantifying the number of edits required to make the two-input similar. A widely used metric based on edit distance within software engineering is Levenshtein distance.

Jaccard Similarity Index (JSI) also works both on the raw text of the representation vector. However, unlike edit distance, JSI is based on the ratio of common terms over their union.

Cosine similarity metric is based on the cosine angle between two vectors across multiple dimensions.

Euclidean distance is a metric based on the space in length between two points on one or many dimension(s).

Below, we briefly summarized the different approaches that are leveraging similarity to aid different software engineering tasks.

The text-based similarity is often leveraged in requirements engineering for reuse recommendation [67], change impact analysis, and tracing [68]. These recommenders are based on the assumption that similarity in one domain (e.g., requirements) could be used as a proxy for similarity in the other domain (e.g., software) [69,70]. A typical recommender

[1] Efstathiou, V., Chatzilenas, C., & Spinellis, D. (2018, May). Word embeddings for the software engineering domain. In *Proceedings of the 15th international conference on mining software repositories* (pp. 38-41).

usually retrieves the most similar artifact (with a link to an artifact in another domain) to the query and uses that as a case to recommend reuse or relevance in the other domain.
In addition, the textual similarity is also leveraged in identifying code clones [71], feature similarity [72], and feature model extraction [73].

2.6. Change impact analysis and prediction

2.6.1. Change Impact Analysis

“Change Impact Analysis (CIA) is the process of exploring the tentative effects of a change in other parts of a system. CIA is considered beneficial in practice, since it reduces cost of maintenance and the risk of software development failures.” [16] In other words, it is the process of inspecting the undesired consequences of a change in a software module.[17] Change impact is a significant matter in software and programming since several innovations are made over time, which may result in negative consequences if not checked and not determined by the changes in the software. A tiny change in software can cause devastating impacts, and it may be challenging to determine the affected function(s). Therefore, it should be required to examine which parts of the software are affected during the software maintenance process and their impacts.

To determine the change set in which the components might be impacted by the change request, CIA first analyzes the source code and the change request. Other components potentially impacted by the items in the change set are then estimated using the change impact analysis approach. The set that results is known as the estimated impact set (EIS). The items in the actual impact set (AIS) are updated after the change is put into practice to fulfil the change request. Because changes may be executed in various methods, the AIS is not always the best option for change requests in practice [18].

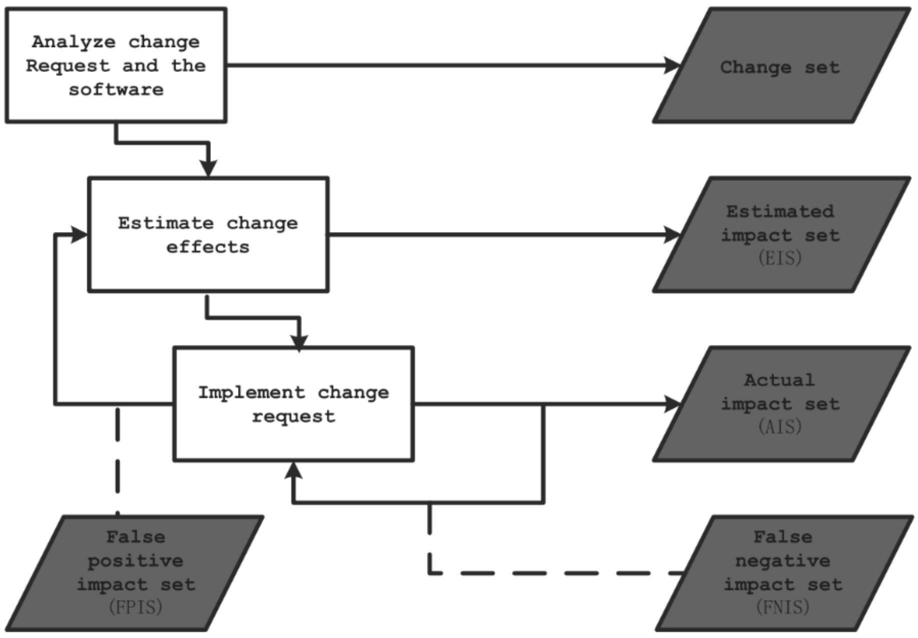

Analyze change Request and the software
Change set
Estimate change effects
Estimated impact set (EIS)
Implement change request
Actual impact set (AIS)
False positive impact set (FPIS)
False negative impact set (FNIS)

Figure 5: Change impact analysis process. © Diamani et al. [3]

The software change impact analysis process is an iterative one, as seen in Figure 5. And when a modification is put into practice, some more impacted aspects that are not in the EIS could be found. The false-negative impact set (FNIS), which reflects an underestimation of effects, is a collection of similar items [19, 20]. The false-positive impact set (FPIS), which denotes an overestimation of impacts in the study, is a set that is typically included in EIS but does not actually need to be changed or redone [19, 20]. These four sets are connected in the following ways:

$$(EIS + FNIS) - FPIS = AIS$$

Estimating an EIS that is as near to the AIS as feasible is the aim of the CIA method. To assess the precision of the impact analysis process, several metrics may be established [21, 22]. Precision and recall are two often utilized measures for CIA method accuracy [23, 24, 5]. They were often applied in a situation involving information retrieval [9]. Precision and recall are defined as follows in the CIA scenario:

$$\text{Precision} = |EIS \cap AIS| / |EIS|$$
$$\text{Recall} = |EIS \cap AIS| / |AIS|$$

While recall gauges how well the EIS accounts for actual changes, precision gauges how well the projected consequences line up with the actual impacts caused by modifications. Maintainers will take less time finding and apply the adjustments using a high-precision EIS. Maintainers are certain that all of the effects of those suggested adjustments will be taken into account because of the high recall EIS.

The CIA method begins with the determination of change demands and all the affected parts, which are referred to as the 'Change Set'. There are several techniques for classifying the change set. The Estimated Impact Set is designed for identifying differences in the software, and it is estimated by using several CIA techniques. The Actual Impact Set is constructed to detect the position of the changes made in the software. Also, another two impact sets, namely, False Negative Impact Set for indicating underestimation and False Positive Impact Set for indicating overestimation of effects are designed. The aim here is to guarantee that the Estimated Impact Set and the Actual Impact Set are the same [18].
Having equal sets of an Estimated Impact and Actual Impact is the primary goal of the CIA; however, it is challenging to implement. By cautiously choosing suitable CIA techniques, this goal can be achieved. There are diverse metrics to examine the precision of CIA techniques, but Precision and Recall are the ones that are used generally. Precision is about what degree Estimated Impact Sets overlap with Actual Impact Sets uncovered by the changes. Recall is about what degree Estimated Impact Set encompasses the real changes in the software [18].
If the Estimated Impact Set has high precision, it signifies that defining the position of the changes and accomplishing the changes take less time. If the Estimated Impact Set has his recall, it indicates that the effects of these suggested changes will be taken into account.

There are two main CIA techniques: Traceability based CIA, and Dependence Based CIA.

2.6.2. Traceability Based Change Impact Analysis

In this analysis, in order to grasp the potential impact of presenting a change in software, some elements like design, code, documents, test cases, etc., are defined and evaluated. This type of analysis focuses on exploring connections between software elements. [18]

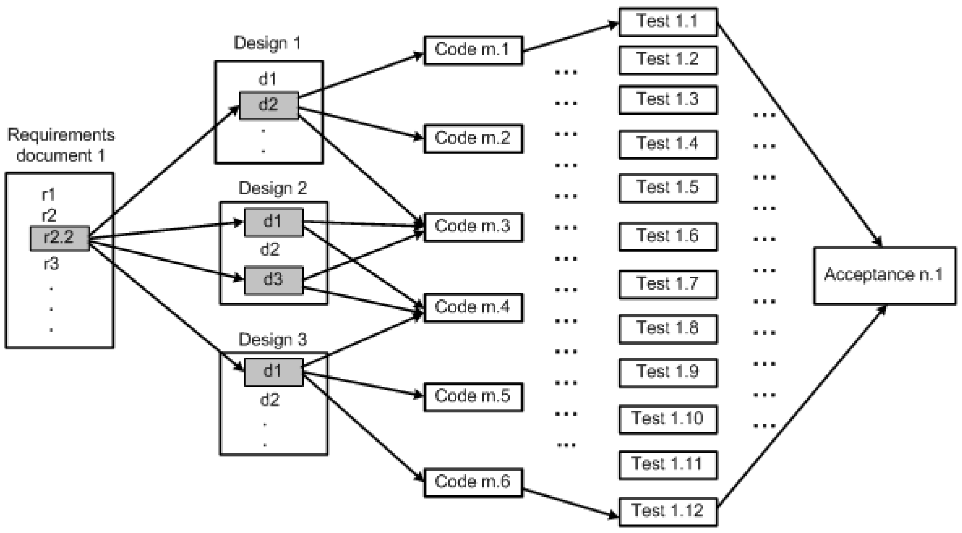


Figure 6: Traceability in software work products ©IEEE, 1991

2.6.3. Dependency Based Change Impact Analysis

In this type of analysis, to discover the impacts of implementing a change, some software artifacts like logic, modules, etc., are taken into account to examine the linkage of these artifacts. While Traceability-based CIA displays impact analysis at unique levels, Dependency-based CIA displays at the same level, for instance, design to design and code to code [18].

In the following sections, we discuss traceability-based change impact analysis techniques, related example studies and challenges, dependency-based change impact analysis techniques, related example studies and challenges. CIA tools which are built for academic purposes and their comparison table, and a commercial CIA tool list.

2.6.4. Traceability Based Change Impact Analysis Techniques

Traceability was described as “the ability to describe and follow the life of an artifact, in both a forwards and backwards direction” by Lucia et al. [20] If a document, for example, a requirement or use case is linked to a feature that needs to change, traceability helps in finding areas in the code and design that need to be preserved. There are 2 major types of traceability, horizontal and vertical. Horizontal traceability refers to traceability between:

- Requirement artifact and coding/testing/design artifact
- Requirement artifact and defect report
- Design artifact and coding artifact

Vertical traceability refers not only to traces between different software artifacts in a software phase, but also dependencies within a software artifact itself, such as dependencies among requirements in a use case specification. Data dependency, control dependency and component dependency use different vertical traceability techniques [61].

In this section, academic articles are analyzed by categorizing traceability as an experimental and empirical case.

A. Experimental Studies:

- Shahid et al. [27] suggested the Hybrid Coverage Analysis Tool (HYCAT) as a tool for managing traceability during software artifact changes. The technology was tested on an On-Board Automobile (OBA), and the findings were positive and noteworthy when compared to previous methodologies.
- Kugele et al. [28] suggested a model-based algorithm to aid trace connection visualization and better comprehend the relevance of each artifact and its effect on the others.
- In order to reestablish traceability linkages in IR techniques, Dit [29] suggested using genetic algorithms. The near-optimal solution is discovered by IR-GA using their method at each stage of the information retrieval process.
- According to Kchaou et al. [30], an IR strategy for ensuring "semantic traceability between use case documentation and sequence diagrams" was created, as well as a graph-based mechanism for modeling structural relationships. They conducted "a quantitative experiment with LSI frequency and Inverse Document Frequency" on JHotDraw 7.4.1, and the findings revealed that LSI had a greater clarity and recall value.
- According to Huang et al. Nejati et al. [31]. Their modeling technique identifies the influence of introducing changes in "requirements on a design" using the Systems Modeling Language (SysML). They used a static slicing technique to obtain an approximated set of impacted model elements and then ranked the resulting set of elements to anticipate the influence of the elements. According to their findings, 4.8 percent of the whole design must be reviewed to determine the elements that are impacted.

B. Empirical and Case Study Based

- The link between software and its code, or traceability, enables engineers to articulate their theories. When software traceability is maintained, according to Ghabi et al. [32], time is saved, and quality is increased. However, the information is not collected at the appropriate moment. They have put up a language for capturing traceability.
- Almasri et al. [33] proposed a model-based approach to telecommunications or embedded systems, in which their model employs dependencies to build two impact sets and EFSM models. Their findings revealed that a single change has a 14 to 38 percent influence on the overall model size.

Traceability based Change Impact Analysis Challenges

- Traceability links between heterogeneous artifacts (e.g., as test case, source code, design, requirements) are required to be established. Yet, the knowledge gap is a main challenge to establish such links. There is a high level of knowledge gap between software documentation and source code. The latter one follows language and program syntax while the formal one is usually expressed in natural language. To recover knowledge-based traceability links between these heterogeneous artifacts, data normalization and human expert verification is needed. [58]
- To determine the problem of recovering knowledge-based traceability links between artifacts of different types, IR has been widely used in recent decades. This approach

establishes traceability relationships with the assumption that two artifacts are potentially related if they share textual similarity. Although, IR-based approaches are error-prone, time-consuming and need human experts to verify selected trace links. [58]

2.6.5. Dependency Based Change Impact Analysis Techniques

Dependency-based in order to determine the impacts of making a change, CIA considers numerous software artifacts like variables, logic, modules, etc. and analyzes their interaction. They might be static, dynamic, or both.

In this section, static and dynamic techniques are examined in two different categories.

A. Static Techniques: Static approaches analyze software artifacts without running the program by using syntax and semantic analysis, text analysis, and change history repositories.

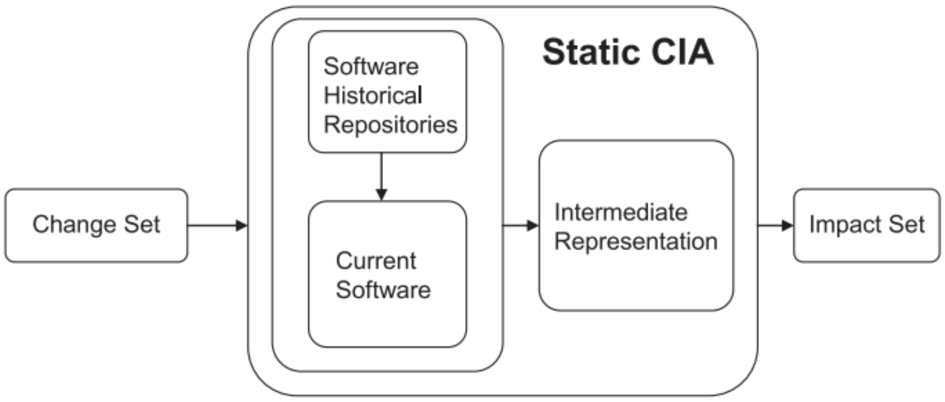


Figure 7: Static change impact analysis process. © Diamani et al. [3]

These techniques concentrate on the structure of the software. Most CIA techniques currently lack support for hidden dependencies and intergranular change impact questions, according to Sharma and Suryanarayana [34]. They invented AUGUR, a static automatic code analysis tool that addresses these issues by retaining semantic and environment dependencies between source code entities across granularities.

According to T Rolfsnes et al. [35], through a new technique named “Targeted Association Rule Mining for All Queries” (TARMAQ), they proposed using evolutionary coupling. They compared it to the ROSE and SVD tools and discovered that it is superior to both and is best suited for “performing robust change impact analysis for heterogeneous systems.”

According to Musco et al. [36], a strategy for forecasting influences circulation using four types of call graphs. To investigate how faults propagate, 17000 mutants were created using ten open-source Java projects and five mutation operators. According to their findings, the simplest basic call graph provides the optimal balance of accuracy and recall.

B. Dynamic Techniques: It consists of both offline and online CIA. It is carried out while the software is running. Offline CIA refers to a CIA approach in which data is obtained and evaluated after the program's execution is complete, whereas Online CIA refers to data retrieved while the program is still running [18].

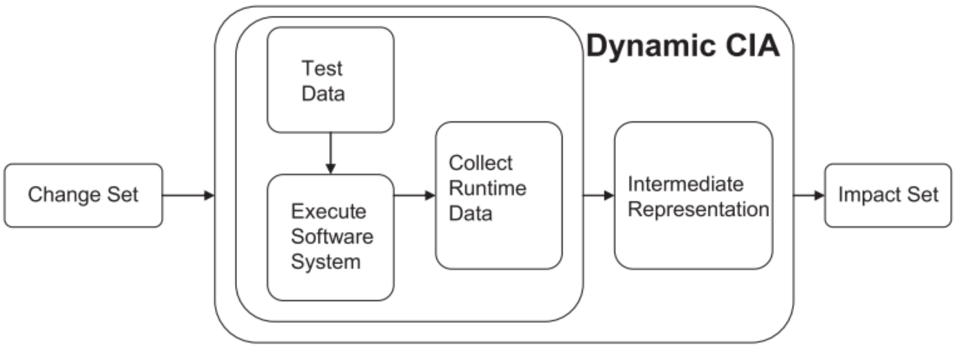


Figure 8: Dynamic change impact analysis process [3]

Cai and Santelices [37] suggested a three-instance approach for generating very precise impact sets at a low cost. To achieve high accuracy, they exploited static dependencies and execution traces. They presented a dynamic approach for Sensitivity Analysis dubbed SENSA in another study [30], which created statement-level impact sets. They used open-source Java applications and case studies to assess SENSA.

Cai and Thain [49] introduced DISTIA, a tool that evaluated the effects within and outside implementation by partly sorting distributed method-execution events and using message forwarding semantics. Their findings indicated that the analysis was completed in one minute and that the size of the impact set was decreased by 43 percent.

To forecast behavioral impact, Rajan and Kroening [39] created a measure that quantifies the change impact using two software versions. Their method is unusual in that it analyzes both versions of the software. They also put their measure to the test in three case studies.

Cai and Santelics [40] used a two-way method to investigate the prediction accuracy of dynamic CIA. To assess accuracy and retrieval, they employed execution differencing and sensitivity analysis. Their findings revealed that most low-cost dynamic analysis methodologies generate erroneous results in most situations, with an average precision of 38-50 percent and a recall of 50-56 percent.

2.6.6. Dependency based Change Impact Analysis Challenges

Inter-granular queries are not supported by most static CIA techniques. An inter-granular query specifies the proposed change at one source code granularity and provides the result of the query at another source code granularity. For example, the query “Display potentially impacted methods when this change will be made in this class” is an intergranular query. Contrarily, most of the current approaches would have only notified what other classes would be impacted by the proposed change. Inter-granular query support in CIA tool can accurately identify the change impact leading to a more effective CIA [34].

Current static techniques determine impacted entities by using control or/and data dependencies. Recall of such methods are low because they are affected by presence of hidden dependencies. Such dependencies have not been deeply researched and not supported by current CIA methods [34].

2.6.7. Tool Support for Change Impact Analysis

(1) **TRIC [42, 43]:** Requirements Management Tool Inferencing and Consistency Checking (TRIC) performs CIA and requirements estimations on software requirements using formal requirement semantics. The software's functionality improved by including capabilities such as displaying incompatible suggested modifications, proposing, propagating, and apply changes, and anticipating changes and their impact on the requirements model.

(2) **ImpactMiner [38]:** A tool that uses dynamic tracing, history mining, and SVN repository queries to estimate an influence set. It features a highly user-friendly GUI and is used as a plugin for the Eclipse tool. The user can easily grasp the findings thanks to the two tabs labelled "Feature view" and "Results view."

(3) **SafeRefactorImpact [43]:** Based on change effect analysis, Safe Refactor Impact is a method for determining if a transformation saves program activities. It works by assessing modifications made to Java or AspectJ applications and creating test cases for the methods that have been affected. It employs Safira, a change impact analyzer that detects affected techniques.

(4) **TraceAnalyzer [32]:** The utility, which is implemented as an Eclipse plug-in, supports many input perspectives. The list of imputes and the customary "trace matrix" (TM) are still present on the right and left sides, respectively. It also offers capabilities that assist engineers in discovering the footprint graph, flagging, and removing issues with accuracy and granularity.

(5) **FaultTracer [50]:** A toolkit which determines atomic changes from abstract syntax trees, finds their dependencies by tracing definition with reference to each used method and field, runs selected tests to emphasize failure-affecting changes and ranks these changes by using spectrum-based fault localization technique for Java programs.

(6) **CIAT [46]:** Change Impact Analysis Tool has two modules, Class Iteration Prediction and Impact Analysis Module. It is an automated tool that is developed based on our previous work on CIA for the software development phase. The uniqueness of the approach or the prototype tool is that the element of integration between static and dynamic analysis techniques.

(7) **Chianti [53]:** An Eclipse plug-in which takes two program versions and a regression test suite, finds interdependent atomic changes from different versions of programs, creates call graphs for test suite and determines potentially impacted methods and relevant affecting changes.

(8) **EAT [57]**: Created to evaluate the CollectEA technique, Execute-After Tool (EAT) focuses to emphasize the benefits of dynamic analysis over static analysis. EAT consists of three components: An analysis module, an instrumentation module and a set of runtime monitors.

(9) **Impala [22]:** An Eclipse plug-in which performs CIA before with data mining algorithms, before execution of changes. By comparing two program versions, Impala creates a changeset that contains potentially impacted entities and detected changes by generating a dependence graph.

(10) **ROSE [57]:** An Eclipse plug-in that mines project's version history with CVS and makes users understand the consequences of making changes. Based on previous commits to version control, ROSE can propose changes to prevent errors. The proposals are ranked by confidence level.

(11) **JRipples [52]:** An Eclipse plug-in that uses static information to analyze dependencies between entities in order to help developers locate the impact set manually, by keeping track of visited elements and the elements that are dependent on them.

Table 3: Academic Change Impact Analysis Tools

Tool Name	Objective	Change Impact Category	Supported Languages	Method	Input	Output	Validation
TRIC	Limiting the impact explosion during change impact analysis and prediction in requirements models	Traceability Based Change Impact Analysis	Java	Requirements Inferencing and Consistency Checking	Change Type Requirement which Changes Introduced	Decision Tree Propagation Path	5 change scenarios in a real software requirements specification
ImpactMiner	Impact analysis at change request level that adapts to the specific software maintenance scenario at hand	Dependency Based Change Impact Analysis	Java	Integrated Impact Analysis	Source Code Search Query	Execution Traces Historical Data Potentially Affected Methods	4 open-source system

Tool Name	Objective	Change Impact Category	Supported Languages	Method	Input	Output	Validation
SafeRefactor Impact	Check whether an object oriented or aspected oriented transformation is behavior preserving	Dependency Based Change Impact Analysis	Java	Behavioral Change Analysis	Source Code	Impacted Methods and Test Cases Generated for Them	5 transformations applied to programs with different sizes (10 LOC to 79 KLOC)
TraceAnalyzer	Automate the traceability between software architectural models and extra-functional results	Traceability Based Change Impact Analysis	Java	Footprint Graph	Trace Assumptions	Video on Demand Footprint Graph Trace Matrix Dependencies List	6 case study systems
CIAT	Overcome the challenges when using both static analysis and dynamic analysis techniques	Traceability and Dependency Based Change Impact Analysis	C++	Class Dependency Filtration	Change Request Source Code	Impacted Class List	3 software development projects
FaultTracer	Ranks program edits in order to reduce developers' effort in manually inspecting all affecting changes	Dependency Based Change Impact Analysis	Java	Syntax Tree Bytecode Manipulation	Source Code Regression Test Suite	Ranked List of Affecting Changes	23 versions of 4 different program
Chianti	Find a subset of the changes that impact a test whose behavior has	Dependency Based Change Impact Analysis	Java	Call Graph Syntax Tree	Source Code Regression Test Suite	Affected Tests Affecting Changes	Versions of a software development project

Tool Name	Objective	Change Impact Category	Supported Languages	Method	Input	Output	Validation
	(potentially) changed.						
Diver	Exploits static dependencies to identify runtime impacts precisely without reducing safety and at acceptable costs	Dependency Based Change Impact Analysis	Java	Call Graph Execution Trace	Java Bytecode Call Queries	Impact Set	4 Java programs
EAT	Introduce a new dynamic analysis approach which is practical, precise and efficient	Dependency Based Change Impact Analysis	Java	Execute After Relation	Source Code	Impact Set	Several releases of 2 programs
Impala	Calculates the impacted elements by identifying all the direct and indirect dependencies of a change	Dependency Based Change Impact Analysis	Java	Call Graph Dependencies	Source Code from Subsequent Revisions	Impact Set	3 software projects
ROSE	Suggests locations for further changes and warn for missing changes	Dependency Based Change Impact Analysis	Java C++ C Python	Historical Analysis	Source Code	Ordered list of suggested places which should be changed	10k transactions in 8 open-source projects

Tool Name	Objective	Change Impact Category	Supported Languages	Method	Input	Output	Validation
Coda	Analyze Scala source code with its change to provide impacted elements	Dependency Based Change Impact Analysis	Scala	Dependency Analysis	Source Code	Ordered list of impacted classes by likelihood	3 open-source projects
JRipples	Provide organizational support to make the incremental change process easier and systematic	Dependency Based Change Impact Analysis	Java	Change Propagation	Source Code	Method/Class Status List	An open source projects

The full table can be reached from here: Tool Tables

Change Impact Analyzer Helpers

(12) **CodeDiff [44]:** This tool used to process all the files in every change-set for source code differences at a fine-grained syntactic level.

(13) **Cobra [45]:** To scan in source code, Cobra employs a lexical analyzer for C. It makes it easier to look for trends, determine whether coding standards and norms are being followed or not, find suspicious code fragments, etc. offers an interactive tool to software developers, peer reviewers, testers, and quality assurance staff.

(14) **FLAT [51]:** A tool for performing feature location using textual searches, execution traces, annotating features and visualization.

Table 4: Academic Change Impact Analysis Tool Helpers

Tool Name	Objective	Supported Languages	Method	Input	Output	Year	Source Code Link
CodeDiff	Source Code Difference Process	Any	Word Differencer	Source Code	HTML Report	-	http://www.safe-corp.biz/products_codediff.htm

Tool Name	Objective	Supported Languages	Method	Input	Output	Year	Source Code Link
Cobra	Source Code Lexical Analysis	C, C++, Java	Lexical Analysis	Source Code	Call/Control Graph	2015	https://software.nasa.gov/software/NPO-50050-1
FLAT	Feature Location	Java	SITIR Approach	Search Query Test Case for Desired Feature	Feature Mapping	2010	https://www.cs.wm.edu/semeru/flat3/

2.6.8. Commercial Change Impact Analysis Tools

In this section, commercial change impact analysis tools are presented with their main features. Also, CIA tool link information and demo information is given.

Table 5: Commercial Change Impact Analysis Tools

Tool Name	Description	Link(s)	Demo Available
Smart TS XL	The Software Intelligence® technology in SMART TS XL provides rapid impact analysis by means of an extensive cross-reference utility, showing users a color-coded graphic of how and where programs interact. You can identify the areas that could require additional attention and testing with the capability to map dependencies between related modules. Responsive and user-friendly, this impact analysis tool greatly reduces the time required to understand and evaluate IT projects. • Build color-coded cross-reference diagrams • Click and follow hyperlinks that connect elements • View specific lines where references occur • Identify which elements are connected and where • Create reports that can be saved, exported and printed	https://in-com.com/solutions/impact-analysis/	YES
Foresight	Foresight provides full visibility and deep insights into the health and performance of your tests and CI/CD pipelines. Assess the risk of changes, resolve bottlenecks, reduce build times, and deliver high-quality software at speed with Foresight.	https://www.runforesight.com/#Change-impact-analysis	YES
Spec-TRACER	“Spec-TRACER™ addresses the objectives defined by safety critical standards and related to traceability data and requirements coverage.	https://www.aldec.com/en/products/r	YES

Tool Name	Description	Link(s)	Demo Available
	Spec-TRACER captures traceability data from miscellaneous design files, verifies it, and produces traceability matrices required for the certification processes." It has a direct integration with IBM DOORS. It facilitates management, traceability, reporting, requirements capture and impact analysis. Change Impact Analysis Know the impact of requirements changes before and after they occur Know the exact number of projects elements that will be impacted	equirements_management/spec-tracer https://www.aldec.com/files/products/SpecTracer_Datasheet.pdf	
Visure	Gain End-to-End Traceability by Automatizing your Change Impact Analysis Process. Empower your team to make better and informed decisions by eliminating manual tracking change impact & providing them an accurate understanding of the implications of a proposed change.	https://visuresolutions.com/features/impact-analysis	YES
Tricentis	Tricentis LiveCompare provides fast, automated impact analysis for any update to your SAP systems. It works across the entire SAP ecosystem, including ECC, CRM, BW, Fiori, and, of course, S/4HANA. When paired with Tricentis Tosca, LiveCompare reduces testing time by 85%, accelerates releases by 40%, and increases quality by 75%. • Use LiveCompare to analyze the impact of change. From audits through to SAP upgrades, identify how code, config and data is impacted by change, as well as security settings. • Automatically compare multiple SAP systems to ensure they are aligned when making change. Tackle the challenges of dual maintenance and transport overrides. • Test less without compromising system quality. Use LiveCompare to identify exactly what to test and why. Integrate with Tricentis Tosca for resilient test automation or your own solution.	https://www.tricentis.com/resources/tricentis-livecompare-data-sheet/ https://www.tricentis.com/wp-content/uploads/2021/07/Tricentis-data-sheet_LiveCompare-for-SAP.pdf	YES

Tool Name	Description	Link(s)	Demo Available
Lattix	**Impact Analysis** Perform impact analysis on a defect, to see what files and packages are affected by a fault in the software. **Changed Based Testing** Perform Change Based Testing by analyzing the impact of code changes and re triggering only those unit and integration level tests affected by the changes.	https://www.lattix.com/products/lattix-2021/	YES
Jama	• Easily navigate upstream and downstream relationships to understand the impact of change and coverage across the development lifecycle. • Save time finding gaps in overall test coverage • Understand change impact before it happens • Produce traceability documentation required by regulators • Relationship Rules are tracked across projects with a visual schematic that shows the impact and reach of information across the organization • Engage in real-time conversations about the impact and prioritization of defects	https://www.jamasoftware.com/platform/jama-connect/features/#tab-id-1	NO
Roadmap Pro	The Change Impact Assessment (CIA) is an online change management assessment tool that: • Measures and compares the likely disruption of a change project on people in different parts of the affected business • Assesses how difficult it could be for people to adapt or commit to change • Updates project risk logs with diagnosis of new barriers to successful implementation • Determines how the impacts and risks inherent in the change inform implementation choices	https://info.changefirst.com/change-impact-assessment-tool https://www.changefirst.com/change-management-products/roadmap-pro	YES
ChangeMiner	Change impact analysis with business logic information in source code. Path-sensitive string analysis technology enables the most accurate application change impact analysis. Robust string analysis engine enables the most accurate application change impact analysis! Improve your application team's productivity by 30~75% with application visibility!	http://www.changeminer.com/	NO

Tool Name	Description	Link(s)	Demo Available
Ktern	To summarize the importance of KTern's Simulation Bot in SAP Change Impact analysis and Release Management. One can say that the bot automates all the possible changes, which can help in better strategizing the release plan showing the possible impact, the stakeholders who are impacted due to these changes.	https://ktern.com/article/release-management-ktern-simulation-bot/	YES
AGS	This All-in-One Change Impact Assessment Tool has been created for: 1. Conducting organizational impact assessments, tracking, reporting, and management 2. Business change assessments and reporting 3. Assessing impacts from the process, system, technology, digitalization, and tool changes 4. Analyzing impacts from culture changes, mindset shifts, business strategy, and vision changes 5. Analyzing enterprise-wide or group-wide transformations 6. Assessment of new policy and procedure impacts 7. M&A and business expansions	https://www.airiodion.com/best-assessment-tool/ https://www.youtube.com/watch?v=IESuWG_nxzU&ab_channel=AGSCorp	NO
Oracle Change Impact Analyser	Change Impact Analyzer is a tool installed separately from PeopleSoft PeopleTools that helps you determine the impact of specific changes you plan to make during an application upgrade. It's an interactive program where you can see the relationships of PeopleSoft definitions in a hierarchical view. Change Impact Analyzer displays several views of analyses in tabular and text views. It's delivered with a set of *rules* that are used to determine the relationships between definitions. Typically, these rules are written in SQL.	https://docs.oracle.com/cd/E92519_02/pt856pbr3/eng/pt/tcia.html (Use navigation menu on the left side.)	NO
Praxie	A **Change Management Impact Analysis** is a method that is used to identify relevant stakeholders in a change management process as well as the risks and benefits that the change management initiative provides to them. Based on this information, your team will be able to discern the impact that the change management program has on key individuals.	https://praxie.com/change-management-impact-analysis-online-tools-templates/	YES
Infotech	In order to lead your staff members through change, you must understand the level of impact the change will have on them. Use this tool to answer key questions that will inform your people change management decisions during the change process. This tool will provide you with staff	https://www.infotech.com/research/change-impact-assessment-	YES

Tool Name	Description	Link(s)	Demo Available
	impact assessment, risk points analysis and recommendations for managers.	tool#unlock-modal	
Change Method	The Change Impact Assessment Framework tool provides a framework for examining the detailed impacts that will arise from the change program. Begin by documenting any headline process changes identified during the Define Future State process, make an initial classification of the changes required and build out the detail as you gather more and more information through surveys, interviews, workshops and even observation.	https://www.changemethod.com/change-impact-assessment-framework/	NO
WhatFix	Navigating through organizational change is a multi-step process. Whatfix helps you scale enterprise-wide changes, improve user engagement, and drive user adoption.	https://whatfix.com/solutions/change-management/	YES

2.6.9. Findings and Future Scope

- To verify validity of trace links generated by IR, some studies recommend training machine learning classification models. Also, several researchers have stated benefits of deep learning-based approaches against IR-based approaches. Mostly, the former can learn unstructured data of any format such as correlations among design and requirement documents [58].

- It is spotted that the relation of CIA parameters and existing metrics is over-studied. Instead of empirically exploring the correlation with old metrics, researchers should propose accurate, direct and novel indicators [54].

- The knowledge gap between CIA tools used by academia and CIA tools used by industry should be filled with a bridge. Creating a roadmap beforehand can help tool planning, development and future plans of the tool [57].

- The outcome of usability inspection and literature review have exposed many fruitful fields of future work. Full usability analysis combined with informal usability inspection can be conducted to determine developers' needs [57].

- The measurement of impact needs to be researched deeper. A new technique can be discovered which has a reliable and helpful metric. Developers can save analysis time with a metric that helps them to decide whether they should implement the changeset [57].

- There is a belief that the CIA is crucial and should be done. However, there is little evidence about which aspect of software development is affected by the CIA [54].

- It is noted that there is not any change impact analysis API library, even though there are plenty of CIA tools. Implemented CIA algorithms in tools can be repackaged in open-source API so that industry can reach them easily [57].

Software change impact analysis (CIA) is a technique used to identify the potential effects caused by software changes, which plays an important role in software development and maintenance. There are many automated tools that apply Change Impact Analysis available. These tools mainly use traceability-based CIA or dependency-based CIA. Dependency based CIA techniques can be divided as static analysis, dynamic analysis or combination of them while traceability-based CIA has 2 major types, horizontal traceability and vertical traceability. We presented studies and challenges for both CIA types. Also, we provided a comparison table for academic based tools with functional metrics and a list of commercial tools to emphasize the status of CIA in industry. Then, we outlined our findings and future scope. Our findings show that CIA research should be broadened with new perspectives and metrics so that the impact of a change can be determined comprehensively. To conclude, CIA is still a popular research topic which has many active researchers and practitioners. It has a great potential to be evolved further with different approaches.

3. Key Innovation Areas

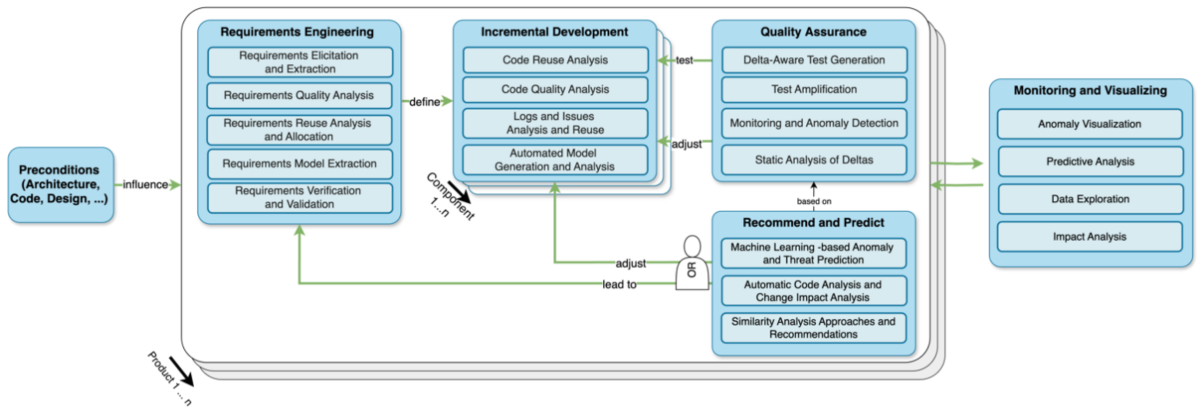

Preconditions (Architecture, Code, Design, ...)
influence
Requirements Engineering
Requirements Elicitation and Extraction
Requirements Quality Analysis
Requirements Reuse Analysis and Allocation
Requirements Model Extraction
Requirements Verification and Validation
define
Incremental Development
Code Reuse Analysis
Code Quality Analysis
Logs and Issues Analysis and Reuse
Automated Model Generation and Analysis
Component 1...n
test
adjust
Quality Assurance
Delta-Aware Test Generation
Test Amplification
Monitoring and Anomaly Detection
Static Analysis of Deltas
based on
Recommend and Predict
Machine Learning -based Anomaly and Threat Prediction
Automatic Code Analysis and Change Impact Analysis
Similarity Analysis Approaches and Recommendations
adjust
lead to
OR
Monitoring and Visualizing
Anomaly Visualization
Predictive Analysis
Data Exploration
Impact Analysis
Product 1 ... n

Figure 7 WP4 Dependencies with SmartDelta Methodology

In this work package, we have three main innovation areas which belong to the recommend and predict module of the SmartDelta methodology as shown in Figure 7. They are based on the quality assurance input from Work Package 3. In the following, we discuss each of the innovation areas along with the projects that the SmartDelta project contributed so far.

A. Novel ML-Based Anomaly and Threat Detection Methods

The development and deployment of machine learning-based methods have transformed anomaly and threat detection across various domains. These approaches have significantly enhanced the ability to detect anomalies, localize errors, and prioritize threats in complex systems, including micro-service architectures, cybersecurity environments, and telemetric data applications. By employing both supervised and unsupervised models, these methods achieve significant accuracy, scalability, and adaptability, enabling proactive responses to potential threats. The integration of these techniques into live environments demonstrates their robustness and readiness for real-world challenges, marking a large step forward in operational security and reliability. The related projects are as follows:

- Prediction of localization of anomalies and errors using ML methods in micro-service-based architecture (NetRD)
- Anomaly detection and prioritizing cybersecurity offenses by utilizing a diverse set of supervised and unsupervised models (Ontario Tech, Glasshouse Systems)
- Anomaly detection for telemetric data (Hoxhunt)

B. Automatic Code Analysis and Change Impact Analysis Approaches

Automated tools for code analysis and change impact evaluation have contributed to advancements in software development practices. These approaches have addressed challenges in knowledge sharing, fault prediction, and technical debt management, enabling teams to focus on high-impact tasks. By automating tedious processes like metrics collection and analysis, these methods reduce manual effort, increase productivity, and ensure the scalability of modern software systems. The achievements in this area have not only improved code maintainability but also facilitated better decision-making. The related projects are as follows:

- Improved knowledge sharing among developers using automatic metrics collection from version control systems for impact analysis (ERSTE, DAKIK, Kuveyt Turk)

- Automatic collection of code analysis metrics of cloud-based software and faults predictions (Ontario Tech, Team Eagle)

- Automatic code analysis for ease of software maintenance (University of Innsbruck and cc.com)

- Automatic analysis of technical debts (Cape of good code, Vaadin)

- Analyze software quality trends based on issues and schedule the issues to find the balance between focussing on improving quality versus adding new features (FOKUS)

C. Similarity Analysis Approaches and Recommendations

The advancements in similarity analysis and recommendation techniques represent an advancement in optimizing software systems. By implementing graph-based methods, hierarchical modularization, and machine learning-driven insights, these approaches enable developers to uncover patterns and produce actionable recommendations efficiently. The ability to identify similarities and address issues proactively has streamlined development workflows, improved modularity, and fostered a deeper understanding of system behavior. The achievements in this domain have laid the groundwork for more intelligent and adaptive software systems, promoting innovation and reducing complexity in software engineering processes. The related projects are as follows:

- Graph based similarity analysis and recommendations (TWT, Software AG, Vaadin)

- Similarity analysis of State Machines using hierarchical modularization (TWT, Akkodis)

- ML-based methods to identify requirements from large data repository and generate recommendations (RISE, Alstom)

- Software requirements and issues analysis using Natural Language Processing and Continual Learning (IFAK, Software AG)

4. Contributions to the State-of-the-art

4.1. Novel ML-Based Anomaly Detection Methods

4.1.1. Prediction of localization of anomalies and errors using ML methods in micro-service-based architecture (NetRD)

Synopsis:

In microservice platforms with high number of users and heavy traffic, it is necessary to monitor the system, take quick action against errors and ensure the maintainability of the system. However, debugging on these platforms can take a long time. This difficulty arises from the need of understanding the behavior of microservices and detecting their interactions. In the first phase of this study, which aims to increase the efficiency of DevOps engineers on the work/time unit, it has been observed that providing microservice flows and interactions saves operation teams a significant amount of time during debugging. Accordingly, the study focused on microservice interactions and anomaly detection. First, different machine learning-based models predicting microservice interactions have been developed and their performances compared. In these models, log patterns are extracted on microservice log data and the interaction map of the mentioned microservice is created by estimating the previous and next microservices that the current microservice interacts with at a certain moment. In the next step, anomalous data were injected into the microservice logs, models were developed to detect these data and their performances were compared. In the experiments, successful estimation results were obtained that can contribute positively to the debugging process.

Related works:

There are many works in the literature that address fault analysis and anomaly detection. In this section, we provide a brief review of recent studies in the existing literature.

X. Zhou et al. [NR1] conducted an industrial survey of typical faults encountered in microservice systems, current debugging methods used in industry, and challenges faced by developers. According to this study, monitoring and visualization analysis techniques are methods that developers use to find various types of errors involving microservice interactions.

Giamattei et al. [NR2] present a systematic study of 71 monitoring tools for DevOps and microservices. The study follows three main phases: search and selection, data extraction, and synthesis. The tools are categorized according to 26 parameters, including general characteristics, monitored aspects, and implementation details. The study includes a comprehensive map of the monitoring tools landscape, a reusable classification framework, and discussions on implications for researchers and practitioners. In particular, the map can be used to understand tool characteristics, identify gaps and make informed decisions in the context of DevOps and microservice monitoring.

Wang et al. [NR3] conducted a survey study on localization and replication of software bugs. In this survey study, the authors discuss the research questions related to the problems studied, the research methodologies used, and the findings of previous research. They analyzed 134 papers published between 2011 and 2021 and investigated defect localization approaches.

Zhou et al. [NR4] conducted an industrial survey and an empirical study to investigate fault analysis and debugging in complex microservice systems. The survey reveals the challenges developers face in microservices debugging and shows the need for improved techniques. The empirical study evaluates the effectiveness of current industrial debugging practices and shows that appropriate tracing and visualization techniques improve microservice debugging. The findings highlight the importance of intelligent trace analysis and visualization and suggest potential directions for future research. The study presents a survey on industrial microservice systems, a benchmark for microservice failure analysis, and insights into improving microservice debugging with advanced tracing and visualization methods.

Yu et al. [NR5] investigate the efficiency and effectiveness of machine learning algorithms such as K-nearest neighbor (KNN) and deep learning methods such as convolutional neural network (CNN) in log anomaly detection considering their computational cost. The study on five general log anomaly detection datasets reveals that basic algorithms such as KNN outperform DL methods in terms of both time efficiency and accuracy. This result is driven by log preprocessing strategies, the simplicity of available log benchmarks, and the nature of binary classification in log anomaly detection. Based on the findings, the authors recommend critically analyzing datasets and research tasks before opting for computationally expensive DL methods in log anomaly detection and exploring simpler approaches as a basis for software engineering tasks.

Yu et al. [NR6] introduce the Nezha approach, which offers an innovative root cause analysis (RCA) for large-scale microservice systems. Their aim is to address the limitations of existing RCA methods, such as poor root cause interpretation and underutilization of data. Nezha combines multimodal observability data, including metrics, traces and logs, and transforms them into a unified event representation to create event graphs. It focuses on obtaining detailed and interpretable RCA by comparing error-free and error-exposed stages, identifying root causes at the code region and source type level. They tested the approach through empirical evaluations on two widely used microservice applications and observed its performance. The study shows that Nezha improves the observability of two microservice applications, is successful in anomaly detection, and contributes to RCA on multimodal data.

Methodology:

CPaaS Platform Worked On:
CPaaS, a telecommunications platform with a high number of microservices, developed in a technology company, operating in 5 different data centers located in 4 different continents, is a software platform that provides communication services to users through a scalable, microservice architecture-based platform by making use of PaaS, which is one of the popular cloud computing models. On the other hand, it can also offer VoIP (Voice over Internet Protocol) APIs on the same platform so that companies can use them in line with their own needs.

Debugging and problem addressing process varies according to the components in the CPaaS platform, which contains a large number of microservices and consists of many different components. The most difficult errors to address for operations teams are in the fields *Routing* and *Services and* are reported directly from user scenarios as they include service functions. The main reason for the challenge here is the need to detect microservice interactions. For example, for debugging process in a basic call scenario, all interacting microservices and their behaviours must be known and understood. This creates a time handicap for a solution with a high number of microservices. On the other hand, it has been observed that the time required for troubleshooting user scenarios is related to knowing the relationship between the scenario and microservices.

Solution Stages:

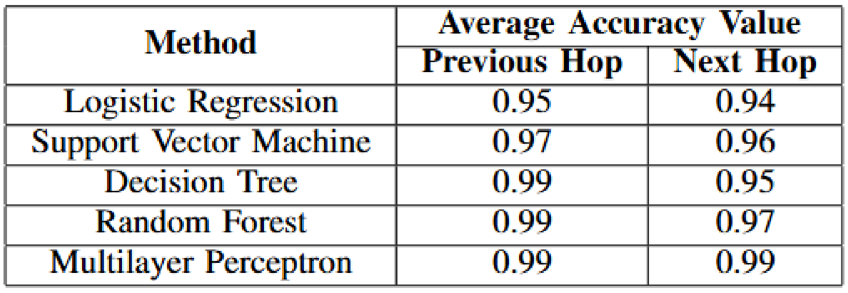
In the first phase, different machine learning algorithms were used to estimate, interactions between microservices and model performances were compared. The experiments were carried out within the framework of data belonging to multiple scenarios, but performance comparison includes the results for both single and multiple scenarios results.

In the second phase, anomaly detection studies were performed. The anomalies injected into the system were manually labelled and the problem was transformed into a classification problem. Classification was made with different algorithms on manually labelled data and model performances were compared. On the other hand, unsupervised labelling process was performed with different data combinations created considering feature importance test results, and the performance of this process against actual values was observed. At the end of this process, the labelling process with the unsupervised method gave almost real results and it was observed that the effort to be applied for manual labelling processes could be eliminated.

Results:

Microservice Interactions Prediction
The *k-fold cross-validation* technique was used in the performance analysis of the interaction prediction models between microservices, and tests were made for the

value of k = 5. The performance metrics were calculated separately for each model during the experiments. The average accuracy values obtained as the mean of three models are given in Table I. The success of estimating “Previous Hop” vary between 95-99% whereas the average success in evaluating “Next Hop” is in the range of 94-99%. Among the tested algorithms, the highest success was achieved with the models created by the MLP algorithm, with 99% for both “Previous Hop” and “Next Hop” predictions.

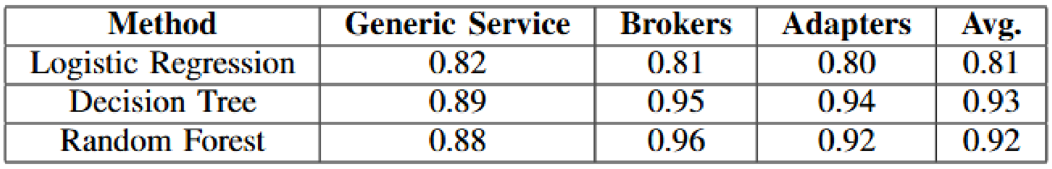


Method	Average Accuracy Value	
	Previous Hop	Next Hop
Logistic Regression	0.95	0.94
Support Vector Machine	0.97	0.96
Decision Tree	0.99	0.95
Random Forest	0.99	0.97
Multilayer Perceptron	0.99	0.99

Table 6: Microservice Interactions Prediction Experimental Results

Anomaly Detection and Prediction

In the performance analysis of the anomaly prediction models in manually labelled data, the tests were repeated for each of the algorithm. There are three models (generic service, brokers, and adapters) created and evaluated by each algorithm. The average accuracy values obtained are given in Table 7.



Method	Generic Service	Brokers	Adapters	Avg.
Logistic Regression	0.82	0.81	0.80	0.81
Decision Tree	0.89	0.95	0.94	0.93
Random Forest	0.88	0.96	0.92	0.92

Table 7: Accuracy Rates of Anomaly Prediction in Manually Labelled Data Experiments

Table 7 shows that the highest success was obtained with the Decision Tree algorithm, and the average percentage of success achieved is 93%. However, despite the high success rates achieved in the experiments, 83% of the anomaly cases were correctly predicted at most. This shows that, in fact, normal situations were predicted more accurately, and thus model performance metrics were positively affected. This is because the data set mainly contains normal data rather than anomalies and unstable data sets are often encountered in the industry.

In the second phase of the anomaly detection experiments, it was aimed to automatically detect and label anomalies. Firstly, a ranking list from highest to lowest value score was obtained via feature importance test. In this respect, the features with the most important degrees for determining the anomaly class were used as input data with different combinations. Here, the highest scored four features were used to create double, triple, quadruple in combinations. Also, the last combination includes all the features except the highest one. Afterwards, anomalies were automatically detected and labelled in these data using the

Isolation Forest algorithm. The average accuracy values obtained for each combination are given in Table III.

Data Combination	Generic Service	Brokers	Adapters	Avg.
IF_1	0.96	0.83	0.83	0.87
IF_2	0.96	0.83	0.83	0.87
IF_3	0.96	0.83	0.83	0.87
IF_4	0.97	0.94	0.87	0.93
IF_5	0.88	0.87	0.90	0.88

Table 8: Accuracy Rates of Anomaly Detection with Unsupervised Methods Experiments

The highest success in anomaly detection was achieved for the quadruple data combinations with the highest significance. The accuracy of the predictions and, accordingly, the performance metrics were calculated by comparing them with the manually labelled actual anomaly values. At least 95% of anomalies were correctly predicted in these models with an average success rate of 93%. The results show that the Isolation Forest method has obtained results that are very close to the manually labelled actual values for the detection of anomalies. Only 5% of injected anomalies were labelled as normal status. Such a small loss allows Isolation Forest to be preferred as an automatic anomaly detection and tagging method, eliminating such manual labelling operations.

Method	Generic Service	Brokers	Adapters	Avg.
Logistic Regression	0.85	0.85	0.86	0.85
Decision Tree	0.89	0.96	0.93	0.93
Random Forest	0.89	0.96	0.92	0.92

Table 9: Accuracy Rates of Anomaly Prediction in Unsupervised Labelled Data Experiments for Quadruple Combined Data (IF_4)}

In the anomaly detection in unsupervised labelled data step, the same supervised classification algorithms were used for evaluation with the same input data, but the targets labelled by the Isolation Forest were used to be predicted. Since Isolation Forest models show the most successful results when labelling anomalies in the quadruple combination data, Table IV shows the average success values of the estimation results using only this target column. According to the results of these experiments, the highest performance was demonstrated by the Decision Tree algorithm. The best degree of success was achieved with a combination of four data. The percentage of success in these experiments ranged between 85% and 93%, and almost the same success rate was achieved with manually labelled data.

Summary:

For CPaaS, a microservice architecture telecommunications platform with a high number of users and heavy traffic, the error addressing process depends on the system components. In the examinations made, it has been observed that the operation teams spend the most time to eliminate the errors experienced in user scenarios, pointing to the routing and service components. On the other hand, it was concluded that knowing the interaction between microservices shortens the error recovery times.

This study, which aims to increase the efficiency of the operation teams on the work/time unit and to enable the new members of the operation team, especially newly graduated engineers with no experience in the industry, to adapt to the debugging processes faster, is based on the aforementioned inference. The study focuses on prediction of interactions between microservices and detecting anomalies on the CPaaS platforms' microservice system.

Note:
More detailed information can be found in the following publications.
1. Interaction Prediction and Anomaly Detection in a Microservices-based Telecommunication Platform
2. Microservice Interaction Prediction in Communication Platform as a Service

References:

[NR1] X. Zhou et al., "Fault Analysis and Debugging of Microservice Systems:Industrial Survey, Benchmark System, and Empirical Study," in IEEE Transactions on Software Engineering, vol. 47, no. 2, pp. 243-260, 1 Feb. 2021.
[NR2] L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dˆınga, A. Koziolek, S. Singh, M. Armbruster, J.M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. Fernandez Vogelin, F. Simon Panojo, "Monitoring tools for DevOps and microservices: A systematic grey literature review," Journal of Systems and Software, Volume 208, 2024.
[NR3] D. Wang, M. Galster, M. Morales-Trujillo," A systematic mapping study of bug reproduction and localization", Information and Software Technology, Volume 165, 2024.
[NR4] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, D. Ding," Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," IEEE Trans. Softw. Eng., 47, 2 (Feb. 2021), 243–260.
[NR5] B. Yu, J. Yao, Q. Fu, Z. Zhong, H. Xie, Y. Wu, Y. Ma, P. He," Deep Learning or Classical Machine Learning? An Empirical Study on Log- Based Anomaly Detection." In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24) New York, NY, USA, Article 35, 1–13.
[NR6] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, Z. Zheng, "Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data," In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)., New York, NY, USA, 553–565

4.1.2. Anomaly detection and prioritizing cybersecurity offenses by utilizing a diverse set of supervised and unsupervised models (Ontario Tech, Glasshouse Systems)

Synopsis:

This section explores the development of Machine Learning (ML) based anomaly detection and offense prioritization tool inside a Security Information and Event Management (SIEM) environment. SIEM solutions aggregate data from multiple sources, enabling analysts to monitor and detect threat patterns, respond to incidents, and manage security effectively. Our application, specifically designed for the QRadar SIEM platform, integrates ML models to enhance both anomaly detection and offense prioritization.
For anomaly detection, the application uses a time-and-space-efficient data extraction process with QRadar's Ariel Query Language (AQL) to manage large data volumes. ML models like Isolation Forest (iForest) and Local Outlier Factor (LOF) are selected for their efficiency in real-time SEIM environments. Point adjustment further refines anomaly detection by identifying anomaly sequences, allowing improved detection.
While anomaly detection helps the analyst to identify potential offense, a probabilistic ML approach is also employed that assigns impact scores to detected offenses. Probabilistic ML Models such as Cluster-based Outlier Probability (COPOD), calculate prediction probabilities, producing a prioritization list that guides SOC analysts in focusing on high-risk events. Evaluated using metrics such as Mean Time to Detect (MTTD) and Mean Time to Resolve (MTTR), this prioritization framework effectively reduces response time, streamlines analyst workflows, and enhances SOC efficiency.
This work, a collaboration between Ontario Tech University and Glasshouse Systems, demonstrates how advanced ML techniques can be integrated into SIEM applications to improve cybersecurity operations through accurate anomaly detection and offense prioritization.

Related works:

The integration of ML for cybersecurity anomaly detection has recently got a lot of attention, focusing on the application of both supervised and unsupervised techniques. Supervised models like Support Vector Machines (SVM) and Random Forests have been widely adopted for their accuracy in known attack scenarios [GHS1]. Meanwhile, unsupervised approaches such as k-means clustering and Autoencoders are effective for identifying new and emerging threats by analysing patterns in data [GHS2] [GHS3]. Hossain et al. (2021) developed an Automatic Event Categorizer for SIEM that utilizes ML to categorize events within a SOC environment, aiming to streamline alert management and reduce manual categorization efforts [GHS4]. This approach demonstrates how machine learning can improve SOC workflows by filtering and categorizing high-volume alerts, which is aligned with our goal of prioritizing cybersecurity offenses based on anomaly scores. Time-series anomaly detection techniques, as discussed by Mejri et al. [GHS5], provide foundational methods, yet the application within real-time SIEM settings is still developing. However, utilizing ML for prioritizing offenses and detecting anomalies of event per second (EPS) in real-time is less explored. This work addresses the gap by implementing solutions that optimizes both detection and prioritization within a live SIEM application.

Methodology:

First, to detect anomaly, our tool employs a series of steps as shown in Figure 9 Upper Left) to ensure efficient and accurate identification of unusual patterns in time series data. The methodology comprises:

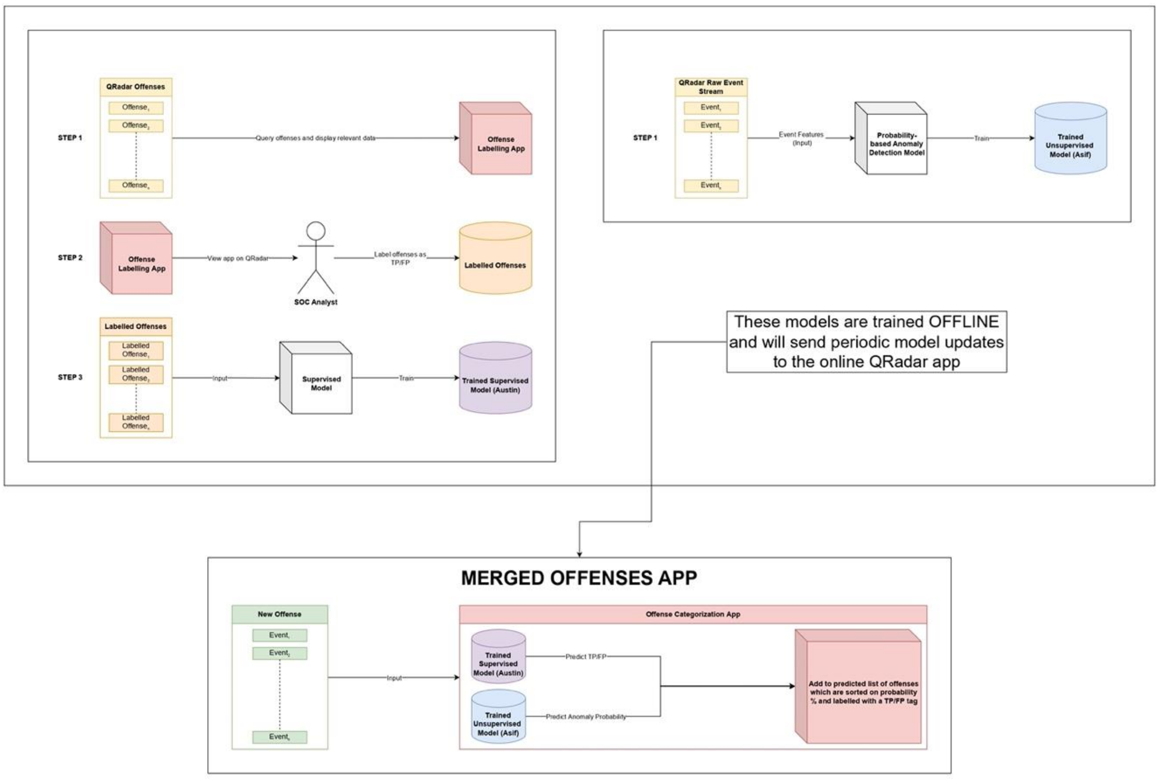

STEP 1
QRadar Offenses
Offense₁
Offense₂
Offenseₙ
Query offenses and display relevant data
Offense Labelling App
STEP 2
Offense Labelling App
View app on QRadar
SOC Analyst
Label offenses as TP/FP
Labelled Offenses
STEP 3
Labelled Offenses
Labelled Offense₁
Labelled Offense₂
Labelled Offenseₙ
Input
Supervised Model
Train
Trained Supervised Model (Austin)
STEP 1
QRadar Raw Event Stream
Event₁
Event₂
Eventₙ
Event Features (Input)
Probability-based Anomaly Detection Model
Train
Trained Unsupervised Model (Asif)
These models are trained OFFLINE and will send periodic model updates to the online QRadar app
MERGED OFFENSES APP
New Offense
Event₁
Event₂
Eventₙ
Input
Offense Categorization App
Trained Supervised Model (Austin)
Predict TP/FP
Trained Unsupervised Model (Asif)
Predict Anomaly Probability
Add to predicted list of offenses which are sorted on probability % and labelled with a TP/FP tag

Figure 9: A high-level visual overview of our proposed framework, showing the key steps in our pipeline.

1. **Time-and-Space-Efficient Data Extraction**: Data is collected from a production environment using QRadar's Ariel Query Language (AQL). Given the large volume of log data, queries are optimized to reduce time and space complexity. Aggregating data into 1-minute intervals helps manage the volume while retaining meaningful insights.
2. **Feature Extraction and Data Pre-processing**: Data from multiple log sources is processed, including separating log sources and applying noise reduction techniques. Events are aggregated using a Simple Moving Average (SMA) to smooth out noise, and sub-sequencing is applied to identify collective anomalies over time, rather than isolated points.

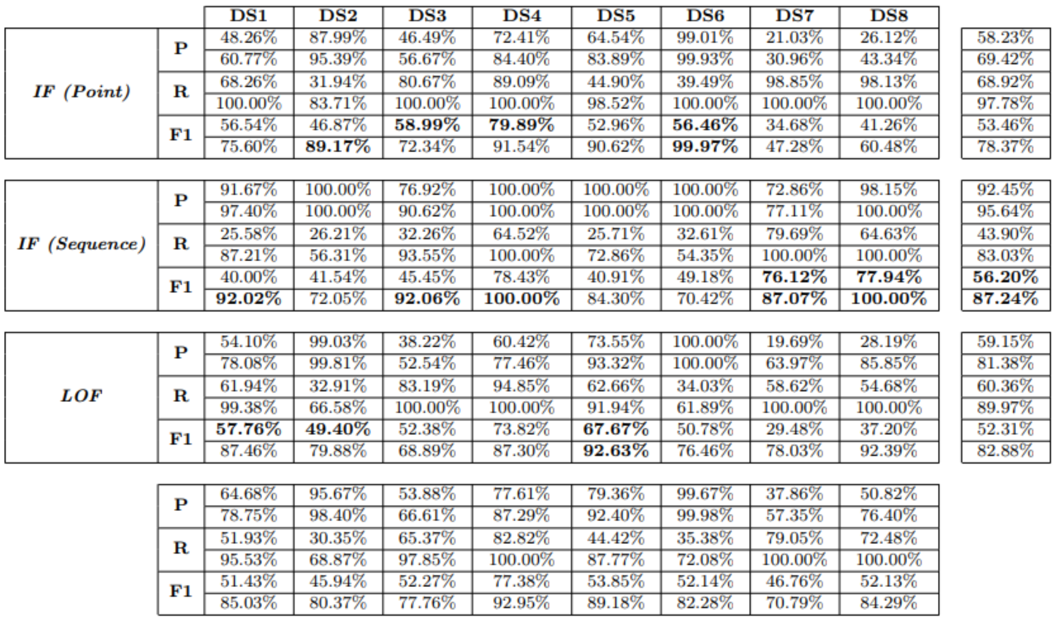
3. **Lightweight Model Selection**: The framework uses classical machine learning models for interpretability, essential for real-world deployment in SOC environments. Key models include Isolation Forest (iForest) for anomaly detection based on isolation trees and Local Outlier Factor (LOF) for detecting density-based anomalies.
4. **Point Adjustment for Anomaly Detection**: Post-processing involves point adjustment, a technique that focuses on detecting the presence of anomalies within sequences rather than isolated events. This adjustment prioritizes quick anomaly flagging over precise duration measurement, aiding analysts in faster threat identification.
5. **Hyper-parameter Tuning and Active Learning**: To optimize the models, an exhaustive grid search tunes hyper-parameters based on F1 scores. Active learning is employed, with feedback from analysts iteratively improving the model's performance. The most relevant parameters from our search:
   - Isolation Forest - contamination=0.03, n_estimators=300, max_samples=300
   - LOF - contamination=0.03, n_neighbors=10000
   - Sub-sequencing - window_length=45, stride=22 (50%)
   - SMA - window_length=20

Then we further collect detected offenses to implement our offense prioritization aspect of the tool. as shown in Figure 9 Upper Right) depicts the automatic offense prioritization system. As mentioned, this system is explicitly designed for QRadar SIEM and utilizes the training data from QRadar.

1. **Data Collection**: Approximately five million events were collected from QRadar, covering diverse attributes like event names, low-level categories, timestamps, and network data. This dataset was enhanced through API calls and AQL queries to retrieve accurate, comprehensive data.
2. **Feature Collection and Selection**: Features critical to detecting and prioritizing offenses were gathered, including attributes like event names, severity, credibility, usernames, and IP addresses. These features allow for a holistic analysis of each offense's potential impact.
3. **Probabilistic ML Models**: A suite of probabilistic models, such as Angle-Based Outlier Detection (ABOD), Cluster-based Outlier Probability (COPOD), and Stochastic Outlier Selection (SOS), were used. These models calculate prediction probabilities for offenses, enabling nuanced classification of potential threats.
4. **Automated Offense Prioritization**: Offenses are assigned scores based on model-generated probabilities and QRadar-calculated magnitudes. The resulting impact score prioritizes offenses by severity, allowing SOC analysts to focus on the highest-risk events.
5. **Experimental Evaluation**: Offenses were evaluated using metrics like Mean Time to Detect (MTTD) and Mean Time to Resolve (MTTR). The model performance was compared with baseline values to assess its effectiveness in real-time threat response.

Results:

Our tool demonstrated notable performance across eight datasets from a production SIEM environment, evaluated primarily through F1 scores to balance precision and recall. In Table 10, P, R, and F1 represent Precision, Recall, and F1 score respectively. For each metric, the top row represents the score with no point-adjustment applied, and the bottom row represents the score with point-adjustment applied. The cells separated on the right side of the table represent the averages across all datasets. Bold values represent the best performing model for each dataset. The separated cells on the bottom represent the averages performance of the tested models across each dataset.



		DS1	DS2	DS3	DS4	DS5	DS6	DS7	DS8	
IF (Point)	**P**	48.26%	87.99%	46.49%	72.41%	64.54%	99.01%	21.03%	26.12%	58.23%
		60.77%	95.39%	56.67%	84.40%	83.89%	99.93%	30.96%	43.34%	69.42%
	R	68.26%	31.94%	80.67%	89.09%	44.90%	39.49%	98.85%	98.13%	68.92%
		100.00%	83.71%	100.00%	100.00%	98.52%	100.00%	100.00%	100.00%	97.78%
	F1	56.54%	46.87%	**58.99%**	**79.89%**	52.96%	**56.46%**	34.68%	41.26%	53.46%
		75.60%	**89.17%**	72.34%	91.54%	90.62%	**99.97%**	47.28%	60.48%	78.37%
IF (Sequence)	**P**	91.67%	100.00%	76.92%	100.00%	100.00%	100.00%	72.86%	98.15%	92.45%
		97.40%	100.00%	90.62%	100.00%	100.00%	100.00%	77.11%	100.00%	95.64%
	R	25.58%	26.21%	32.26%	64.52%	25.71%	32.61%	79.69%	64.63%	43.90%
		87.21%	56.31%	93.55%	100.00%	72.86%	54.35%	100.00%	100.00%	83.03%
	F1	40.00%	41.54%	45.45%	78.43%	40.91%	49.18%	**76.12%**	**77.94%**	**56.20%**
		92.02%	72.05%	**92.06%**	**100.00%**	84.30%	70.42%	**87.07%**	**100.00%**	**87.24%**
LOF	**P**	54.10%	99.03%	38.22%	60.42%	73.55%	100.00%	19.69%	28.19%	59.15%
		78.08%	99.81%	52.54%	77.46%	93.32%	100.00%	63.97%	85.85%	81.38%
	R	61.94%	32.91%	83.19%	94.85%	62.66%	34.03%	58.62%	54.68%	60.36%
		99.38%	66.58%	100.00%	100.00%	91.94%	61.89%	100.00%	100.00%	89.97%
	F1	**57.76%**	**49.40%**	52.38%	73.82%	**67.67%**	50.78%	29.48%	37.20%	52.31%
		87.46%	79.88%	68.89%	87.30%	**92.63%**	76.46%	78.03%	92.39%	82.88%
	P	64.68%	95.67%	53.88%	77.61%	79.36%	99.67%	37.86%	50.82%	
		78.75%	98.40%	66.61%	87.29%	92.40%	99.98%	57.35%	76.40%	
	R	51.93%	30.35%	65.37%	82.82%	44.42%	35.38%	79.05%	72.48%	
		95.53%	68.87%	97.85%	100.00%	87.77%	72.08%	100.00%	100.00%	
	F1	51.43%	45.94%	52.27%	77.38%	53.85%	52.14%	46.76%	52.13%	
		85.03%	80.37%	77.76%	92.95%	89.18%	82.28%	70.79%	84.29%	

Table 10: Anomaly detection tool performance

- When paired with point adjustment and sub-sequencing techniques, **iForest** achieved the highest F1 scores across most datasets. This model proved highly effective in detecting diverse anomaly patterns within the data.
- LOF performed well in identifying outliers in high-density datasets, leveraging local density deviations. Although slightly outperformed by Isolation Forest in some scenarios, LOF remained a reliable model for anomaly detection.
- The tool achieved an average F1 score of 87.24% with **iForest**, indicating its ability to detect true anomalies accurately while minimizing false positives and negatives. LOF also demonstrated competitive precision and recall values, especially in data-rich event streams.

On the hand, when prioritizing offenses our tool demonstrated significant improvements in managing high-volume security alerts within the QRadar SIEM environment In Table11, the "Offense ID" uniquely identifies each offense. The "Average Prediction Probability" reflects the output from the ML models, while the "SOC Rating" is the severity level assigned by SOC analysts. The comparison between the ML models' prediction probabilities and SOC ratings sheds light on how likely offenses are to be true positives. For example, Offense 1169 has a high prediction probability of 0.95, aligning with its "Critical" SOC rating, suggesting a strong likelihood of it being a true positive. However, Offense 1212 also has a high prediction probability of 0.95, despite a "Low" SOC rating, showing cases where ML models predict high probabilities but differ from analysts' severity assessments due to additional context.

ID	COPOD	ECOD	IForest	Avg. Prediction Probability	SOC Rating
1169	1	1	0.85	0.95	Critical
1171	0.90	1	0.71	0.87	High
1189	0.85	1	0.59	0.813	Low
1197	0.85	1	0.64	0.83	High
1201	0.85	1	0.60	0.816	Moderate
1212	1	1	0.87	0.95	Low

Table 11: Comparison of ML Model Scores and SOC Ratings for Offenses

ID	COPOD	ECOD	IForest	Avg. Impact Score	SOC Rating
1169	1	1	0.8365	0.9455	Critical
1171	0.615	0.7589	0.6703	0.6814	High
1189	0.77227	1	0.5790	0.7837	Low
1197	0.9067	1	0.6740	0.860	High
1201	0.6327	0.8602	0.5630	0.6853	Moderate
1212	0.9975	1	0.8387	0.9454	Low

Table 12: Comparison of Impact Score and SOC Ratings for Offenses

Table 12 compares impact scores (representing the potential severity of offenses) with SOC ratings, revealing further insights. Different ML models generate varying impact scores for the same offense, reflecting differing assessments of severity. For example, Offense 1171 has a low average impact score of 0.6814 across models but is assigned a "High" SOC rating. This discrepancy suggests that while the models may indicate a lower severity, analysts consider other contextual factors that raise the offense's perceived severity level.

Summary:

We have outlined the development of an anomaly detection and offense prioritization tool in the QRadar SIEM environment to improve SOC operations. Our tool effectively identifies unusual patterns in data, while the offense prioritization system assigns impact scores to prioritize high-risk threats. Key findings show that these tools reduce detection and response times, enhance alignment with SOC analyst assessments, and demonstrate how ML-based analysis can complement human expertise.

Reference:

[GHS1] L. Breiman, "Random Forests," Machine Learning, vol. 45, no. 1, pp. 5-32, 2001.
[GHS2] C. Bishop, "Pattern Recognition and Machine Learning," Springer, 2006.
[GH3] X. Zhu et al., "Anomaly Detection in Network Traffic: A Machine Learning Perspective," IEEE Transactions on Network and Service Management, vol. 17, no. 1, pp. 87-101, 2020.
[GH4] S. M. M. Hossain, R. Couturier, J. Rusk, and K. B. Kent, "Automatic event categorizer for SIEM," Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering (CASCON '21), pp. 104-112, 2021.
[GH5] N. Mejri, L. Lopez-Fuentes, K. Roy, P. Chernakov, E. Ghorbel, and D. Aouada. "Unsupervised Anomaly Detection in Time-series: An Extensive Evaluation and Analysis of State-of-the-art Methods". In: arXiv preprint arXiv:2212.03637 (2022)

4.1.3. Anomaly detection for telemetric data (Hoxhunt)

Synopsis:

This section explores the development of a machine learning-based anomaly detection tool designed to monitor microservices in a Software as a Service (SaaS) environment. Microservices, characterized by their modular and distributed nature, require robust observability to detect and resolve anomalies that could affect system performance, security, or privacy. Our application integrates telemetry data collection using OpenTelemetry, provides a pluggable machine learning model infrastructure which initially employs unsupervised machine learning models—K-means clustering and Autoencoders—to identify anomalies, but allow for more sophisticated models to be inserted. Finally, it provides actionable insights via a visualization dashboard.

The tool aggregates telemetry data at the service handler and hour levels, enabling it to process critical metrics such as error messages, latency, request counts, error rates, and availability. These metrics are fed into the given machine learning models that detect the anomalies. A visualization interface highlights anomalies, aiding developers and support teams in promptly identifying and remediating potential issues.

The toolset is aimed to improve the detection of anomalies in microservices, enhancing operational reliability and efficiency.

Related Works:

Existing research on anomaly detection in distributed systems often relies on supervised learning methods that require labelled data. For instance, Zhou et al. [HH4] applied supervised models to telemetry data, achieving high detection accuracy. However, supervised approaches are impractical in dynamic, real-time environments due to the lack of labelled anomalies.

Unsupervised methods, such as K-means clustering and Autoencoders, have gained traction for their ability to identify patterns without labelled datasets. Chandola et al. [HH1] emphasized the challenges of anomaly detection, such as high false-positive rates and the heterogeneity of anomalies, which unsupervised models aim to address. Previous studies like those by Meng et al. [HH2] and Samir and Pahl [HH3] demonstrated the utility of machine learning in identifying anomalies through performance metrics and trace analysis, but often involved anomaly injection, a method unsuitable for production systems.

The tool research builds on these findings, focusing on practical implementations of unsupervised models in a live SaaS environment, leveraging telemetry data collected through OpenTelemetry.

Methodology:

The development of the anomaly detection tool involved the following high-level steps:

1. Telemetry Data Collection:
    - 1.1. Data was gathered using OpenTelemetry, capturing logs, traces, and metrics from the company’s microservices.
    - 1.2. Relevant metrics were identified, including latency, request counts, errors, availability, and time.
2. Data Aggregation and Preprocessing:
    - 2.1. Data was aggregated at the service handler and hour levels to provide actionable insights and reduce noise.
    - 2.2. Service handlers generating excessive noise were excluded to improve detection accuracy.
3. Model Implementation:
    - 3.1. K-means clustering: Utilized to group data points and detect outliers.
    - 3.2. Autoencoder: A neural network model trained to minimize reconstruction errors, flagging deviations as anomalies.
4. Visualization:
    - 4.1. Dashboards were developed using Apache Superset, displaying visualizations including heatmaps, word clouds, and detailed tables for detected anomalies.
5. Evaluation:
    - 5.1. Qualitative feedback from the team and quantitative analysis of past incidents were used as main metrics assess the tool’s performance in facilitating a more rapid iterative process

Results:

The anomaly detection tool was evaluated using both qualitative and quantitative methods:

1. Model Performance:
    1.1. K-means achieved an initial detection accuracy of 57.1%, outperforming the Autoencoder's 28.6%
    1.2. Both models detected different types of anomalies, indicating complementarity
2. Efficiency:
    2.1. Handler-level aggregation significantly reduced noise compared to trace-level detection.
    2.2. The models demonstrated robustness in identifying anomalies in a dynamic environment.
3. Actionability:
    3.1. The visualization dashboard provided clear and actionable insights, augmenting the capabilities of the development and support teams to locate and investigate anomalies efficiently.

Summary:

By combining telemetry data collection, unsupervised learning models, and a user-friendly visualization interface, the telemetry anomaly analyzer tool addresses critical challenges in monitoring distributed systems in a SaaS environment. While the suite of tools is designed for a specific use case, the methods and findings can be generalized to other contexts, highlighting the potential of unsupervised machine learning in improving software observability. Future work includes enhancing model robustness, exploring additional algorithms, and scaling the solution for broader applications.

References:

[HH1] Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. ACM computing surveys (CSUR), 41(3), 1–58.
[HH2] Meng, L., Ji, F., Sun, Y., & Wang, T. (2021). Detecting anomalies in microservices with execution trace comparison. Future Generation Computer Systems, 116, 291–301.
[HH3] Samir, A., & Pahl, C. (2019). Dla: Detecting and localizing anomalies in containerized microservice architectures using markov models. In 2019 7th international conference on future internet of things and cloud (ficloud) (pp. 205–213).
[HH4] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., . . . He, C. (2019). Latent error prediction and fault localization for microservice applications by learning from system trace logs. In Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering (pp. 683–694).

4.2. Automatic Code Analysis and Change Impact Analysis Approaches

4.2.1. Improved knowledge sharing among developers using automatic metrics collection from version control systems for impact analysis (ERSTE, DAKIK, Kuveyt Turk, Cape of Good Code)

Synopsis:

This section investigates the use of automatic code analysis and change impact analysis as a mechanism to facilitate knowledge sharing among developers by leveraging metrics automatically collected from version control systems (VCS). The research focuses on designing an integrated toolchain that collects, analyzes, and visualizes code metrics, such as knowledge sharing risks, churn rates, commit frequencies, code complexity, and module dependencies. This tool provides actionable insights for understanding the impact of code changes on software quality, team productivity, and project timelines. The goal is to empower development teams to make informed decisions while maintaining high code quality and team coherence.

The approach uses automated techniques to extract metrics from VCS repositories, combined with tools to parse code in code entity level such as classes and functions. A visualization dashboard enables developers to identify high-risk modules, track code evolution, and understand the broader implications of their contributions. This framework is designed to enhance collaboration, minimize technical debt, and improve knowledge transfer within software teams.

Related works:

Research in automated code analysis and change impact analysis has focused on improving software quality through static and dynamic analyses. Notable studies by Mockus and Weiss [TURK1] introduced the concept of mining version histories to identify defect-prone modules. Similarly, Hassan [TURK2] proposed leveraging historical metrics from VCS to predict maintenance effort and software reliability. These works established the foundation for using data-driven techniques to analyze software artifacts.

Recent advancements include tools like CodeScene and SonarQube, which analyze codebase health and identify hotspots but often lack integration with predictive models or actionable insights tailored to change impact. Studies by Gousios et al. [TURK3] emphasized the importance of real-time metrics for pull request evaluation, while Zimmermann et al. [TURK4] explored coupling metrics and their role in change propagation.

DETANGLE is another recent effort focusing on knowledge distribution within teams through collaborative tools that emphasize shared understanding of code changes.

Their platform provides mechanisms for identifying knowledge gaps, distributing expertise across teams.

This research builds on these efforts by incorporating automated metrics collection, predictive analysis, and visualization tailored for knowledge sharing and team productivity. By focusing on the social and technical dimensions of code changes, this work bridges the gap between static analysis tools and collaborative development needs.

Methodology:

The proposed methodology for automatic code analysis and change impact analysis comprises the following steps:

1- Data Collection:

- Automatically collect metrics from VCS (e.g., Git), focusing on commit histories, authorship, code churn, file dependencies, and test coverage.
- Extract additional metadata, such as timestamps, branch information, and merge histories, to contextualize changes.
- Extract of code entities (both classes and functions) from the source files, as well as their associated callee and caller relationships with other code entities, is achieved using a specialized tool, Understand, provided by SciTools
- Collect PRs and reviewers of the PRs

2- Data Preprocessing:

- Normalize and aggregate metrics at the module and project levels to reduce noise and ensure consistency.
- Apply filtering to exclude non-informative commits (e.g., formatting changes or comments).
- Establish associations between commits and user stories through various means, including analyzing commit messages, pull request metadata, and linked issue references.
- Map reviewers to the commits they reviewed via pull requests and subsequently link them to the corresponding files. This mapping supports knowledge distribution by providing insights into the collaborative ownership of code.
- Resolve discrepancies in developer records caused by Git's reliance on environment-provided email addresses, consolidating entries for the same individual across different aliases. This is executed utilizing an internally developed module known as 'Similar Contributor Matching.' This module scrutinizes the username portion of every contributor's email address (preceding the @ symbol), in addition to the person's name. It correlates these elements across the totality of project contributors, thus detecting any overt similarities. These entities, having surpassed a predetermined threshold of similarity, are then displayed through our User Interface. Prior to the commencement of the analysis, these findings are matched and consolidated for improved data harmonization and precision.

3- Visualization and Reporting:

- Develop dashboards that display code health, hotspots, and dependency maps.
- Provide insights into knowledge silos and developer collaboration patterns to highlight areas needing improved communication.

4- Evaluation:

- Validate the approach using historical project data and feedback from development teams.

Results:

By leveraging the integrated toolchain consisting of automated metrics collection, predictive analysis, and visualization, significant improvements in knowledge sharing among developers were realized. This research's implementation in Kuveyt Turk and Vaadin provided key insights:

Knowledge Transfer Efficiency:
The implementation identified code areas with elevated risk values and low cohesion indicative of knowledge silos. Prioritizing these areas for information sharing resulted in improved communication and decreased knowledge gaps within the team.

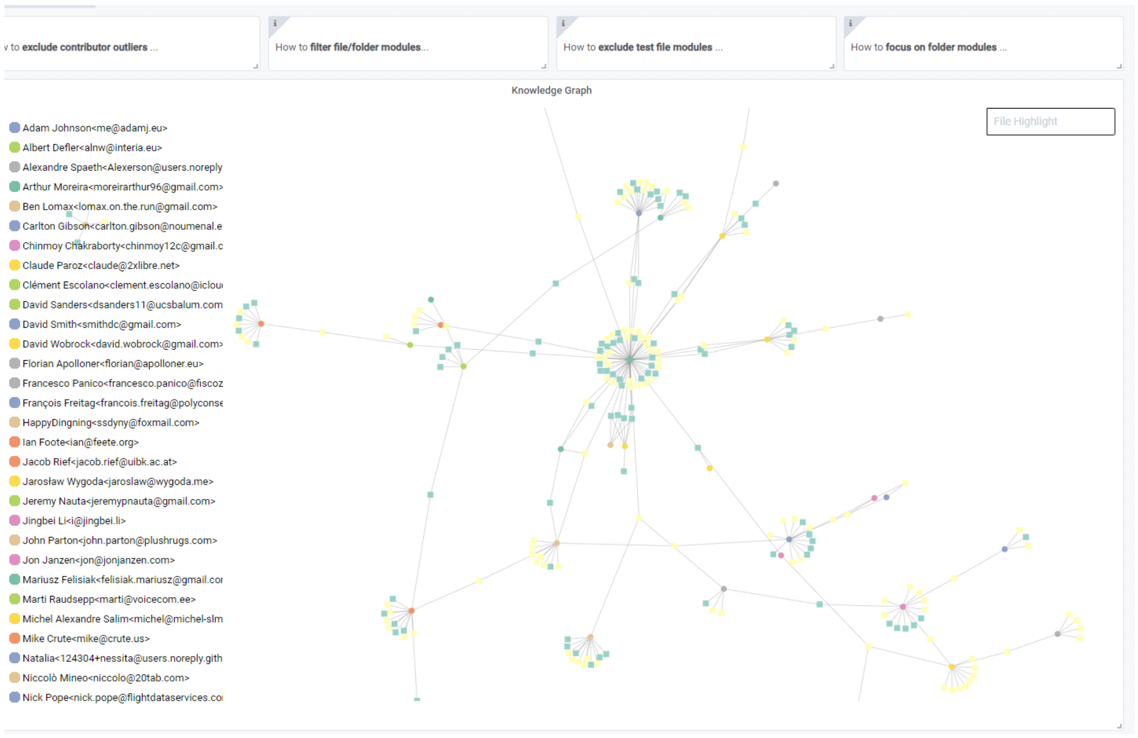
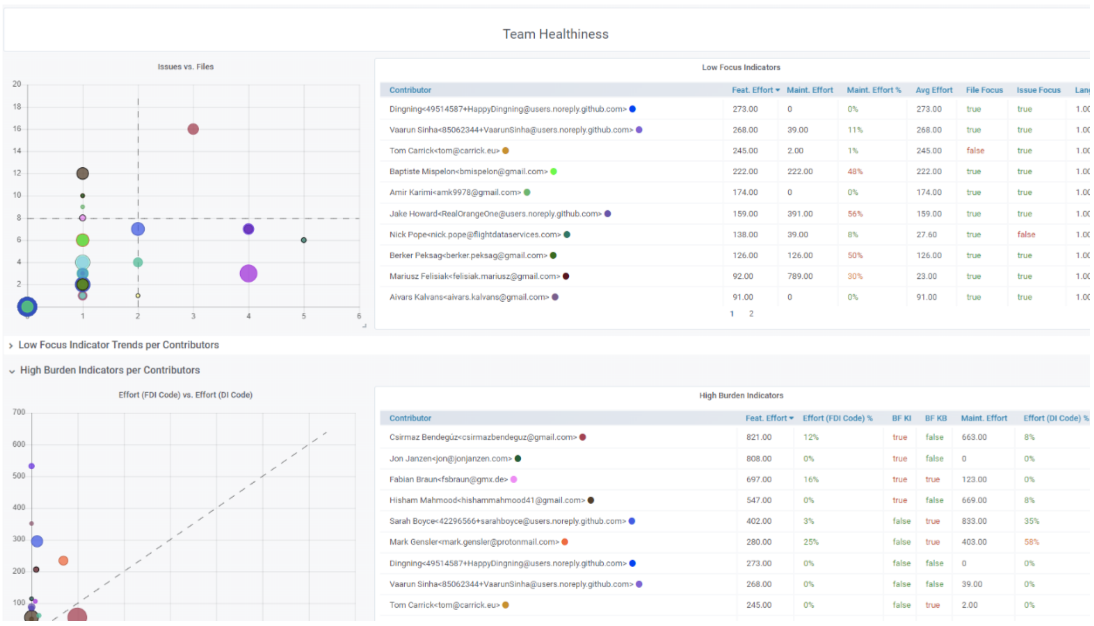

v to exclude contributor outliers ...
How to filter file/folder modules...
How to exclude test file modules ...
How to focus on folder modules ...
Knowledge Graph
File Highlight
Adam Johnson<me@adamj.eu>
Albert Defler<alnw@interia.eu>
Alexandre Spaeth<Alexerson@users.noreply
Arthur Moreira<moreirarthur96@gmail.com>
Ben Lomax<lomax.on.the.run@gmail.com>
Carlton Gibson<carlton.gibson@noumenal.e
Chinmoy Chakraborty<chinmoy12c@gmail.c
Claude Paroz<claude@2xlibre.net>
Clément Escolano<clement.escolano@iclou
David Sanders<dsanders11@ucsbalum.com
David Smith<smithdc@gmail.com>
David Wobrock<david.wobrock@gmail.com>
Florian Apolloner<florian@apolloner.eu>
Francesco Panico<francesco.panico@fiscoz
François Freitag<francois.freitag@polyconse
HappyDingning<ssdyny@foxmail.com>
Ian Foote<ian@feete.org>
Jacob Rief<jacob.rief@uibk.ac.at>
Jarosław Wygoda<jaroslaw@wygoda.me>
Jeremy Nauta<jeremypnauta@gmail.com>
Jingbei Li<i@jingbei.li>
John Parton<john.parton@plushrugs.com>
Jon Janzen<jon@jonjanzen.com>
Mariusz Felisiak<felisiak.mariusz@gmail.co
Marti Raudsepp<marti@voicecom.ee>
Michel Alexandre Salim<michel@michel-slm
Mike Crute<mike@crute.us>
Natalia<124304+nessita@users.noreply.gith
Niccolò Mineo<niccolo@20tab.com>
Nick Pope<nick.pope@flightdataservices.co

Figure 10: Knowledge Sharing Network Diagram

Aided Decision Making:

Metrics such as churn rate and commit frequency provided insights into the most volatile areas of the codebase, emphasizing the need for process improvements.

Visualizations and Informed Decisions:
The visualization dashboard enabled developers to track code evolution and understand the broader implications of their code contributions. These graphical displays made it easier for developers to identify high-risk modules, effectively mitigating potential future issues.


Team Healthiness
Issues vs. Files
Low Focus Indicator Trends per Contributors
High Burden Indicators per Contributors
Effort (FDI Code) vs. Effort (DI Code)

Low Focus Indicators

Contributor	Feat. Effort	Maint. Effort	Maint. Effort %	Avg Effort	File Focus	Issue Focus	Lan
Dingning<49514587+HappyDingning@users.noreply.github.com> ●	273.00	0	0%	273.00	true	true	1.0
Vaarun Sinha<85062344+VaarunSinha@users.noreply.github.com> ●	268.00	39.00	11%	268.00	true	true	1.0
Tom Carrick<tom@carrick.eu> ●	245.00	2.00	1%	245.00	false	true	1.0
Baptiste Mispelon<bmispelon@gmail.com> ●	222.00	222.00	48%	222.00	true	true	1.0
Amir Karimi<amk9978@gmail.com> ●	174.00	0	0%	174.00	true	true	1.0
Jake Howard<RealOrangeOne@users.noreply.github.com> ●	159.00	391.00	56%	159.00	true	true	1.0
Nick Pope<nick.pope@flightdataservices.com> ●	138.00	39.00	8%	27.60	true	false	1.0
Berker Peksag<berker.peksag@gmail.com> ●	126.00	126.00	50%	126.00	true	true	1.0
Mariusz Felisiak<felisiak.mariusz@gmail.com> ●	92.00	789.00	30%	23.00	true	true	1.0
Aivars Kalvans<aivars.kalvans@gmail.com> ●	91.00	0	0%	91.00	true	true	1.0

1 2

High Burden Indicators

Contributor	Feat. Effort	Effort (FDI Code) %	BF KI	BF KB	Maint. Effort	Effort (DI Code) %
Csirmaz Bendegúz<csirmazbendeguz@gmail.com> ●	821.00	12%	true	false	663.00	8%
Jon Janzen<jon@jonjanzen.com> ●	808.00	0%	true	false	0	0%
Fabian Braun<fsbraun@gmx.de> ●	697.00	16%	true	true	123.00	0%
Hisham Mahmood<hishammahmood41@gmail.com> ●	547.00	0%	true	false	669.00	8%
Sarah Boyce<42296566+sarahboyce@users.noreply.github.com> ●	402.00	3%	false	true	833.00	35%
Mark Gensler<mark.gensler@protonmail.com> ●	280.00	25%	false	true	403.00	58%
Dingning<49514587+HappyDingning@users.noreply.github.com> ●	273.00	0%	false	false	0	0%
Vaarun Sinha<85062344+VaarunSinha@users.noreply.github.com> ●	268.00	0%	false	false	39.00	0%
Tom Carrick<tom@carrick.eu> ●	245.00	0%	false	true	2.00	0%

Figure 11: Team Healthiness tables and bubble charts

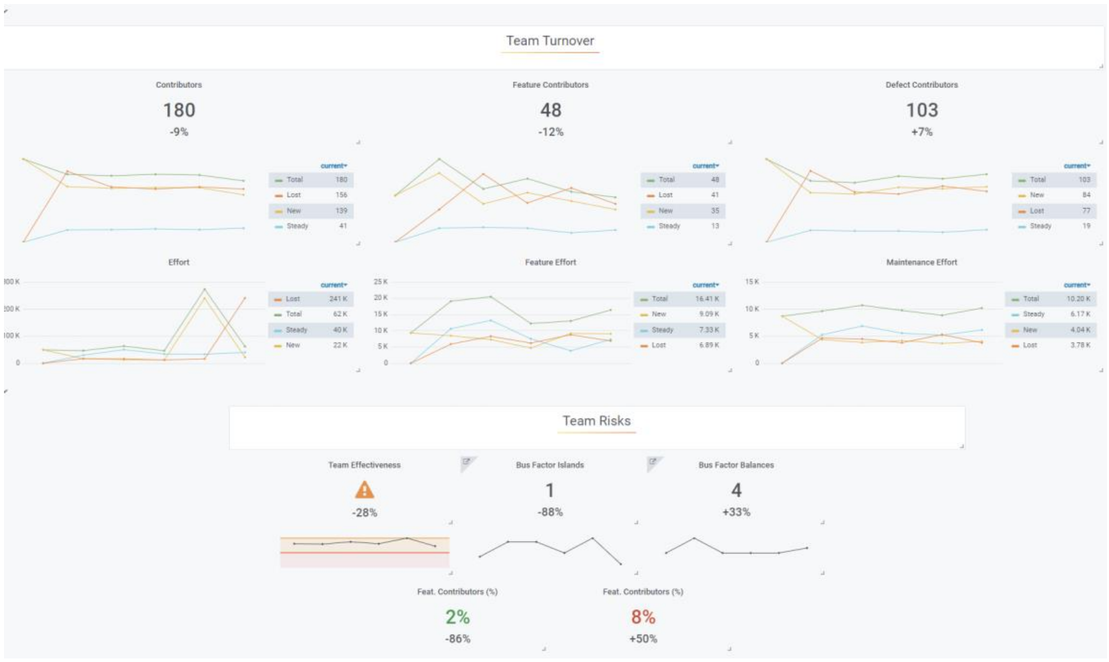
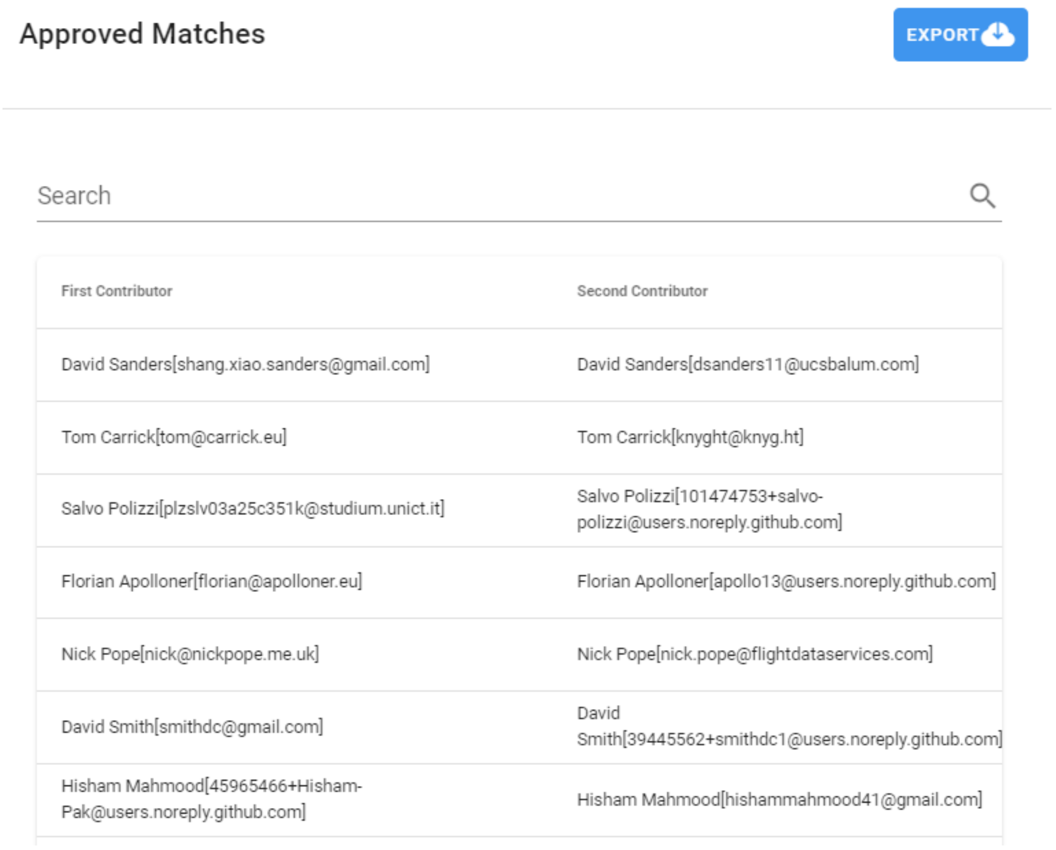


Figure 12: Knowledge Risks and Team Turnover dashboard

Data Harmonization:

The 'Similar Contributor Matching' module established connections between developers' records, thus creating a cohesive and holistic view for improved decision making. This process mitigated discrepancies caused by Git's reliance on environment-supplied email addresses, indicating how automation significantly enhances data accuracy.



First Contributor	Second Contributor
David Sanders[shang.xiao.sanders@gmail.com]	David Sanders[dsanders11@ucsbalum.com]
Tom Carrick[tom@carrick.eu]	Tom Carrick[knyght@knyg.ht]
Salvo Polizzi[plzslv03a25c351k@studium.unict.it]	Salvo Polizzi[101474753+salvo-polizzi@users.noreply.github.com]
Florian Apolloner[florian@apolloner.eu]	Florian Apolloner[apollo13@users.noreply.github.com]
Nick Pope[nick@nickpope.me.uk]	Nick Pope[nick.pope@flightdataservices.com]
David Smith[smithdc@gmail.com]	David Smith[39445562+smithdc1@users.noreply.github.com]
Hisham Mahmood[45965466+Hisham-Pak@users.noreply.github.com]	Hisham Mahmood[hishammahmood41@gmail.com]

Figure 13: Matched contributors of open source Django project

Summary:

This research has proven its value in enhancing knowledge sharing among developers in a software team environment using automatic code analysis and change impact analysis. The implementation of this research in Kuveyt Turk significantly transformed their development process.

Key outcomes realized include:

Improved Knowledge Transfer: The research pinpointed areas with high knowledge sharing risks allowing developers to focus their efforts on these specific areas.

Data Harmonization: By effectively leveraging the 'Similar Contributor Matching' module, the system was able to align entries of the same individual across different aliases, providing a holistic view of individual contributions.

Enhanced Decision-Making: The implementation of visualization dashboards enabled developers to understand the broader implications of their code contributions, leading to more informed decisions.

This research's integration into intra-team knowledge sharing has demonstrated its ability to drive meaningful improvements in software quality and team coherence. It bridges the gap between static analysis tools and collaborative development needs, positioning it as a valuable tool for developers working on complex projects.

References:

[TURK1] A. Mockus and D. M. Weiss, "Predicting risk of software changes," in Bell Labs Technical Journal, vol. 5, no. 2, pp. 169-180, April-June 2000, doi: 10.1002/bltj.2229.
[TURK2] A. E. Hassan, "The road ahead for Mining Software Repositories," 2008 Frontiers of Software Maintenance, Beijing, China, 2008, pp. 48-57, doi: 10.1109/FOSM.2008.4659248.
[TURK3] D. Mitropoulos, G. Gousios, P. Papadopoulos, V. Karakoidas, P. Louridas and D. Spinellis, "The Vulnerability Dataset of a Large Software Ecosystem," 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), Wroclaw, Poland, 2014, pp. 69-74, doi: 10.1109/BADGERS.2014.8.
[TURK4] T. Zimmermann, A. Zeller, S. Diehl and P. Weißgerber, "Mining Version Histories to Guide Software Changes" in IEEE Transactions on Software Engineering, vol. 31, no. 06, pp. 429-445, June 2005, doi: 10.1109/TSE.2005.72.

4.2.2. Automatic collection of code analysis metrics of cloud-based software and faults predictions (Ontario Tech, Team Eagle)

Synopsis:

Given the intricate composition and complex nature of airfield operations software, it is paramount to ensure sufficient software quality throughout its entire development life cycle. We have developed an automated solution for SQA metrics acquisition and the analysis of quality-related data for a real-world airfield software system. The end goal of this endeavour is to facilitate an end-to-end framework, composed of distinct tools and pipelines, to automate the extraction of software quality metrics from an airfield software system, analyze the historical metrics data over time, and perform predictive analysis using machine-learning approaches.

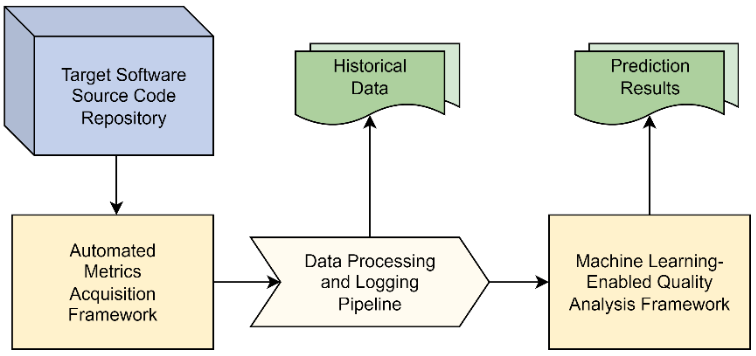
Related works:

The Automated Metrics Acquisition Framework comprises a selection of distinct tools, each of which is responsible for separate functionality. The technologies used are summarized below.

SonarQube Community Edition (open source) SonarQube source continuous inspection platform for software source code [TE1]. SonarQube provides the capability to perform automatic code reviews to detect bugs, code smells, and security vulnerabilities in numerous programming languages [TE1]. SonarQube can perform static code analysis and offers various metrics to help developers improve the quality of their code [TE1]. In this framework, SonarQube is used for collecting quality-related metrics from the project repository using static-analysis principles. The metrics being collected are related to bugs, code smells, code vulnerabilities, and security hotspots.
Azure Repos Azure Repos is a version control service provided by Microsoft Azure, designed to help development teams manage and track changes to their source code [2]. Azure Repos is integrated with other Azure DevOps services, providing a comprehensive solution for the entire development life cycle [TE2]. The reason for adopting the Azure DevOps tool suite is because the target software system utilizes Azure's DevOps services, and the source code resides in an Azure Repos cloud repository.
Azure VM Azure Virtual Machines (VMs) are on-demand, scalable computing resources provided by Microsoft Azure [TE3]. These VMs run in the cloud and allow users to deploy and manage virtualized Windows or Linux servers [TE3]. In the context of this framework, an Azure VM is used for hosting a SonarQube instance and executing data extraction and logging scripts.
Azure Pipelines Azure Pipelines is a cloud-based continuous integration and continuous delivery (CI/CD) service provided by Microsoft Azure [TE4]. It enables developers to automate the building, testing, and deployment of applications [TE4]. In this framework, it is used to facilitate the end-to-end process when triggered by repository changes or a scheduled trigger.

Methodology:

This section introduces a comprehensive system model that serves as a conceptual framework for understanding the solution being developed. This system model visually represents the major components, relationships, and processes of the system.


Target Software Source Code Repository
Historical Data
Prediction Results
Automated Metrics Acquisition Framework
Data Processing and Logging Pipeline
Machine Learning-Enabled Quality Analysis Framework
Figure 14: System Model

The system model comprises three principal components: the automated metrics acquisition framework (AMAF), the data processing and logging pipeline (DPLP), and the machine learning-enabled quality analysis framework (MLQAF). Each of these components operates autonomously, exists in isolation, and engages in data communication following a controlled flow delineated by the relationships shown in Figure 1. The repository housing the source code of the target software serves as the input of the system, with the system generating historical data logs and machine learning predictions as its output artifacts.

Automated Metrics Acquisition Framework
The source code of the target software system resides within an external, cloud-hosted repository, where the AMAF accesses this repository through secure tokens employed for authentication and communication. The AMAF constitutes a set of distinct and autonomous tools, as detailed in the subsequent sections. For the system model, the AMAF is treated as a singular component from the scope of the overarching system. The architecture supporting the Automated Metrics Acquisition Framework can be seen on Figure 2.

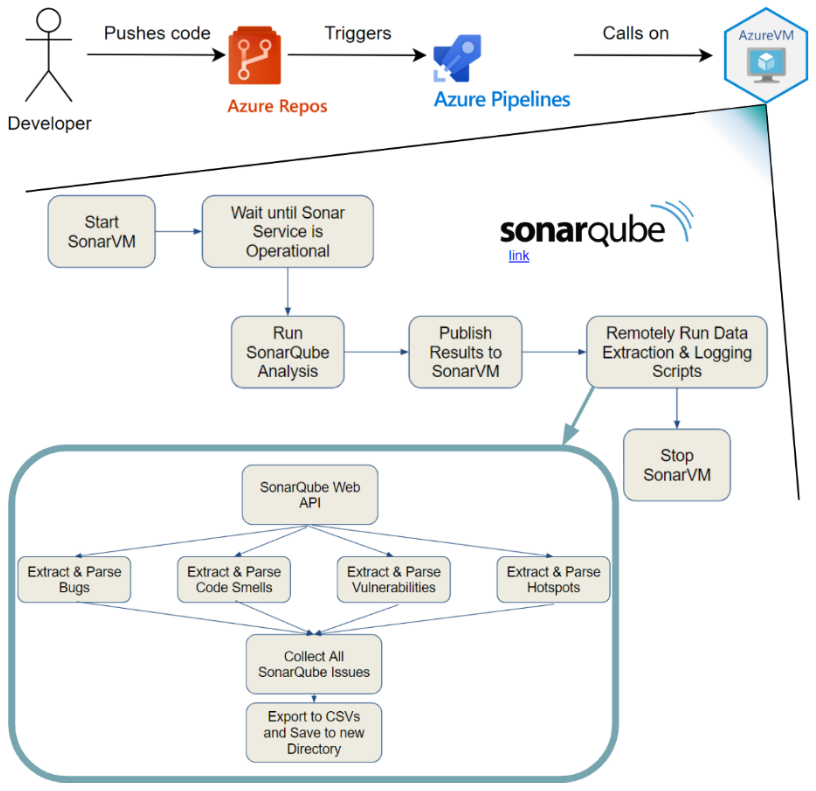

Developer
Pushes code
Azure Repos
Triggers
Azure Pipelines
Calls on
AzureVM
Start SonarVM
Wait until Sonar Service is Operational
sonarqube
link
Run SonarQube Analysis
Publish Results to SonarVM
Remotely Run Data Extraction & Logging Scripts
Stop SonarVM
SonarQube Web API
Extract & Parse Bugs
Extract & Parse Code Smells
Extract & Parse Vulnerabilities
Extract & Parse Hotspots
Collect All SonarQube Issues
Export to CSVs and Save to new Directory

Figure 15: AMAF Architecture

Data Processing and Logging Pipeline
The DPLP functions as an independent tool, comprising a collection of scripts designed to process the raw data produced by the AMAF. This pipeline systematically generates refined and organized metrics, culminating in historical data intended for subsequent analysis. Alternatively, historical data logs can be treated as standalone output artifacts.

Machine Learning-enabled Quality Analysis Framework

The MLQAF employs historical data logs as input and deploys machine learning models to be designated and trained on a selected series of data. This automated process features preconfigured parameters for the models, ensuring consistency in both input and output dimensions. Currently, the MLQAF produces quality trends or classification predictions as output artifacts. The MLQAF currently consists of four predictive models and one generative model. The models used were as follows: Predictive Models, Linear Regression, Logistic Regression, Decision Tree Regression, Isolation Forest, Generative Adversarial Network (GAN). The 3 regression models are used making quality trend predictions based on historical data, the isolation forest model is used to perform anomaly detection on API testing results, and the generative adversarial network (GAN) is used for producing significantly larger datasets to assess the long-term feasibility of the various models.

Results:

The versatility of the Machine-learning-enabled Quality Analysis Framework (MLQAF) opens avenues for diverse applications in software quality assurance. This section explores a collection of sample use cases (UCs) that showcase the adaptability and effectiveness of the MLQAF in addressing various challenges within the software development lifecycle.

UC 1: Quality Trend Analysis
The MLQAF serves as a powerful tool for conducting in-depth quality trend analysis, allowing developers to gain valuable insights into the evolution of software quality metrics over time. By leveraging historical data, the MLQAF predicts future quality trends, allowing teams to proactively address potential issues and optimize software quality throughout the development cycle.

UC 2: Anomaly Detection on API Response Times
All methods of assessing the airfield software's quality thus far have been focused on static analysis techniques. Work is currently being done to extend the system to also include run time performance metrics, the first selected being response times made to API endpoints on the airfield software's back-end business logic. Performing anomaly detection on the API response times consists of a two-step process, performing the API tests themselves, and applying an anomaly detection model on the logged data.

UC 3: Assessment of Model Feasibility
As mentioned previously, the notion of a generative adversarial network (GAN) was applied in this system to assess the long-term feasibility of various prediction models. This is done by substantially increasing the input data pool sizes for the training of models substantially and observing the effects on prediction accuracy.

Found below are some sample results from a quality trend analysis performed using a series of metrics collected from Team Eagle's airfield software.

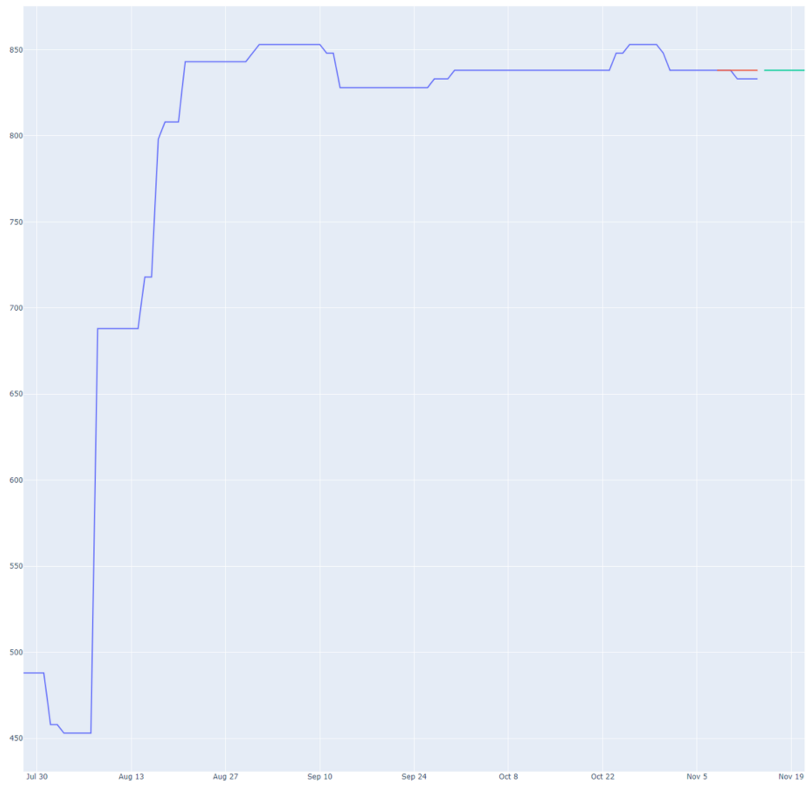


Figure 16: Developer Effort to Resolve All Bugs (Logistic Regression n)

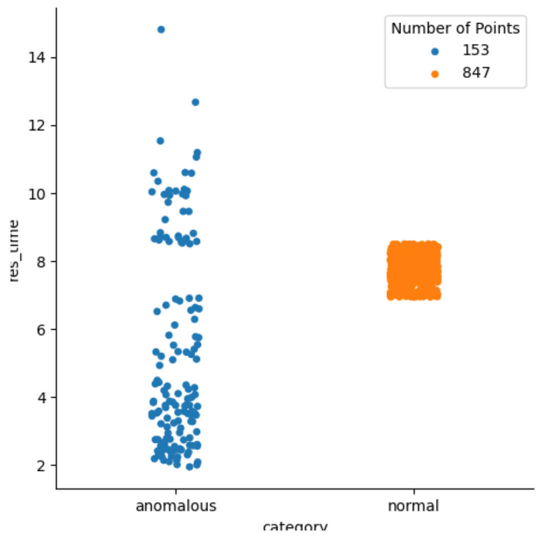


Figure 17: Sample Anomaly Detection Results (Isolation Forest)

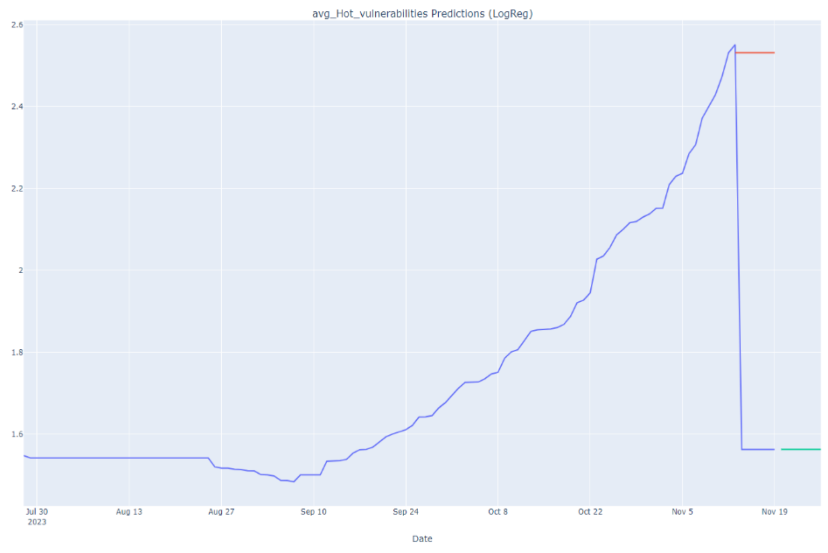


Figure 18: Average Hotspot Vulnerabilities with Additional GAN-generated Data (Logistic Regression)

Summary:

The Automated Metrics Acquisition Framework, leveraging tools like SonarQube, Azure Repos, and Pipelines, automates the extraction of software quality metrics from the airfield software system. The Data Processing and Logging Pipeline consists of intermediary procedures which process the data, visualizes metrics, and facilitates quality trend analysis. The Machine Learning-enabled Quality Analysis Framework utilizes various machine learning models, including linear regression, logistic regression, decision tree regression, and isolation forest, to predict, classify, and analyze quality trends. Additionally, a Generative Adversarial Network (GAN) is employed to assess the long-term feasibility of predictive models by augmenting input datasets. The sample results demonstrate the effectiveness of the system, showcasing predictions made by different regression models and the impact of increase input data pools using GAN-generated data. Anomaly detection has been applied on API response times using the isolation forest model, providing insights into deviations from expected behavior.

In conclusion, the solution in development offers a comprehensive approach to automate SQA processes, enhance data-driven decision-making, and improve the overall quality of software systems, such as Team Eagle Ltd.'s airfield software. Future work includes refining models, experimenting with additional models, further incorporating run-time metrics, and extending the system's capabilities to further advance automated software quality assurance.

References:

[TE1] “Code quality tool & secure analysis with SonarQube,” Clean Code: Writing Clear, Readable, Understandable & Reliable Quality Code, https://www.sonarsource.com/products/sonarqube/.

[TE2] Vijayma, “Collaborate on code - Azure Repos,” learn.microsoft.com. https://learn.microsoft.com/en-us/azure/devops/repos/get-started/what-is-repos?view=azure-devops

[TE3] Cynthn, “Overview of virtual machines in Azure - Azure Virtual Machines,” learn.microsoft.com. https://learn.microsoft.com/en-us/azure/virtual-machines/overview
[TE4] Juliakm, “What is Azure Pipelines? - Azure Pipelines,” learn.microsoft.com. https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops

4.2.3. Automatic analysis of technical debts (Cape of Good Code, Vaadin)

Synopsis:

This section focuses on the DETANGLE Analysis Suite, a part of the broader SmartDelta Quality Optimization and Recommendation Methodology. It offers automated quality assurance tailored for incremental industrial software development, focusing on the detection and management of different types of technical debt.

DETANGLE computes Key Performance Indicators (KPIs) such as Maintenance Effort, Feature Effort, Feature Effort Effectiveness to evaluate them as symptoms of Technical Debt. Additionally, it provides architecture health factor metrics like Feature Debt and Contributor Friction. It enables development teams to monitor software quality trends, identify architectural hotspots (by correlating KPIs and health factors), and prioritize refactoring efforts. By integrating data from code repositories, issue trackers, testing and DevOps tools, DETANGLE provides a comprehensive analysis of development activities, facilitating informed decisions on cost-effective quality enhancements.

Within SmartDelta's framework, DETANGLE contributes to software quality trend analysis and prediction by quantifying and visualizing the impact of technical debt on modularity, maintainability, and extensibility. Its application to the Vaadin Flow framework has demonstrated its effectiveness in identifying architectural bottlenecks and guiding refactoring decisions, supporting long-term quality improvement.

Related Work:

Technical debt management is a widely researched area in software engineering. Existing tools, such as SonarQube, provide static code analysis to identify code smells, duplication, and complexity [VD1]. While such tools effectively highlight issues, they often lack the ability to predict trends or provide actionable recommendations for addressing architectural and modular challenges.

In the context of the SmartDelta project, DETANGLE has been applied to analyze software quality trends and identify architectural hotspots. Its ability to quantify and visualize the impact of technical debt on modularity and maintainability extends beyond traditional approaches, offering targeted recommendations for improving code quality on architecture and design level. DETANGLE further integrates test coverage and review activity data into its analysis, thus including process quality into its comprehensive evaluation of the interplay between the symptoms and root causes of technical debt [VD2].

During its evaluation on the Vaadin Flow framework, DETANGLE identified critical areas requiring refactoring and offered actionable insights to guide design improvements. By addressing gaps in traditional analysis tools, DETANGLE has proven effective in supporting long-term quality improvements and fostering informed decision-making in incremental industrial software development [VD3].

Methodology:

The DETANGLE Analysis Suite employs a comprehensive, data-driven methodology to analyze technical debt, architecture health, and team collaboration in software systems. Its approach integrates various data sources and computes a wide range of metrics, providing actionable insights into software quality and maintainability.

- Data Integration
  - DETANGLE gathers data from development tools to capture a comprehensive view of code changes, issue resolutions, and testing activities. Examples include repositories for tracking modifications, issue tracking systems for development activities, and optional inputs like code quality or testing frameworks for detailed insights. This integration ensures a holistic understanding of both technical and collaborative aspects of software projects.
- Technical Debt KPIs and Health Factor Metrics
  - DETANGLE calculates metrics across multiple categories to provide an overall view of software quality:
    - Effort KPIs: Maintenance Effort %, Primary Effort %, Primary Effort Effectiveness, and Maintenance Effort Ineffectiveness.
    - Architecture Health: Metrics such as the Feature Debt Index (Primary/Feature Debt Index), Contributor Friction Index, Defect Density, and Defect Impact assess modularity, maintainability, and extensibility.
    - Code Health: Derived from tools like SonarQube, these include complexity, duplicated lines, maintainability, reliability, and security findings.
    - Technical Debt Effort Prediction: Provides estimates for addressing code- (integrating the prediction from tools like SonarQube) and architecture-related debt.
- Team Collaboration Metrics
  - DETANGLE evaluates team health and collaboration through:

        - Team KPIs: Metrics like Team Fluctuation/Turnover and Team Effectiveness assess productivity and knowledge retention.
        - Team Health Factors: Bus Factor Knowledge Islands, Knowledge Balances, Coordination, and Healthiness highlight areas for team improvement and risk mitigation.
- Visualization
    - DETANGLE generates detailed visualizations, including:
        - Network Graphs: Show dependencies and coupling at feature and contributor levels.
        - Architecture Dashboards: Provide insights into modularity challenges and architectural health.
        - Collaboration Visuals: Help teams identify risks in knowledge sharing and collaboration. These tools enable root cause analysis and help prioritize targeted refactoring and team interventions.
- Cost/Benefit Analysis
    - DETANGLE predicts the effort required to remediate technical debt and measures the potential benefits (like reduced maintenance effort or higher feature effort effectiveness), enabling cost/benefit analyses for informed decision-making. This allows teams to balance short-term fixes against sustainable long-term improvements.
- Trend Analysis
    - DETANGLE tracks changes in metrics over time to identify trends in software quality and team collaboration. This historical perspective helps teams proactively manage technical debt and maintain consistent software health.

Results:

The DETANGLE Analysis Suite was applied to the Vaadin Flow framework to identify technical debt hotspots and provide actionable recommendations for legacy code refactoring. Key findings are outlined below:

- Architectural Hotspots
    - DETANGLE identified specific areas in the flow-server/frontend module with elevated Feature Debt Index values, highlighting strong feature coupling and low cohesion. These hotspots were prioritized for refactoring to improve modularity and reduce unintended side effects during new feature development.
    - Metrics such as Defect Density and Defect Impact pinpointed files and folders most susceptible to recurring bugs and follow-up issues, emphasizing the need for architectural improvements.
- Refactoring Recommendations
    - Using network graphs and modularity analysis, DETANGLE provided detailed insights into problematic areas of the codebase. Recommendations included:
        - Splitting complex source files to improve cohesion.

        - Extracting and reorganizing code into new, more modular components.
        - Addressing feature and contributor coupling to enhance architectural extensibility.
    - These recommendations enabled the Vaadin team to focus their efforts on refactoring high-impact areas of legacy code.
- Visualizations and Root Cause Analysis
    - DETANGLE's architecture dashboards and network graphs facilitated a clear understanding of feature and contributor dependencies. These visualizations were useful in identifying the root causes of technical debt and planning refactoring strategies.
- Impact on Legacy Code Refactoring
    - DETANGLE supported the Vaadin team in identifying and addressing technical debt in legacy modules, ensuring that the codebase became more modular and maintainable. By focusing on architectural hotspots, the team reduced risks associated with feature development and maintenance in the refactored areas.

Summary:

The DETANGLE Analysis Suite has proven its value as a tool for identifying and managing technical debt, specifically in the context of legacy code refactoring. Its application to the Vaadin Flow framework, as part of the SmartDelta Quality Optimization and Recommendation Methodology, provided actionable insights into architectural hotspots and guided effective refactoring efforts.

Key outcomes include:

- Identification of Architectural Hotspots: DETANGLE pinpointed modules with high Feature Debt Index values and elevated defect density, helping the Vaadin team focus on critical areas for legacy code refactoring.
- Targeted Refactoring Recommendations: The tool delivered recommendations, such as modularizing tightly coupled code, splitting complex, low-cohesion files to improve code maintainability and extensibility.
- Enhanced Decision-Making: DETANGLE's visualizations, including network graphs and architecture dashboards, enabled the team to conduct root cause analysis and prioritize high-impact improvements.

DETANGLE's structured methodology, combining effort-based KPIs, architectural health metrics, and actionable recommendations, ensured that the Vaadin team could address legacy code challenges effectively. By focusing on technical debt hotspots, the tool supported incremental improvements in modularity and maintainability, aligning with the broader goals of SmartDelta.

Through its integration into the project, DETANGLE has demonstrated its ability to drive meaningful improvements in software quality, particularly in complex, legacy codebases. This positions it as a critical tool for managing technical debt in industrial software systems.

Reference:

[VD1] “Code quality tool & secure analysis with SonarQube,” Clean Code: Writing Clear, Readable, Understandable & Reliable Quality Code, https://www.sonarsource.com/products/sonarqube/

[VD2] “Technical debt - why the term causes more confusion than clarity and how to do it better!” https://capeofgoodcode.com/hubfs/Downloads/Technische%20Schulden/CoGC_Whitepaper_Tech_Debt_Analysis_with_DETANGLE%C2%AE.pdf

[VD3] “The Vaadin Flow Web Framework - On the Highway to a New Quality Level”, https://capeofgoodcode.com/en/knowledge/architecture-quality-trends-vaadin-flow-webframework

4.2.4. Automatic code analysis for historical code analysis and quality assessment (University of Innsbruck and cc.com)

Synopsis:

One of the widely used code analysis tools is SonarQube. However, SonarQube has some limitations in historical code analysis:

1. SonarQube and possible plugins update over time so that the quality measurement approaches could change. That means, for your analysis, if you update SonarQube or plugins, the comparability of your code artifacts suffers.

2. Consider whether you take over a system or have a long-term project that expects a SonarQube integration. How can you analyze the history of the related project? SonarQube focuses on the integration of "current" commits.

3. Maybe you only want to regard a specific time range of commits, analysis commits of (a) specific person(s) or a particular branch.

4. Additionally, we identified that it would be advantageous to evaluate the quality of other projects to establish a comparative benchmark. This approach allows for a more objective assessment by providing context and reference points, helping to identify relative strengths and areas for improvement in an evaluated project.

A tool named *SoHist* was developed to overcome these limitations and enhance functionalities for historical analysis. Consequently, points 1. – 3. were addressed in WP3, while point 4. in SoHist v2 was the primary focus of WP4, which is addressed further here.

Methodology:

Therefore, we conducted Exploratory Data Analysis (EDA). We followed the data analysis steps outlined by Tufféry [cc1] and adhered to the data analysis guidelines from the Empirical Standards for Software Engineering [cc2] to ensure rigorous practices. The overall process is displayed in the above figure.

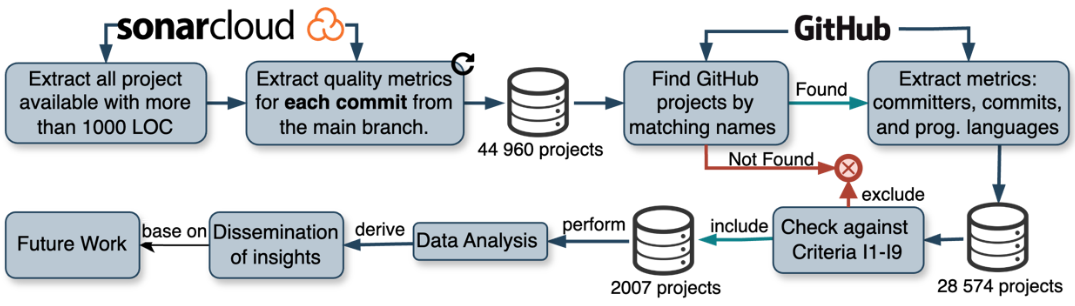

sonarcloud
GitHub
Extract all project available with more than 1000 LOC
Extract quality metrics for each commit from the main branch.
44 960 projects
Find GitHub projects by matching names
Found
Extract metrics: committers, commits, and prog. languages
Not Found
exclude
Check against Criteria I1-I9
28 574 projects
include
2007 projects
perform
Data Analysis
derive
Dissemination of insights
base on
Future Work

Figure 19: Code Analysis Methodology

For the data mining done in June 2024, we selected SonarCloud, due to its open structure and popularity. SonarCloud provides several metrics on code quality for individual commits on the main branch. In total, 44 960 projects with more than 1000 Lines of Code (LOC) were publicly available. Additionally, we utilized GitHub to obtain relevant information that was not available on SonarCloud. Consequently, only SonarCloud projects that were on GitHub, identified using the difflib6 library, were included in the analysis. All other projects were excluded.

Based on the data mining, we have selected 28 574 distinct projects, which consisted of a range of metrics. Table 13 describes selected metrics and provides descriptions relevant to the rest of the paper. In the table, some software quality metrics include a highlighted [R] to denote that we also use their densities, calculated by dividing the metric by the project's LOC. This adjustment allows for a more effective comparison of projects independent of their size. Otherwise, we have observed a significant impact of LOC on other metrics, consistent with findings from previous studies [cc3, cc4].

Table 13: Bug and Code Smell Descriptions

	Name	Description [C=Count,% =Percentage,B=Boolean]
Violations	Bug [R]	A concert coding mistake that can lead to an erroror unexpected behavior at runtime [C].
	Code Smell [R]	Refers to any issue that makes code confusing and hard maintain. They do not necessarily lead to errors [C].
	Vulnerability [R]	A weakness that can be exploited to compromise security [C].
Com	Cyclomatic Complexity	Counts independent paths through the code, indicating testing complexity [%].

	Cognitive Complexity	A qualification of how difficult it is to understand code (SonarQube developed metric)[%].
Test Cov.	Line Coverage	Indicates the percentage of lines of code that have been executed during testing [%].
	Branch Coverage	Measures the percentage of branches or decision points in the code that have been executed during testing.
	Coverage	SonarQube´s own test coverage [%].
Dup.	Duplicated Lines [R]	Number of lines of code that have an identical code line [C].
	Duplicated Blocks [R]	Number of blocks of code that have identical code lines [C].
Project	Lines of Code (LOC)	Count of lines of programming code in all files [C].
	Languages	Project´s programming languages and theirLOC [Array of Language with LOC].
	Committers	Total number of contributors via commits [C].
	Commits	Total count of individual changes made on a repository [C].
	Repository Stars	Reflects the popularity (user likes) by the community [C].
	Is not Forked	Outlines if have project emerged from another one [B].

Nevertheless, the 28 574 projects may present limitations concerning the overall validity of the assumptions made. Consequently, we established a series of Inclusion Criteria to filter and select projects based on defined relevance and quality standards, outlined in the following Table 14.

Table 14: Counts of Issues Across Different Software Projects

No.	Criteria and Description	#Proj.
I1	At least 1000 LOC.	44 960
I2	Publicly available on GitHub.	28 574
I3	Is not forked from another project.	22 104
I4	More than 100 GitHub commits.	10 893
I5	More than 4 GitHub committers.	7 154
I6	More than 10 GitHub repository stars.	2 844
I7	At least 10 complete SonarCloud analyses.	2 007

For the data analysis of research objectives, we utilized Jupyter Notebook along with Pandas for data manipulation, Seaborn and Matplotlib for visualization, and Scipy and Statsmodel for statistical tests and analyses. After aggregating and cleaning the data, we examined each metric, following Tufféry [cc1]. We analyzed the distribution of the data and tested for normality and skewness using Q-Q plots as well as D'Agostino and Pearson's normality test (univariant analysis). Our findings indicate that each quality metric is not normally distributed. Given this nonnormality, we employed non-parametric approaches. To address the second research objective (bivariant analysis), we selected Spearman rank correlation to calculate the correlation between two metrics, such as Cyclomatic Complexity

Density and Bug Density. By comparing these attributes and calculating a score within the range of -1 to +1, we aimed to gain insights into their relationship of different software quality metrics. This correlation approach remains robust even when dealing with skewed variables or extreme values.

Results:

Distribution Analysis of Software Quality Metrics

On the first chart of Fig. 20, we show the distribution of the three Test Coverage metrics. Each of these has a median coverage of at least 65%. If we consider 80% of the projects with the highest Line Coverage, we have at least 52.5%.

Regarding Complexity, we distinguish between Cyclomatic Complexity and Cognitive Complexity. In an initial attempt at chart visualization, considering only the latest analysis, we observed a peak at zero for Cognitive Complexity Density. This anomaly can be attributed to a known defect in SonarSource, as documented in the issue report7. Consequently, for projects containing JavaScript and TypeScript code, we used only the most recent analysis conducted before the release of SonarCloud version 10. Additionally, it can be observed that Cognitive Complexity Density is shifted to the left relative to Cyclic Complexity Density.

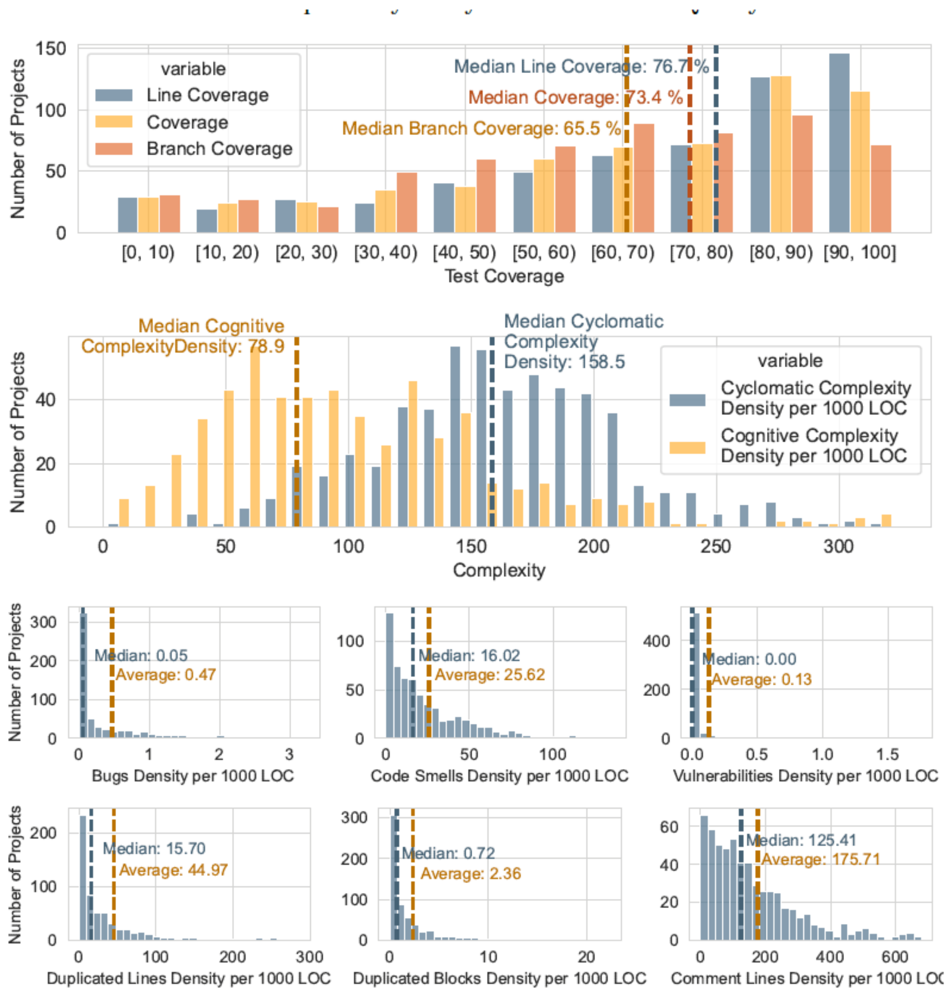


Figure 20: Distribution Analysis of Software Quality Metrics

We also examined the distribution of **Violations**: Bugs, Code Smells, and Vulnerabilities. For every 1000 LOC, Code Smells are the most prevalent, occurring at a rate of 16 per 1000. This could be explained by the fact that most Sonar Rules pertain to Code Smells, and the urgency of addressing them is relatively low. In contrast, Bugs and Vulnerabilities are less common. To be precise, 81% of projects report zero Vulnerabilities, and 45% of projects show no Bugs in their most recent analysis.

Next, we closely examined the **Duplication** densities at both the line and block levels. The distributions appear to be almost identical in shape. On average, a Duplicated Block contains 22 LOC.

Lastly, we looked at the **Comment Line** density. On average, the median shows that 120 lines of comments are used to document every 1000 LOC.

Objective 2: Correlation among Software Quality Metrics

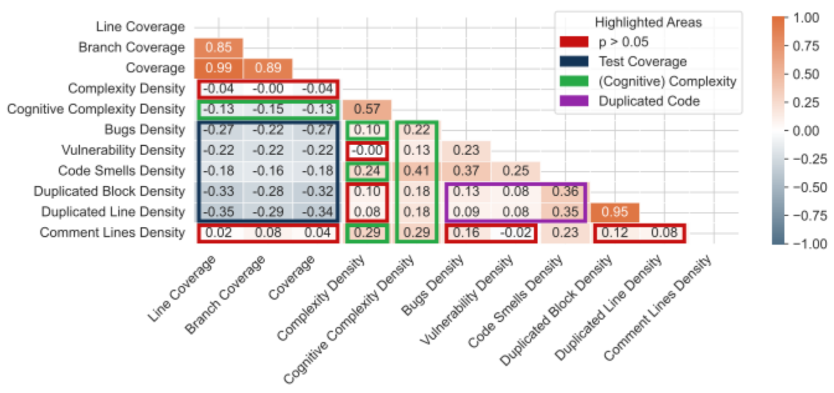


Figure 21: Correlation among Software Quality Metrics

Based on the results of Spearman's correlation analysis of the software quality metrics, specific p-values exceed the **$p > 0.05$** threshold and are highlighted with a red box in the correlation matrix of Fig. 21. Notably, Cyclomatic Complexity and Comment Lines Density frequently exhibit non-significant correlations. All other metrics have p-values below 0.05, permitting further exploration of their correlations using other annotation boxes.

First, for **Test Coverage**, we consider Line Coverage, Branch Coverage, and Coverage. The correlation between these coverage metrics and issue metrics (Bugs, Vulnerabilities, Code Smells Density) is negatively weak. Also, the coefficients for duplication densities per line and block fall within the range of $-0.19 < \rho < -0.40$, indicating a weak negative correlation.

Additionally, we observe that **(Cognitive) Complexity** has a weak positive correlation with metrics: Violations, Comment Lines, and Code Duplication. Additionally, there is a moderate positive correlation with Code Smells. This is obvious because a Sonar Rule flags high complexity as a Code Smell within the code. As already outlined, Cyclic Complexity often has $p > 0.05$; however, if not, it has a similar correlation behavior as Cognitive Complexity.

Similarly, within the green box, the **Duplicated Code** shows a nearly moderate correlation to code smells for the same reason. In contrast, Bugs and Vulnerabilities exhibit only a very weak correlation with Duplicated Code.

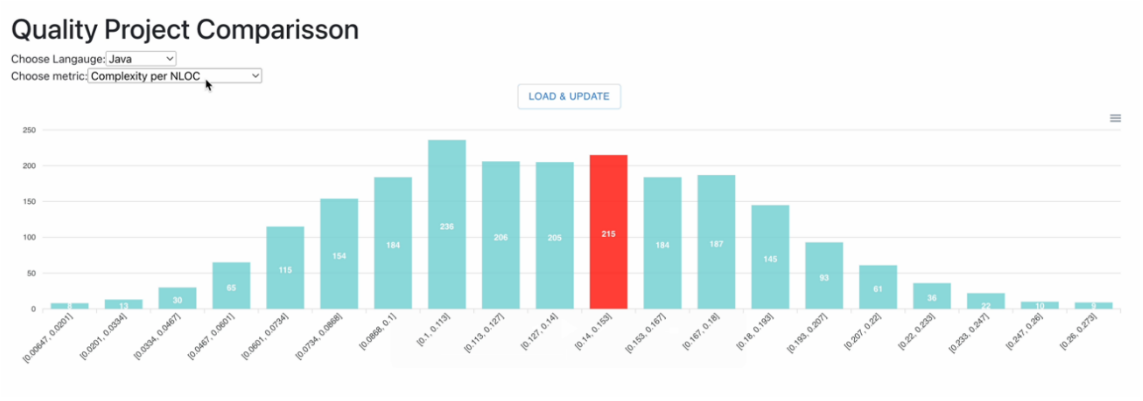
Considering the correlations within the **Categories of Metrics** shown in Fig. 3, we observe the following:

- **Test Coverage ↔ Coverage ↔ Line Coverage**: Strong
- **Duplicated Blocks Density ↔ Duplicated Lines Density:** Strong
- **Cyclomatic Complexity Density ↔ Cognitive Complexity Density**:
- Moderate

- **Bug Density** ↔ **Code Smells Density** ↔ **Vulnerability Density**: Weak

Summary and Usage for SoHist v2:

With this data, users can benchmark their project's quality metrics against a large set of comparable projects, considering various programming languages and selected quality criteria. A chart visualizes the distribution of these metrics across other projects, yielding insights into metrics like Test Coverage, Code Smell Density, and more. An example of Code Complexity is given in Fig. 22.


Quality Project Comparisson

Choose Langauge: Java

Choose metric: Complexity per NLOC

Figure 22: Usage for SoHist v2

References

[CC1] S. Tuffery, Data Mining and Statistics for Decision Making. Wiley Series in Computational Statistics, Chichester, West Sussex ; Hoboken, NJ: Wiley, 2011.

[CC2] P. Ralph, N. bin Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, B. B. N. de França, C. A. Furia, G. Gay, N. Gold, D. Graziotin, P. He, R. Hoda, and S. Vegas, "Empirical Standards for Software Engineering Research," 2020.

[CC3] Y. Gil and G. Lalouche, "On the correlation between size and metric validity," Empirical Software Engineering, vol. 22, pp. 2585–2611, Oct. 2017.10. M. A. A. Mamun, C. Berger, and J. Hansson, "Correlations of software code metrics:

[CC4] An empirical study," in Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, (Gothenburg Sweden), pp. 255–266, ACM, Oct. 2017.

[CC5] M. M. Mukaka, "Statistics corner: A guide to appropriate use of correlation coefficient in medical research," Malawi Medical Journal, vol. 24, pp. 69–71, Sept. 2012.

4.2.5. Analyze software quality trends based on issues and schedule the issues to find the balance between focussing on improving quality versus adding new features (FOKUS)

Synopsis:

This section describes the methods and tools for assessing the health of a software product under continuous development by looking manly at the upcoming issues and the reactions these provoke. Machine learning is used both for adding useful information to individual issues (e.g. classification, criticality, related code) as well as for time series analysis manly about changing frequencies eventually allowing to predict expected issue volumes for the future. Especially the development of the response time to issues is a good indicator for the overall health of a software system. The scheduling of issues should try to keep the response time in a reasonable corridor – i.e. postpend adding new features if fixing bugs tends to take longer and longer already.

Related works:

Our tool for unifying an enriching the issues is closely related to "CatIss: An Intelligent Tool for Categorizing Issues Reports using Transformers" [FOKUS1]. CatIss is a tool for categorizing GitHub issue reports using transformer-based models. By leveraging RoBERTa, a transformer known for its strong performance in NLP tasks, CatIss effectively automates the classification of issues into categories like bug, enhancement, and support. This work establishes a significant step forward in issue classification, as previous approaches relied primarily on traditional machine learning models, which often struggled to capture the contextual nuances in unstructured text. The model fine-tuning in CatIss adapts the transformer to domain-specific data, enhancing classification accuracy. The paper [FOKUS 1] provides a framework and model upon which our tool builds, with adaptations and extensions to improve functionality and applicability for specific repositories. Our work extends CatIss specifically by integrating an updated data processing pipeline, fine-tuning strategies, and configurable label management, allowing our tool to better serve the needs of ongoing software development projects.
For identifying and filtering out duplicate issues with machine learning there is a long research tradition. In [FOKUS 2] dating back to 2017 for instance, using Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) already resulted in high accuracy. More recently, for instance in [FOKUS 3] an approach for unsupervised learning is proposed.
Decision trees are used in [FOKUS 4] to assess the severity and priority of new bugs and the approach allows detecting and forecasting faults.

Methodology:

Our method for an intelligent and software “health” oriented issue management consists of three major steps:
1. unifying, enriching and filtering the issue data
2. analyze the issue trends
3. make assumptions about the expected soon incoming issues and schedule the actual new issues with the goal to keep the average response time within certain limits

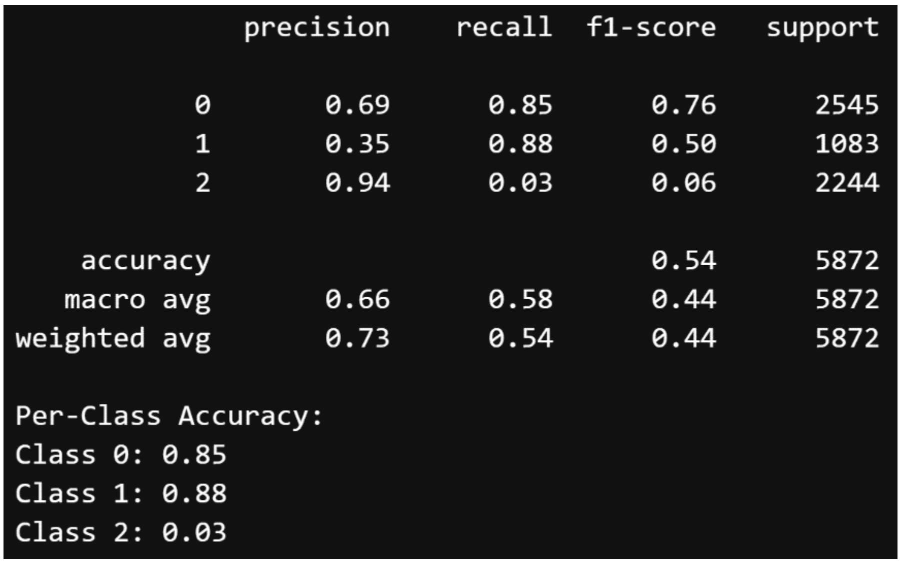
The first step is required since issues created by human beings will most likely not all have a comparable amount of amount, structure and quality of information. Even with predefined input forms and input assistant systems some fields will not be proper filled for many issues. Potential causes are that issuers do not have the required knowledge to provide the data they are asked for or that they are do not care, assuming that the textual description will be enough. Fortunately, with the help of machine learning and particularly with natural language processing, it becomes doable to automatically generate missing data. Within the SmartDelta project, such methods and tools for classifying different kinds of issues, for assessing their criticality and for identifying related code artefacts are developed. Identifying and filtering out potential duplicate issues is also crucial for any quantitative issue trend analysis. Applying methods and tools such es those developed in the SmartDelta project to get rid of duplicates concludes the first step.

For the actual issue trend analysis (step two), we focus on the available response times to already closed issues. There are at least five different time values between which the timespans are worth considering: The time when the issue is created, the time when the issue is assigned to someone who can eventually solve the issue, the time of the first response to the issuer, the time when the discussion of the issue in order to understand it ends and then the time when the issue is completed either by providing a solution or by concluding that no change is going to happen for that issue. Of course, not all issues have a discussion before solving them starts. However, if there is a discussion, then the timespan for that discussion is eventually not entirely a developer response time since it might include waiting for some clarification by the issuer to communicate what he really wants. Those waiting times must be subtracted. And dividing the discussion time by the number of clarification cycles yields the average discussion response time. The timespan for implementing a solution also needs to be corrected based on the amount of work required to make the solution. Dividing the implementation time by the number of lines of code altered already gives a simple correction, but more sophisticated approaches are applicable, too. The development of the average response timespan values for fine grained groups of issues – especially for issues of the same kind and criticality – reflects the overall software quality trend because it is exactly what the customers experience. In contrast to just looking at the development of the total number of open issues for instance, our method considers eventually changing capacities for managing the issues. If a software system massively grows and therefore more and more people are working on it, for instance, then an increasing number of open issues is expected and not necessarily a sign for a quality decline since there are also more developers dealing with the issues. The response timespans do reflect

altering capacities and capabilities – as long as the timespan lengths do not get out of control, it is fine for the customers.
Visualizing software quality trends graphically using the development of issue response timestamps might already be valuable, but there is more that can be done to support the strategic issue and more general software development management. In step three we try to predict how many issues of a certain kind and criticality are expected in the near future and how that will most likely affect the average lengths response timespans. The idea is to early recognize potential upcoming overloads before the customer support actually suffers from worse feedback and painfully slow fixes for their issues. If response times tend to increase too much or if higher issue frequencies are prognosed, the management is advised to consider focussing on improving the quality of the software with the already implemented features instead of increasing the complexity by adding additional stuff.

Results:

For unifying and enriching issues, we developed an issue classification tool using RoBERTa, a transformer known for its strong performance in NLP tasks. The tool provides a sophisticated data processing pipeline, fine-tuning strategies, and configurable label management. Here are some results for the two GitHub repositories vaadin/flow and grafana/grafana:



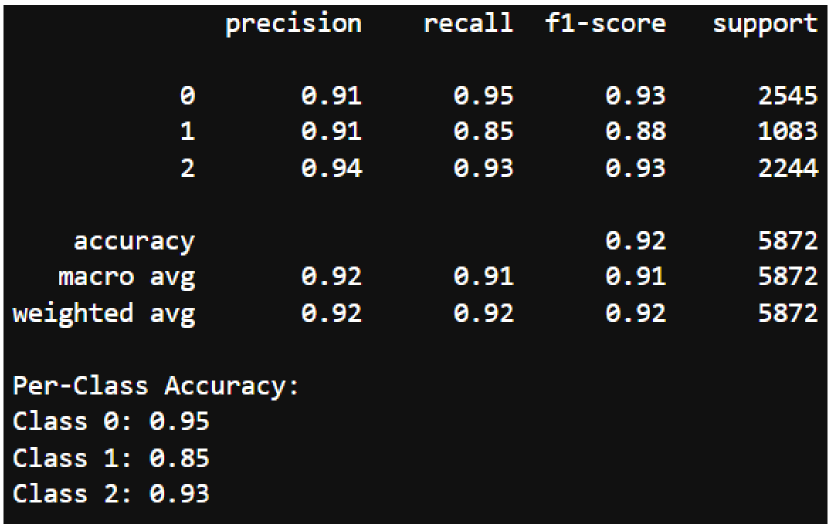
```
              precision    recall  f1-score   support

           0       0.69      0.85      0.76      2545
           1       0.35      0.88      0.50      1083
           2       0.94      0.03      0.06      2244

    accuracy                           0.54      5872
   macro avg       0.66      0.58      0.44      5872
weighted avg       0.73      0.54      0.44      5872

Per-Class Accuracy:
Class 0: 0.85
Class 1: 0.88
Class 2: 0.03
```

Figure 23: Results for grafana/grafana without fine-tuning



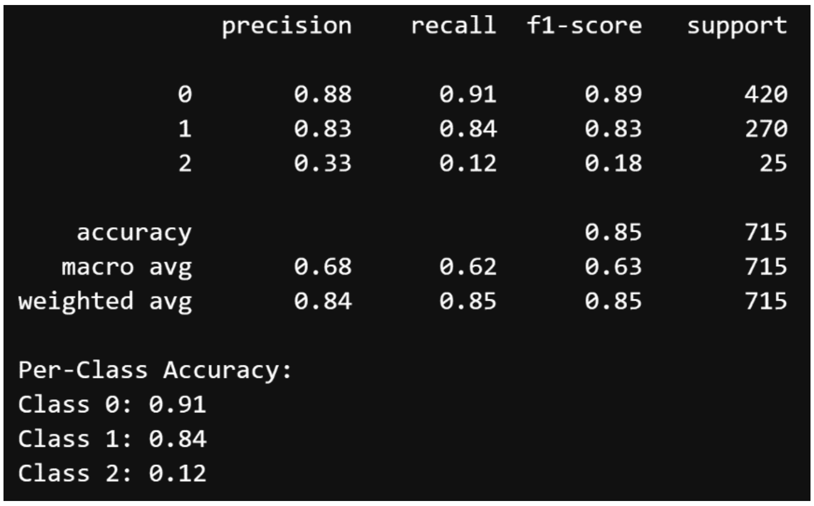
```
              precision    recall  f1-score   support

           0       0.91      0.95      0.93      2545
           1       0.91      0.85      0.88      1083
           2       0.94      0.93      0.93      2244

    accuracy                           0.92      5872
   macro avg       0.92      0.91      0.91      5872
weighted avg       0.92      0.92      0.92      5872

Per-Class Accuracy:
Class 0: 0.95
Class 1: 0.85
Class 2: 0.93
```

Figure 24: Results for grafana/grafana with fine-tuning



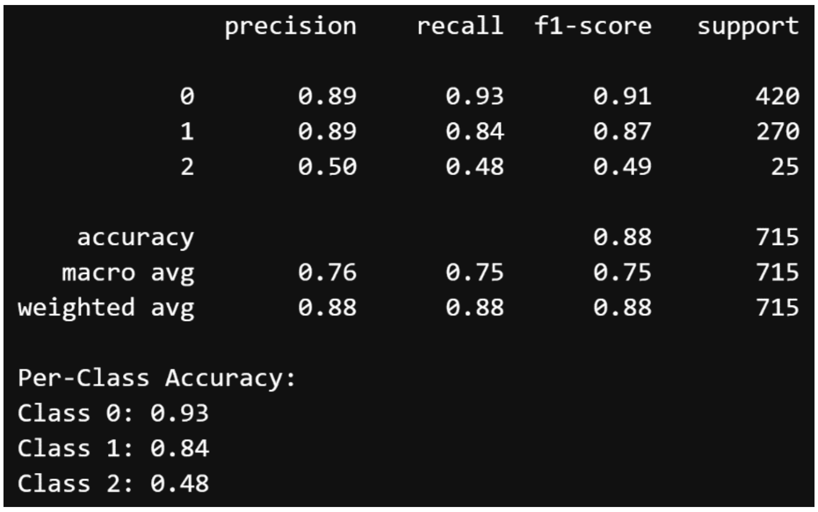
```
              precision    recall  f1-score   support

           0       0.88      0.91      0.89       420
           1       0.83      0.84      0.83       270
           2       0.33      0.12      0.18        25

    accuracy                           0.85       715
   macro avg       0.68      0.62      0.63       715
weighted avg       0.84      0.85      0.85       715

Per-Class Accuracy:
Class 0: 0.91
Class 1: 0.84
Class 2: 0.12
```

Figure 25: 3 Results for vaadin/flow without fine-tuning



```
              precision    recall  f1-score   support

           0       0.89      0.93      0.91       420
           1       0.89      0.84      0.87       270
           2       0.50      0.48      0.49        25

    accuracy                           0.88       715
   macro avg       0.76      0.75      0.75       715
weighted avg       0.88      0.88      0.88       715

Per-Class Accuracy:
Class 0: 0.93
Class 1: 0.84
Class 2: 0.48
```

Figure 26: Results for vaadin/flow with fine-tuning

Our tool for visualizing the development of the average response times (step two) takes advantage of the enriched issue data by using it for focussing on comparable issues.

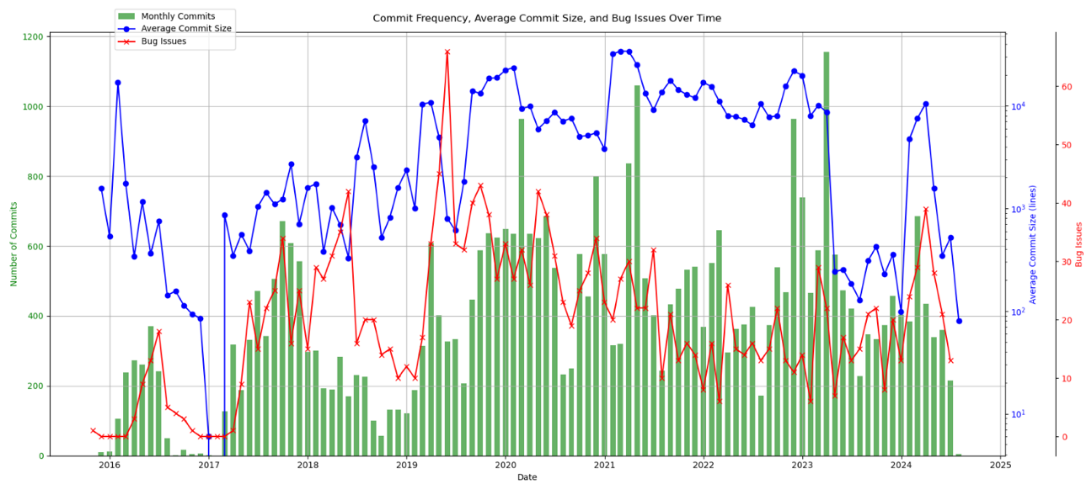


Figure 27: Commit Frequency, Average Commit Size and Bug Issue Count Over Time for the vaadin/flow repository with a log scale for average commit size

Time series prediction for the expected upcoming issues is notoriously difficult. First of all, it requires a large number of issues over a long period of time for learning. Additionally, for good results it will probably be necessary to take changes of the code base and in the developer community into account. We are still trying to figure out how to make accurate forecasts and the work will be continued beyond the SmartDelta project. Nonetheless, our response time analysis of only the real existing issues can already give sound guidance for prioritizing bug fixes over feature requests and thereby help to improve the scheduling of issues.

Summary:

With our approach, just by carefully analysing the responses to issues it is possible to show software quality trends in a continuous development process. Furthermore, it is possible to guide the issue scheduling so that the average response times will stay within acceptable limits by recommending when it is appropriate to add new features and when it is better to focus on improving the quality of the already implemented stuff.

Reference:

[FOKUS1] M. Izadi, “CatIss: an intelligent tool for categorizing issues reports using transformers,” in Proceedings of the 1st International Workshop on Natural Language-based Software Engineering, Pittsburgh Pennsylvania: ACM, May 2022, pp. 44–47. doi: 10.1145/3528588.3528662.
[FOKUS2] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta and N. Dubash, "Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques,"

2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 2017, pp. 115-124, doi: 10.1109/ICSME.2017.69

[FOKUS3] H. Wang, Z. Gao, C. Wang and G. Sun, "Unsupervised Learning Exploration for Duplicate Bug Report Detection," *2023 IEEE 11th International Conference on Information, Communication and Networks (ICICN)*, Xi'an, China, 2023, pp. 851-857, doi: 10.1109/ICICN59530.2023.10392320

[FOKUS4] Tran, H.M., Le, S.T., Nguyen, S.V. *et al.* An Analysis of Software Bug Reports Using Machine Learning Techniques. *SN COMPUT. SCI.* **1**, 4 (2020). https://doi.org/10.1007/s42979-019-0004-1

4.3. ML-Based Similarity Analysis Approaches and Recommendations

4.3.1. Similarity analysis of State Machines using hierarchical modularization (TWT, Akkodis)

Synopsis:

A State Machine (SM) is a behaviour model of a system. It consists of a finite number of states and transitions and is also called Finite-State Machine (FSM). Starting from specific state and a given input, the machine performs transitions resulting in outputs [TWT1]. In our case, we are considering state machines that are following defined ISO standards (like the ISO 15118-20).

A comparison between states machines is necessary, for example to track the evolution over time of specific requirements of norms or to recognise possible reuse opportunities. Manually analysing State Machines is a time-consuming task, especially comparing one State Machine against numerous others is hence hardly possible. GSR as a tool streamlines this process, particularly when the State Machines adhere to specific logic, such as transitions defined by ISO standards. By leveraging background knowledge, our tool saves time and enhances the quality of the analysis. This makes it possible to compare a state machine to numerous other State Machines from a large database to find and recommend similar or comparable State machines.

Related works:

The idea of model checking as an automatic verification technique has already been around since the early 80’s with Clarke’s and Emerson’s work [TWT 2]. Model checking serves as a powerful method for evaluating a finite state system's description in relation to its formal specification, systematically identifying potential errors [TWT 3].

There are different methods to compare two models. One that gained a lot of traction due to its high flexibility is the Graph Edit Distance (GED) method. GED is a method that measures the similarity between two graphs based on the amount of distortion required to convert a graph into another one. This method enables a selection process for the cost model within a specific application of graph edit distance. Additionally, the precise calculation of graph edit distance may utilize various algorithms, such as a tree search algorithm [TWT 4].

Methodology:

Given a state machine as input, the tool searches for the most similar State machine in each database. It is assumed that the states of all state machines are the parts of a fixed ISO norm. The methodology consists of the following steps:

1. **Hierarchical labelling of the state space:** The parts of the ISO norm are labelled hierarchically based on their contents. This grouping of states is also called modularization. Different layers can be defined here for a more precise partitioning of the State Machine. This step must be done only once.

2. **Comparison of State Machines:** Iteratively, the tool compares the input state machine to every state machine in the database:
    a. **Hierarchical Modularization:** Based on the labelling of the ISO norm, the state machines are decomposed in hierarchical modules. Through this step, a comparison on module level is possible. Such a comparison enables a more industry driven similarity analysis for the whole State Machine, as changes on specific parts can be focused on.

    b. **Identification of matching modules:** Based on the hierarchical modularization, the corresponding modules of the two state machines are matched throughout the layers.

    c. **Comparison of modules:** Starting with the modules in the lowest layer, the modules of the state machines are compared using the graph edit distance and the states of the modules are mapped. After a module has been compared, all the states and transitions that are part of it are collapsed into one state. This process is repeated till all the modules have been compared.

    d. **Determination of the Similarity:** Recursively, the similarity values of all modules are determined via graph edit distance. The similarity values depend on the type of deviations between the state machines. 6 types are covered: Addition/Removal of an edge, Addition/Removal of a state, Relabelling of a state or edge. Each type has a different weight of influence on the similarity analysis that can be customized.

    e. **Determination of the Deltas:** Using the state mappings between the corresponding modules, the nodes of the state machines are mapped as well as the transitions. This results in a mapping, also called “delta paths”, describing the differences between the two state machines.

    f. **Data storage:** The output of the analysis, also called the delta paths, is then stored in a JSON format. The output contains the definition of the State

Machines, the modularization, the similarity values for each module and the delta paths.

Results:

When analysing such tools in an industry context there are two main parameters that we focus on: process speed-up and accuracy.

Speed-up:

Time saving is a major focus point for industries. Manually comparing two state machines is a time-consuming process that can take between minutes and hours, depending on the complexity of the state machines. The required time rises exponentially with the complexity.

The GRS tool is able to compare states machines in the range of seconds depending on their complexity.

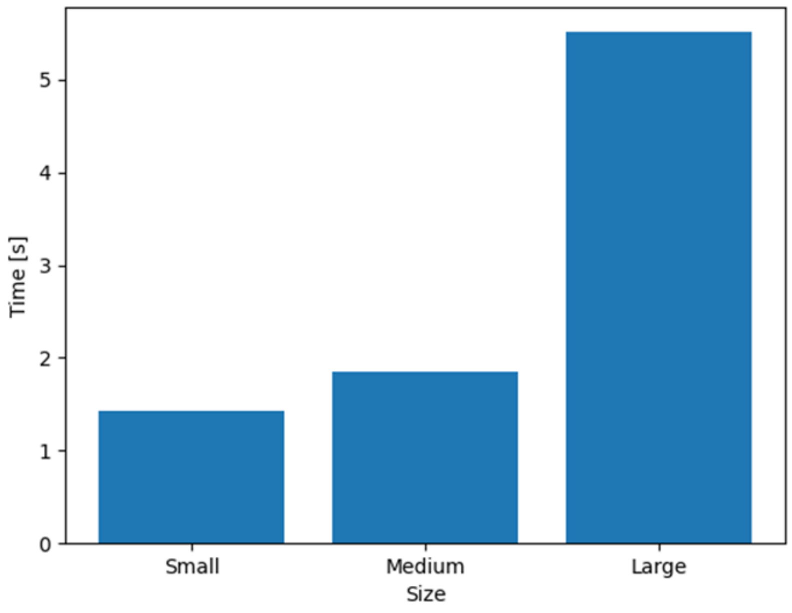


Figure 28: Evolution of comparison time depending on the state machine size

As shown in the graph above, a comparison between “small” (less than 10 states) state machines take 1.43 seconds. For “medium” (less than 20 states) state machines, the comparison takes 1.85 seconds. For “large” (above 20 states) state machines, the time goes up to 5.51 seconds. This shows a significant speed-up compared to a manual comparison.

With this, a comparison on a whole database can be performed in a few hours instead of multiple days if done manually, depending on the size of the database.

Accuracy:

To measure the capacity of the tool to find all the differences between two State Machines, we performed a test in which we tracked the reliability of the generated delta paths during the analysis.

For this we took a base state machine, that we will call “base_example”, and randomly generated 5 sets of 10.000 variations of the state machine. The number and type of modifications is randomly chosen for each variation. For the number of modifications, it is defined between 0 and 3 and for the types it chooses between all the 6 types of modifications mentioned in the methodology section above. The “base_example” taken here is a state machine containing 18 states and 53 transitions and can be classified as having a medium complexity.

In the first step the 10.000 variations are compared with the “base_example” and the delta paths are computed. For every variation, the computed delta path is applied to the “base_example” and it is checked if the resulting state machine is identical to the tested variation.

The below shows the success rate of going from one state machine to the other based on the computed delta paths.

Table 15: Success rate of finding the correct delta paths

Set	Set 1	Set 2	Set 3	Set 4	Set 5
Success rate [%]	91,04	91,46	91,22	91,20	91,31

In each set a success rate of above 90% was achieved.

Summary:

The GSR tool enables a fast and efficient comparison of a state machine against numerous other State Machines from a large database, a task that would otherwise be impractical and highly labour-intensive when done manually. Additionally, it allows for focused comparisons on specific regions or functionalities. This is done by partitioning the State machines into modules based on the clustering and comparing those modules via graph edit distance. Based on the module similarities, a similarity of the state machines is computed. Moreover, deltas between the State machines are determined and returned on demand.

Initial tests indicate that the differences between the state machines are accurately detected and changes in state machines are allocated in correct regions of the state machines. However, the detailed evaluation is still pending.

References:

[TWT1] 27 March 2024 at the Wayback Machine, Wright, D. R. (2005), Finite State Machines, NC State University, https://web.archive.org/web/20140327131120/http:/www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf

[TWT2] Clarke, Edmund M., and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic." *Workshop on logic of programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981.

[TWT3] Clarke, Edmund & Grumberg, Orna & Peled, Doron. (2001). Model Checking.

[TWT4] Riesen, K. (2015). Graph Edit Distance. In: Structural Pattern Recognition with Graph Edit Distance. Advances in Computer Vision and Pattern Recognition. Springer, Cham. https://doi.org/10.1007/978-3-319-27252-8_2

4.3.2. Graph based similarity analysis and recommendations (TWT, Software AG, Vaadin)

Synopsis:

The rapid digitalization has led to increasingly complex software systems. Managing this complexity poses significant challenges for developers, particularly in tasks such as maintenance, code reuse across different projects, and adherence to regulatory requirements. To address these challenges and accelerate software development cycles, a novel tool called Code Similarity Investigator (CSI) has been developed. CSI utilizes graph-based code similarity analysis to automate code reuse suggestions, streamline API replacements, enhance code refactoring processes, and prioritize test cases, thereby reducing the time developers spend on repetitive and mundane tasks.

Related works:

Efficiently handling large and complex codebases has been a longstanding challenge in software engineering. Traditional methods for detecting code similarities often rely on clone detection techniques, which compare code based on syntactic patterns. Tools like CCFinder [AG1] and JPlag [AG2] have been used for plagiarism detection and clone analysis by identifying exact or near-exact code duplicates. However, these methods are typically limited to specific programming languages and may not effectively capture semantic similarities.
Recent research has shifted towards semantic code analysis using graph-based representations. Code Property Graphs (CPGs) have emerged as a powerful tool, combining Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Program Dependency Graphs (PDGs) into a unified model. This rich representation enables more nuanced analysis of code semantics. Studies like Suneja et al. [AG3] have leveraged CPGs for vulnerability detection by identifying patterns in code graphs.
Machine learning approaches, including Graph Neural Networks (GNNs), have also been explored to learn embeddings of code graphs for similarity detection. Works such as DeepSim [AG4] encode control and data flow into high-dimensional feature vectors to measure functional code similarity. However, these methods often face challenges related to computational complexity and the availability of large, annotated datasets for training.

Methodology:

CSI focuses on leveraging graph-based models to identify code similarities within large codebases. The methodology comprises several key steps:

1. Code Representation Using Code Property Graphs (CPGs): Source code is transformed into CPGs using tools like Joern. CPGs integrate syntactic and semantic information by combining ASTs, CFGs, and PDGs, providing a comprehensive representation of the code structure and behaviour.
2. Subgraph Extraction: Relevant subgraphs are extracted from the full CPGs to focus on specific code sections, such as methods or classes. Selecting appropriate subgraphs is crucial to balance detail and computational efficiency. Subgraphs that are too large may introduce unnecessary complexity and hinder performance, while overly small subgraphs may lack sufficient context.
3. Graph Similarity Measurement with Graph Edit Distance (GED): The core of the similarity analysis relies on computing the GED between code subgraphs. GED measures the minimal number of edit operations required to transform one graph into another, where edit operations include adding, deleting, or substituting nodes and edges [AG5]. By quantifying these differences, GED provides a customizable way to assess the similarity between code sections through application specific costs for each operation.
4. Due to the NP-hard nature of exact GED computation, approximate algorithms are employed to make the process tractable for large graphs. These approximations aim to balance accuracy and computational efficiency, allowing the methodology to scale to real-world codebases.
5. Similarity Classification: Based on close collaboration with Software AG and Vaadin, the Code Similarity Investigator (CSI) labels the similarity into four classes:
   - None: No similarity detected.
   - Low: Minor similarities that may not warrant action.
   - Medium: Moderate similarities suggesting potential for code reuse or refactoring.
   - High: Significant similarities indicating strong candidates for code reuse, refactoring, or applying similar fixes.

Results:

Preliminary tests of the Code Similarity Investigator (CSI) were conducted using data provided by Vaadin, based on their open-source web application development platform. A total of 29 code pairs were evaluated to compare the automated similarity assessments generated by CSI with manual assessments conducted by developers at Vaadin.
The similarity levels were categorized into four classes: None, Low, Medium, and High. Each code pair received an assessment from both the automated tool and the manual evaluation.

To quantify the agreement between the two assessments, we calculated the difference between the two similarity assessments based on the following mapping: None = 1, Low = 2, Medium = 3, High = 4. This indicates how closely the tool matches human judgment.

The results are summarized as follows:

- Exact Match (Difference of 0): 18 out of 29 code pairs (62%)
- Close Agreement (Difference of 1): 9 out of 29 code pairs (31%)
- Significant Disagreement (Difference of 2): 2 out of 29 code pairs (7%)
- Completely Off (Difference of 3): 0 out of 29 code pairs (0%)

These preliminary results demonstrate that CSI aligns closely with expert evaluations, with 93% of cases showing exact matches or close agreements. CSI can furthermore be finetuned for a certain use case, which was not done for these preliminary tests and can further improve its performance.

Summary:

The Code Similarity Investigator (CSI) offers a promising solution to the challenges posed by complex and evolving software systems. By modelling code as CPGs and utilizing GED for similarity measurement, CSI automates tedious tasks, ensures consistency across projects, and reduces development time. Future work will focus on extensive validation across larger and more diverse codebases, optimization of the GED approximation algorithms, and exploration of machine learning techniques to enhance the accuracy and scalability of the methodology.

References:

[AG1] Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7), 654–670.
[[AG2] Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science, 8(11), 1016.
[AG3] Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J., & Morari, A. (2020). Learning to map source code to software vulnerability using code-as-a-graph. arXiv preprint arXiv:2006.08614.
[AG4] Zhao, G., & Huang, J. (2018). DeepSim: Deep learning code functional similarity. In Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 141–151). ACM.
[AG5] Fischer, A., Riesen, K., & Bunke, H. (2017). Improved quadratic time approximation of graph edit distance by combining Hausdorff matching and greedy assignment. Pattern Recognition Letters, 87, 55–62.

4.3.3. ML-based methods to identify requirements from large data repository and generate recommendations (RISE, Alstom)

Synopsis:

Requirements are often mixed with supporting information, contractual obligations, and other customer-supplied documents. This makes it harder to analyze and understand customers' wishes and perceptions of the end system, especially in the process of responding to a call for tender in a project-based industry. Therefore, it is essential and a prerequisite to software development to extract technical specifications from customer-supplied documents such as tender documents. The identified technical specifications can be used as a base for other downstream activities, such as feasibility analysis, requirements quality assurance, and requirements allocation for development and verification. Manual requirements extraction from large documents is a resource-intensive and experience-dependent process and is subject to human fatigue [RISE1]. In addition, it is not scalable and could add additional delays to the project procurement process in the already very competitive bidding processes.

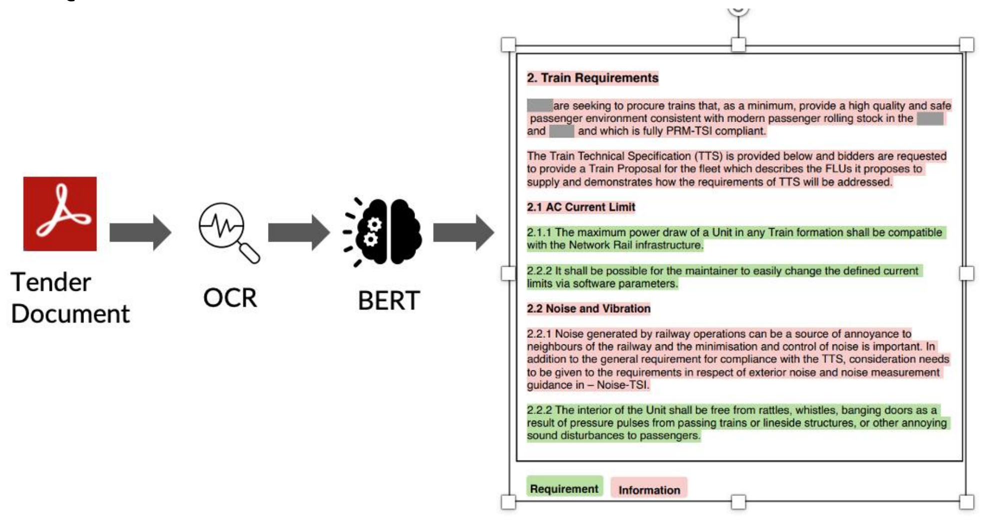
Related works:

Literature has been focusing on distinguishing requirements from other information using machine learning. Like our use case, Winkler *et al.* [RISE2] propose a deep learning (DL) classifier based on Convolution Neural Networks (CNNs) to identify requirements from additional material stored in IBM DOORS. Falkner *et al.* [RISE3] propose a Naive Bayes (NB) classifier---trained on unique words---to identify requirements from Request of Proposal (RFP) documents within the railway safety domain. Furthermore, Abualhaija *et al.* [RISE4] proposes an automated ML-based approach to demarcate requirements in textual specifications by considering one sentence as a unit of classification. They empirically evaluate ML classifiers on the industrial dataset consisting of 12 documents. In addition, Sainani *et al.* [RISE5] defines a two-step methodology to first extract requirements from 20 Software Engineering (SE) contracts and then allocate them to their specific types. For identification and extraction of requirements, Bi-LSTM yields the best results compared to ML algorithms. To allocate identified requirements in sub-classes, BERT (Bi-directional Encoder Representations from Transformers) performed better in terms of F-1 score.

Methodology:

We proposed two approaches, requirements identifier (REQ-I) [RISE 6] and requirements allocator (REQA) [RISE7] to support the project bidding and feasibility phase for one use-case provider in SmartDelta. The REQ-I approach enable faster and more automated requirements extraction and identification from tender documents for later analysis, feasibility and implementation. On the other hand, the REQA approach is meant to smartly recommend various teams for the identified requirements, for implementation and verification. We briefly introduce the approaches, their evaluation setup and the obtained results from the evaluation, as follows.

1) REQ-I



2. Train Requirements

[redacted] are seeking to procure trains that, as a minimum, provide a high quality and safe passenger environment consistent with modern passenger rolling stock in the [redacted] and [redacted] and which is fully PRM-TSI compliant.

The Train Technical Specification (TTS) is provided below and bidders are requested to provide a Train Proposal for the fleet which describes the FLUs it proposes to supply and demonstrates how the requirements of TTS will be addressed.

2.1 AC Current Limit

2.1.1 The maximum power draw of a Unit in any Train formation shall be compatible with the Network Rail infrastructure.

2.2.2 It shall be possible for the maintainer to easily change the defined current limits via software parameters.

2.2 Noise and Vibration

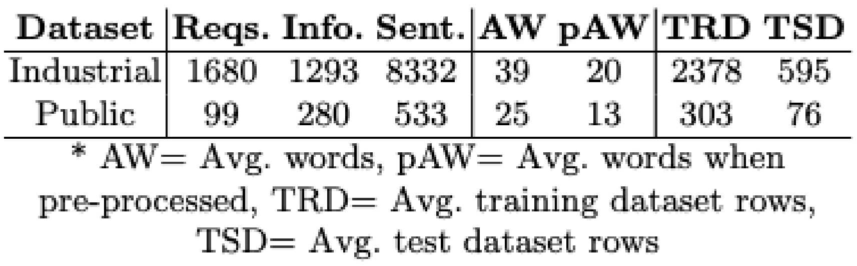
2.2.1 Noise generated by railway operations can be a source of annoyance to neighbours of the railway and the minimisation and control of noise is important. In addition to the general requirement for compliance with the TTS, consideration needs to be given to the requirements in respect of exterior noise and noise measurement guidance in – Noise-TSI.

2.2.2 The interior of the Unit shall be free from rattles, whistles, banging doors as a result of pressure pulses from passing trains or lineside structures, or other annoying sound disturbances to passengers.

🟩 Requirement 🟥 Information

Figure 29: REQ-I Approach: Approach for requirements extraction and identification

The REQ-I approach takes new tender documents as input and produces a PDF file with highlighted requirements. The approach first parses the text of the PDF files using optical character recognition (OCR) and then applies a fine-tuned version of the BERT model to classify the text into requirements or non-requirements. Below, we detail the evaluation setup used to evaluate the approach and obtain the results. We focused the evaluation on the effectiveness of the requirements identification process. To achieve this, we considered five already annotated tender documents from Alstom. These five documents were annotated by experts, and requirements among the documents were identified. In addition, to allow replication, we also considered a public dataset.



Dataset	Reqs.	Info.	Sent.	AW	pAW	TRD	TSD
Industrial	1680	1293	8332	39	20	2378	595
Public	99	280	533	25	13	303	76

* AW= Avg. words, pAW= Avg. words when pre-processed, TRD= Avg. training dataset rows, TSD= Avg. test dataset rows

Figure 30: REQ-I Data: Considered data from REQ-I evaluation

As shown in Figure REQ-I Data, in the industrial data, around 1680 requirements were identified by experts, while the rest of the 1293 text chunks were considered to be additional supporting information. We use five-fold validation to avoid model overfitting and enable generalizability of the results. On average 2378 textual chunk were considered across the five folds for training various classifiers for requirements identification.

Considered classifiers included traditional classifiers, deep language models, and few-shot classifiers. For traditional classifier, we feed term-frequency inverse document frequency (TF-IDF) based vectors to the classifiers Support Vector Machines (SVM), Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), and Naïve Bayes (NB). For a fair comparison and tuning, we applied random multi-search optimization to select the optimal hyperparameters. SVM and LR achieved better results on evaluation metrics when trained with normalized and reduced TF-IDF vectors using PCA. However, the rest of the ML pipelines---RF, DT, and NB---performed better with normalized TF-IDF vectors without PCA-based dimensionality reduction. In addition, we also consider a baseline random pipeline (W. Rand.) that classifies input as a requirement or not based on their frequency distribution in the dataset.

For deep language model-based classifiers, we considered the seminal GLoVE and FastText based embedding for the LSTM classifier. We considered the REQ-I approach based on BERT uncased model and few other variants of the approach SciBERT, RoBERTa, XLMRoBERTa (XRBERT), DistilBERT (DisBERT), and XLNet.

Finally, for few-shot classifiers, we considered MiniLM and S-BERT-based classifiers with only 10% and 20% of the data to evaluate their performance of "few" shot classification.

As typical in the NLP domain, pre-processing of the input text might impact classification performance. Therefore, we also consider the datasets both with (pipeline with names starting with "p") and without pre-processing.

We use the standard evaluation metrics for text classification, as follows:

- Accuracy (A) is the ratio of the number of correct predictions and the total predictions.
- Precision (Prec. Or P) is the ratio of correct positive predictions and the total number of positive predictions.
- Recall (Rec. Or R) quantifies the number of correct positive predictions from all possible positive predictions.
- F1 score (F1) is the harmonic mean of precision and recall.

We report the macro and weighted average across the fold for all our evaluation metrics in the results section.

2) REQA

After requirements are identified and agreed upon, it is essential to allocate those requirements to the right teams for implementation and verification. In this regard, we also proposed the REQA approach for smart requirements allocation to teams. The approach uses both traditional and state-of-the-art machine learning approaches to achieve this in an explainable manner. The REQA approach is composed of two main modules, Assigner and Augmenter, as shown in the following Figure. The Assigner module is responsible for generating a representation for the input requirement and for suggesting a possible allocation based on the results of a classification algorithm. Given a requirement, the Assigner outputs a list of potential allocation classes, ranked by likelihood.

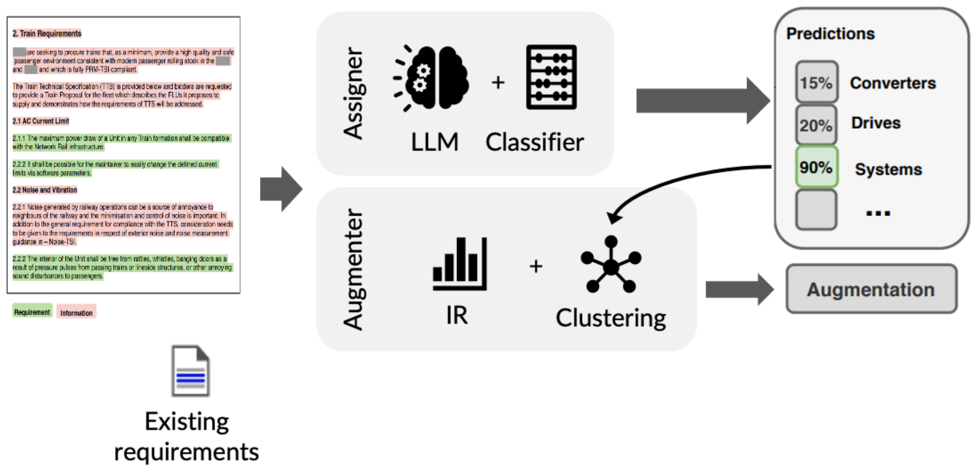


Figure 31: REQA Approach: Approach for requirements allocation

Only the most probable class is shown to the user, while the ranked list will be used by the Augmenter. The Augmenter module produces additional information to complement the predictions generated by the Assigner. This additional information helps in providing the most likely classes derived from lexical similarity-based measurements. The Augmenter searches for the most similar requirements in the training set used to train the Assigner. Then, it checks whether the classes produced by the Assigner match the classes with the most similar requirements identified based on lexical features. This can be regarded as a complementary channel to better inform the requirements analyst in deciding the allocation. Below we detail the evaluation methodology of the REQA approach.

The REQA tool for requirements allocation to teams was evaluated on 1680 requirements that were already allocated to various teams at Alstom. As shown in Figure REQA Data, the requirements were allocated to 15 different teams at the company responsible for developing various sub-systems. We use five-fold validation to avoid model overfitting and enable generalizability of the results. On average 1344 requirements were considered across the five folds for training various classifiers for requirements allocation.

Considered classifiers for comparison included traditional classifiers and classifiers based on deep language models. For traditional classifiers, we feed term-frequency inverse document frequency (TF-IDF) based vectors to the classifiers like the setup for REQ-I but instead of Naïve Bayes we use the multi-class version (MNB). In addition, we also consider a baseline random pipeline (W. Rand.) that classifies input as a requirement or not based on their frequency distribution in the dataset.

For deep language model-based classifiers, we considered the seminal FastText based embedding for the LSTM classifier. We considered the REQA approach based on SciBERT model and few other variants of the approach BERT base, and RoBERTA.

We use the same evaluation metrics as of REQ-I.

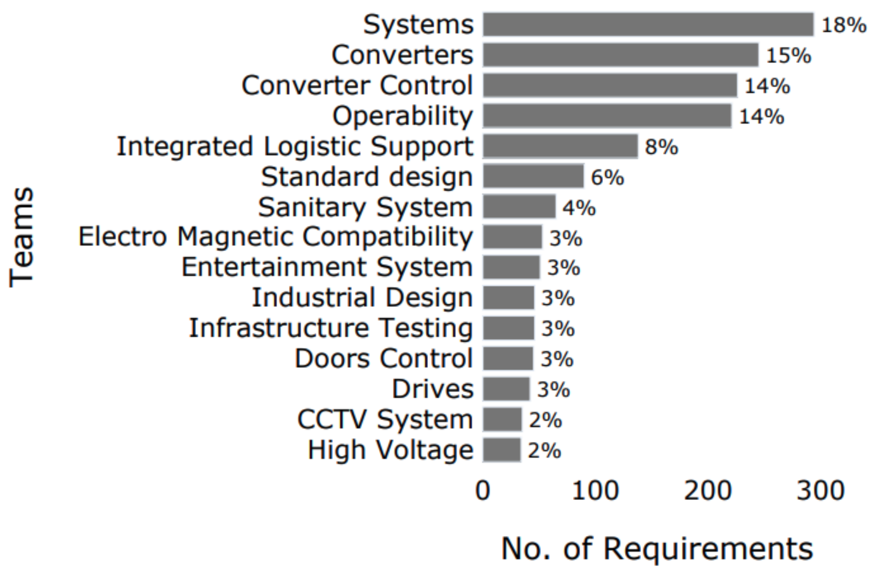
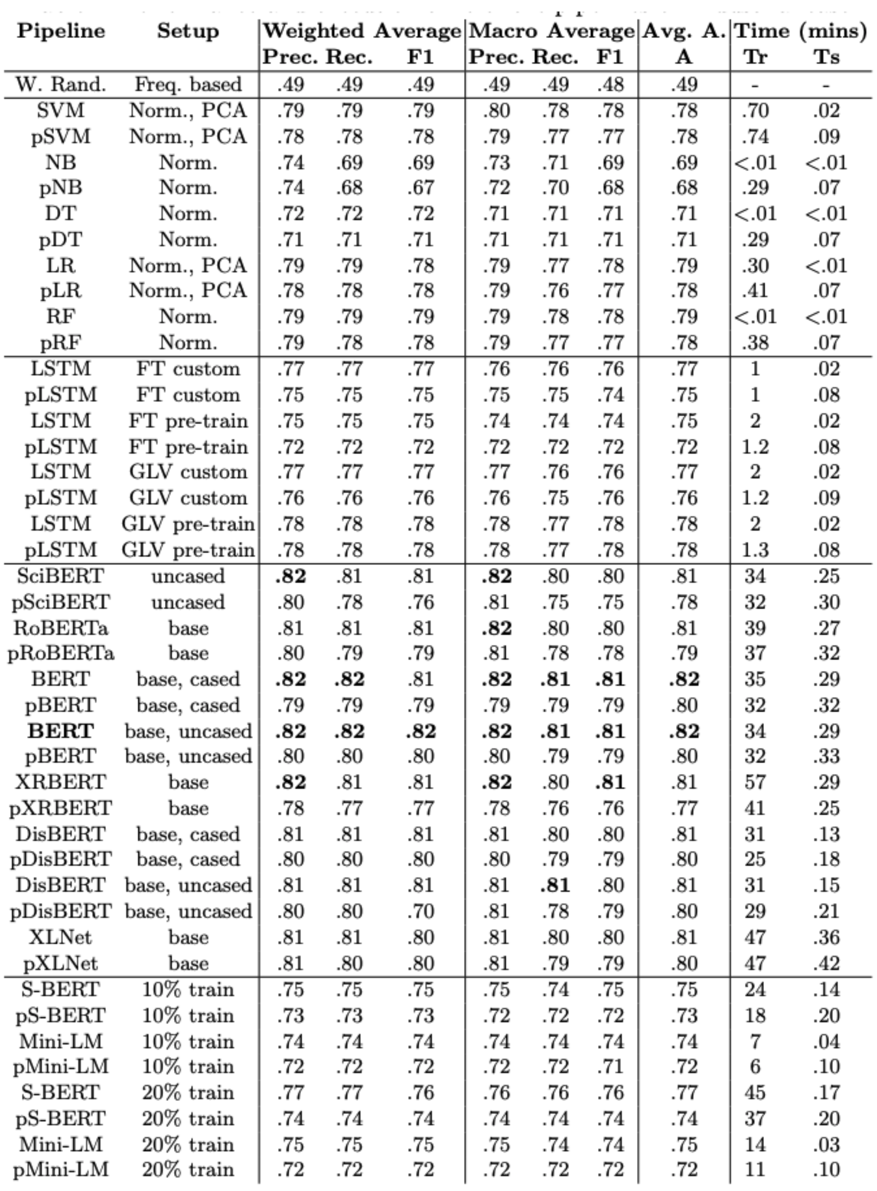


Figure 32: REQA Data: Considered data for REQA evaluation

Results:

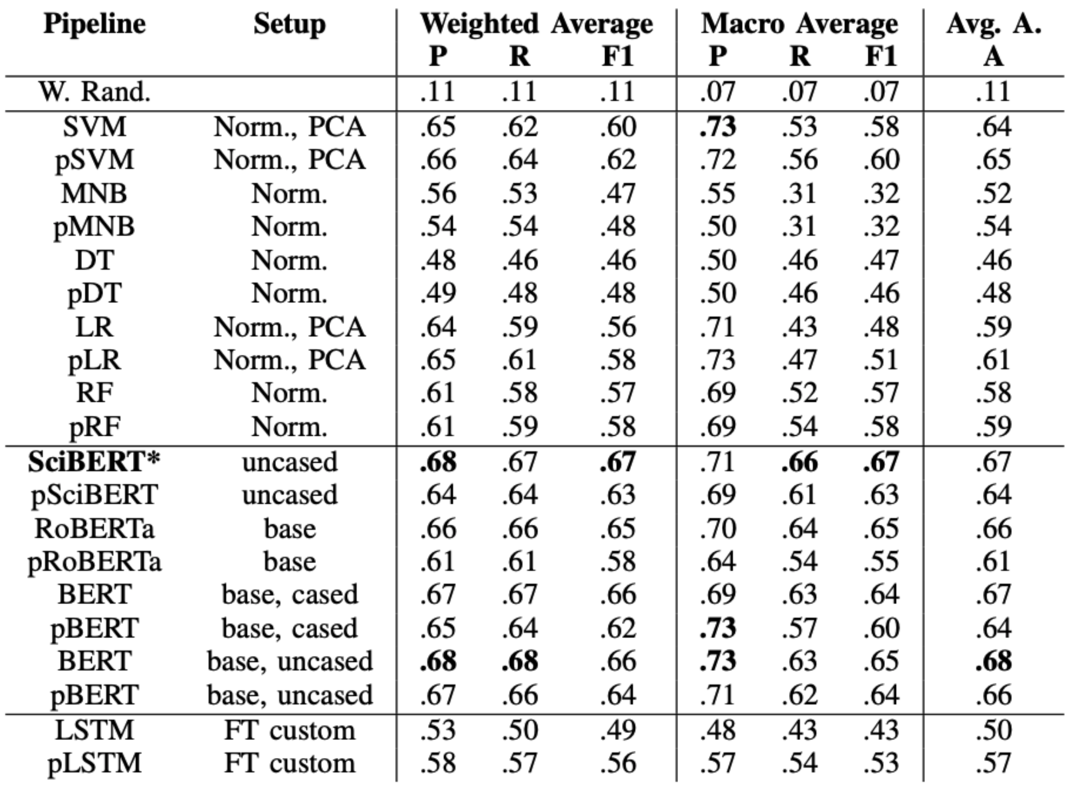
1) REQ-I: The requirement identification part of the toolchain uses the BERT large language model for identifying requirements in tender documents with an average accuracy of 82%. The toolchain also allows checking the quality of the extracted technical specifications from the tender documents. Particularly, VARA+ compute metrics, such as Automated Readability Index, Complexity, and subjectivity to allow quality assessment of the extracted requirements. We evaluated various binary classifiers for our requirements identification sub-tool to select the best one for the pipeline. As shown in Figure ReqIdentifier, we evaluate weighted random (W. Rand.), Support Vector Machine (SVM), multinomial Naive bayes (NMB), Decision trees (DT), Logistic Regression (LR), Random Forest (RF), BERT and its variants, and LSTM and its variants. We achieve an average accuracy of 82% in requirements identification in large tender documents with the BERT language model. Results also show that the BERT-based requirements identification approach performs the best in terms of precision (P), recall (R) and their harmonic mean (F1 score) with an average accuracy of 82%.



Pipeline	Setup	Weighted Average			Macro Average			Avg. A.	Time (mins)	
		Prec.	Rec.	F1	Prec.	Rec.	F1	A	Tr	Ts
W. Rand.	Freq. based	.49	.49	.49	.49	.49	.48	.49	-	-
SVM	Norm., PCA	.79	.79	.79	.80	.78	.78	.78	.70	.02
pSVM	Norm., PCA	.78	.78	.78	.79	.77	.77	.78	.74	.09
NB	Norm.	.74	.69	.69	.73	.71	.69	.69	<.01	<.01
pNB	Norm.	.74	.68	.67	.72	.70	.68	.68	.29	.07
DT	Norm.	.72	.72	.72	.71	.71	.71	.71	<.01	<.01
pDT	Norm.	.71	.71	.71	.71	.71	.71	.71	.29	.07
LR	Norm., PCA	.79	.79	.78	.79	.77	.78	.79	.30	<.01
pLR	Norm., PCA	.78	.78	.78	.79	.76	.77	.78	.41	.07
RF	Norm.	.79	.79	.79	.79	.78	.78	.79	<.01	<.01
pRF	Norm.	.79	.78	.78	.79	.77	.77	.78	.38	.07
LSTM	FT custom	.77	.77	.77	.76	.76	.76	.77	1	.02
pLSTM	FT custom	.75	.75	.75	.75	.75	.74	.75	1	.08
LSTM	FT pre-train	.75	.75	.75	.74	.74	.74	.75	2	.02
pLSTM	FT pre-train	.72	.72	.72	.72	.72	.72	.72	1.2	.08
LSTM	GLV custom	.77	.77	.77	.77	.76	.76	.77	2	.02
pLSTM	GLV custom	.76	.76	.76	.76	.75	.76	.76	1.2	.09
LSTM	GLV pre-train	.78	.78	.78	.78	.77	.78	.78	2	.02
pLSTM	GLV pre-train	.78	.78	.78	.78	.77	.78	.78	1.3	.08
SciBERT	uncased	**.82**	.81	.81	**.82**	.80	.80	.81	34	.25
pSciBERT	uncased	.80	.78	.76	.81	.75	.75	.78	32	.30
RoBERTa	base	.81	.81	.81	**.82**	.80	.80	.81	39	.27
pRoBERTa	base	.80	.79	.79	.81	.78	.78	.79	37	.32
BERT	base, cased	**.82**	**.82**	.81	**.82**	**.81**	**.81**	**.82**	35	.29
pBERT	base, cased	.79	.79	.79	.79	.79	.79	.80	32	.32
BERT	base, uncased	**.82**	**.82**	**.82**	**.82**	**.81**	**.81**	**.82**	34	.29
pBERT	base, uncased	.80	.80	.80	.80	.79	.79	.80	32	.33
XRBERT	base	**.82**	.81	.81	**.82**	.80	**.81**	.81	57	.29
pXRBERT	base	.78	.77	.77	.78	.76	.76	.77	41	.25
DisBERT	base, cased	.81	.81	.81	.81	.80	.80	.81	31	.13
pDisBERT	base, cased	.80	.80	.80	.80	.79	.79	.80	25	.18
DisBERT	base, uncased	.81	.81	.81	.81	**.81**	.80	.81	31	.15
pDisBERT	base, uncased	.80	.80	.70	.81	.78	.79	.80	29	.21
XLNet	base	.81	.81	.80	.81	.80	.80	.81	47	.36
pXLNet	base	.81	.80	.80	.81	.79	.79	.80	47	.42
S-BERT	10% train	.75	.75	.75	.75	.74	.75	.75	24	.14
pS-BERT	10% train	.73	.73	.73	.72	.72	.72	.73	18	.20
Mini-LM	10% train	.74	.74	.74	.74	.74	.74	.74	7	.04
pMini-LM	10% train	.72	.72	.72	.72	.72	.71	.72	6	.10
S-BERT	20% train	.77	.77	.76	.76	.76	.76	.77	45	.17
pS-BERT	20% train	.74	.74	.74	.74	.74	.74	.74	37	.20
Mini-LM	20% train	.75	.75	.75	.75	.74	.74	.75	14	.03
pMini-LM	20% train	.72	.72	.72	.72	.72	.72	.72	11	.10

Figure 33: ReqIndentifier: Evaluation results REQ-I

2) REQA: As a railway vehicle typically consists of more than 20 sub-systems, once the requirements are extracted, they must be allocated to various teams responsible for the development and testing of those sub-systems. In this regard, the VARA+ toolchain provides the REQA approach for the allocation of requirements to various teams. The approach combines large language models with case-based recommender systems to assign requirements to teams (and generate useful explanations for the allocation to enable a well-informed allocation decision.



Pipeline	Setup	Weighted Average			Macro Average			Avg. A.
		P	R	F1	P	R	F1	A
W. Rand.		.11	.11	.11	.07	.07	.07	.11
SVM	Norm., PCA	.65	.62	.60	**.73**	.53	.58	.64
pSVM	Norm., PCA	.66	.64	.62	.72	.56	.60	.65
MNB	Norm.	.56	.53	.47	.55	.31	.32	.52
pMNB	Norm.	.54	.54	.48	.50	.31	.32	.54
DT	Norm.	.48	.46	.46	.50	.46	.47	.46
pDT	Norm.	.49	.48	.48	.50	.46	.46	.48
LR	Norm., PCA	.64	.59	.56	.71	.43	.48	.59
pLR	Norm., PCA	.65	.61	.58	.73	.47	.51	.61
RF	Norm.	.61	.58	.57	.69	.52	.57	.58
pRF	Norm.	.61	.59	.58	.69	.54	.58	.59
SciBERT*	uncased	**.68**	.67	**.67**	.71	**.66**	**.67**	.67
pSciBERT	uncased	.64	.64	.63	.69	.61	.63	.64
RoBERTa	base	.66	.66	.65	.70	.64	.65	.66
pRoBERTa	base	.61	.61	.58	.64	.54	.55	.61
BERT	base, cased	.67	.67	.66	.69	.63	.64	.67
pBERT	base, cased	.65	.64	.62	**.73**	.57	.60	.64
BERT	base, uncased	**.68**	**.68**	.66	**.73**	.63	.65	**.68**
pBERT	base, uncased	.67	.66	.64	.71	.62	.64	.66
LSTM	FT custom	.53	.50	.49	.48	.43	.43	.50
pLSTM	FT custom	.58	.57	.56	.57	.54	.53	.57

Figure 34: REQAev: Evaluation with various pipelines for REQA

As shown in Figure *REQAev*, we evaluate various classifiers for the REQA approach on Alstom's use case. In particular, we evaluate weighted random (W. Rand.), Support Vector Machine (SVM), multinomial naive bayes (NMB), Decision trees (DT), Logistic Regression (LR), Random Forest (RF), BERT and its variants, and LSTM. Results show that BERT-based REQA approach performs the best in terms of precision (P), recall (R) and their harmonic mean (F1 score) with an average accuracy of 68%.

Summary:

In summary, with REQ-I it is possible to identify and extract technical requirements from large tender documents. The approach also makes the process more automated and less dependent on human expertise. On the other hand, we also support the allocation of the identified requirements with our REQA tool for smart allocation to teams within company for implementation and verification.

References:

[RISE1] Berry, D.M.: Empirical evaluation of tools for hairy requirements engineering tasks. Empirical Software Engineering 26(6), 1–77 (2021)
[RISE2] Winkler, J., Vogelsang, A.: Automatic classification of requirements based on convolutional neural networks. In: 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW). pp. 39–45. IEEE (2016)
[RISE3] Falkner, A., Palomares, C., Franch, X., Schenner, G., Aznar, P., Schoerghuber, A.: Identifying requirements in requests for proposal: A research

preview. In: International Working Conference on Requirements Engineering: Foundation for Software Quality. pp. 176–182. Springer (2019)
[RISE4] Abualhaija, S., Arora, C., Sabetzadeh, M., Briand, L.C., Traynor, M.: Automated demarcation of requirements in textual specifications: a machine learning-based approach. Empirical Software Engineering 25(6), 5454–5497 (2020)
[RISE5] Sainani, A., Anish, P.R., Joshi, V., Ghaisas, S.: Extracting and classifying requirements from software engineering contracts. In: 2020 IEEE 28th International Requirements Engineering Conference (RE). pp. 147–157. IEEE (2020)
[RISE6] Bashir, S., Abbas, M., Saadatmand, M., Enoiu, E.P., Bohlin, M., Lindberg, P. (2023). Requirement or Not, That is the Question: A Case from the Railway Industry. In: Ferrari, A., Penzenstadler, B. (eds) Requirements Engineering: Foundation for Software Quality. REFSQ 2023.
[RISE7] S. Bashir, M. Abbas, A. Ferrari, M. Saadatmand and P. Lindberg, "Requirements Classification for Smart Allocation: A Case Study in the Railway Industry," 2023 IEEE 31st International Requirements Engineering Conference (RE), Hannover, Germany, 2023, pp. 201-211, doi: 10.1109/RE57278.2023.00028.

4.3.4. Automatic issue labeling and similarity analysis using advanced natural language processing (IFAK, Software AG)

Synopsis:

Efficient management of software requirements and issues is a cornerstone of successful software development. In a productive development environment, several dozen or hundreds of new or adapted requirements or bug reports may appear per day and must be processed manually. An established way to process issues in a structured manner is to assign labels or tags so that they can be processed more quickly and in a more targeted manner. However, this involves a lot of manual work, reading the texts and discussing them if necessary. It requires expert knowledge, is very time-consuming and error prone. Classification of software requirements and issues is helpful for many purposes, such as prioritization in processing (e.g. less time for solving security-relevant issues), assigning specific people/teams for design, implementation and testing, creating specific test cases (e.g. performance testing) and supporting bug fixing (e.g. using knowledge from former bug fixes).

Another critical challenge in issue management is the risk posed by duplicates and strongly related issues. These occur when multiple users report the same or closely connected problems using slightly different descriptions or terminology. Such issues often appear not only within the same product but also across different versions, variants, or even entirely separate products, further complicating their detection. In fully manual processes, such connections are often overlooked, leading to inefficiencies where teams unknowingly address the same problem multiple times or fail to consider interlinked issues holistically. Identifying related issues can significantly reduce redundancy, allowing teams to resolve multiple instances of a problem simultaneously. Additionally, uncovering interlinked problems helps address root causes comprehensively, improving system stability and reducing recurring errors. By streamlining issue management, organizations can enhance efficiency, accelerate resolution times, and deliver more reliable software.

Related works:

Requirements classification is an evolving research area where state-of-the-art natural language processing (NLP) techniques have not yet been fully exploited. Much of the existing work relies on traditional machine learning approaches or keyword-based methods due to limited dataset access, with models like Support Vector Machines (SVM) and Naive Bayes being commonly used [IFAK1]. Text vectorization techniques like TF-IDF are often used to convert requirement text into a format suitable for these models [IFAK 2]. Recent advancements have introduced deep learning models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), including bidirectional LSTMs and GRUs, to enhance classification performance [IFAK 3]. However, many studies face challenges such as data scarcity and imbalanced datasets, often limiting classification to functional and non-functional requirements without delving into subcategories. The lack of standardized definitions for requirements further complicates classification, as the same text can be interpreted differently depending on its context. Few studies have explored alternative learning methods, with active learning, transfer learning [IFAK 4], and zero-shot learning [IFAK 5] offering promising directions for better handling limited data. In this context, the so-called "catastrophic forgetting" is the well-known Achilles' heel of deep neural networks, that the knowledge learned from previous tasks is forgotten when the networks are retrained to adapt to new tasks.

A similar situation can be seen with Duplicate Bug Report Detection. This research area of identifying duplicate bug reports involves various natural language processing methods. Current approaches are either statistical methods based on words or Machine Learning/Deep Learning models based on syntactic information. For example, BM25 is a traditional information retrieval method that relies on textual and categorical features, while Siamese Pair employs deep learning with LSTM and CNN to encode textual and categorical data separately. Ranking bug reports for duplicates is preferred over classification in real-world scenarios, as it reflects practical usage more accurately. Traditional classification settings often overestimate performance due to unrealistic assumptions about candidate pairs. [IFAK 6]

Methodology:

Within the project SmartDelta, IFAK has developed two tools for Automatic Issue Labeling (AILA) and Automatic Issue Similarity Analysis (AISA).

AILA – Automatic Issue Labeling Tool:

We developed solutions leveraging three primary learning techniques to address the challenges of incremental requirements classification:

Transfer Learning: We utilized knowledge from pre-trained models like BERT and fine-tuned them on domain-specific tasks. This approach allowed our model to generalize effectively with limited labeled data across diverse domains.

Multi-Task Learning (MTL): We trained models on multiple tasks concurrently, sharing representations to improve performance across related tasks. Our implementation included hard and soft parameter sharing to enhance data efficiency and minimize overfitting.

Continual Learning (CL): We adopted a sequential learning process, enabling the model to retain knowledge from previous tasks while adapting to new tasks. Techniques such as experience replay, elastic weight consolidation, and adapter modules were used to mitigate catastrophic forgetting.

Additionally, we addressed the issue of class imbalance through data augmentation methods like synonym replacement and back-translation, generating synthetic data for underrepresented classes. Class weighting schemes, including normalized and log-transformed weights, were employed to balance the influence of minority and majority classes during training. Our architecture was based on a BERT model with multiple classification heads tailored to specific tasks, such as security and other non-functional categories. By treating each dataset as a new task, we ensured adaptability to incremental data streams. We have applied recent continual lifelong learning methods to accumulate past knowledge and use it for future learning and knowledge reasoning. In this way, the model learns better with little data for incrementally growing datasets.

AISA – Automatic Issue Similarity Analysis Tool:

We developed an issue similarity analysis approach using advanced language models, specifically SentenceBERT, to provide a semantic approach for identifying related issues. The method leverages ChromaDB as a vector database to store semantic embeddings, ensuring scalability and efficiency for large datasets. Vector embeddings for a large database of historical issues are generated, which supports multiple sentence-transformer models. Querying for similar issues is conducted, where the top k most similar results can be retrieved. Cosine similarity is used as the metric to rank the similarity between issues. To enhance processing speed, we use the HNSW (Hierarchical Navigable Small World) algorithm, which reduces the computational load by narrowing down the comparison set intelligently. Before analysis, we apply manual pre-filtering to exclude non-relevant issues, optimizing the process and ensuring faster, more accurate results.

To improve interpretability, results can be visualized in a user interface, highlighting the most common words between two issues using the KeyBERT library. This enables clear identification of semantic overlaps between related issues.

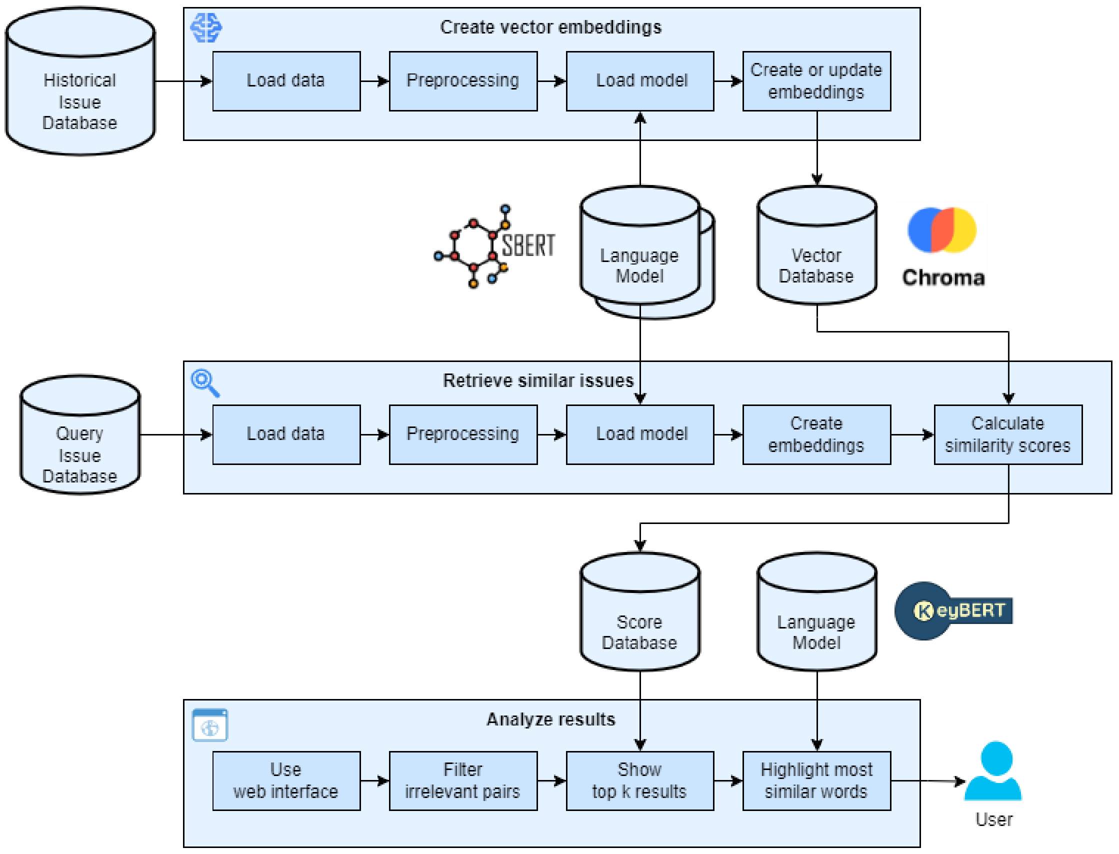


Figure 35: Automatic Issue Similarity Analysis pipeline

Results:

As a first step, IFAK has experimentally evaluated the two tools for Automatic Issue Labeling (AILA) and Automatic Issue Similarity Analysis (AISA) on publicly available data. In a second step, Software AG has thoroughly evaluated both tools for their use case, particularly with regard to security-related issues and recommendations for possible reuse.

AILA – Automatic Issue Labeling Tool:

In our experimental evaluation, we assessed the performance of various learning techniques for incremental requirements classification. We utilized five public datasets representing different software domains, such as security and web development, to ensure diversity in classification tasks (SecReq, CWE, Slankas, PURE, PROMISE). Our experiments involved splitting the datasets into training and testing sets, applying data augmentation techniques, and employing class weighting to address data imbalance. We focused on both macro and weighted F1 scores to evaluate the accuracy of classifications for both minority and majority classes.

We implemented and compared single-task learning, transfer learning, multi-task learning, and three variations of continual learning: experience replay, elastic weight consolidation, and adapter modules. Continual learning with experience replay emerged as the most effective approach, achieving the highest weighted F1 scores and the lowest forgetting measures, demonstrating robust retention of prior knowledge while adapting to new tasks. In contrast, single-task learning yielded the lowest scores, underscoring the limitations of isolated task training.

	Security class	Security sub-classes	Non-functional classes	Average
Macro F1 score				
TL	78.2±3.6	26.6±3.7	37.8±5.0	47.5±4.1
STL	68.3±3.9	0.0±0.0	20.2±1.8	29.5±1.9
MTL	87.5±3.0	39.1±5.1	**46.4±1.3**	57.7±3.1
CL-ER	**91.6±1.1**	**45.6±3.5**	43.1±2.8	**60.1±2.5**
CL-EWC	79.7±0.9	24.7±1.3	38.7±5.3	47.7±2.5
CL-Adapter	36.4±0.1	30.8±0.9	23.1±1.6	30.1±0.9
Weighted F1 score				
TL	83.1±3.1	42.2±4.5	79.1±2.1	68.1±3.2
STL	76.9±3.3	0.0±0.0	60.3±1.2	45.7±1.5
MTL	90.9±2.7	55.1±2.9	**84.0±0.8**	76.7±2.1
CL-ER	**94.9±0.7**	**61.1±1.4**	83.3±1.2	**79.8±1.1**
CL-EWC	84.5±0.8	40.2±2.3	79.7±2.4	68.2±1.8
CL-Adapter	61.7±0.1	45.4±0.7	62.1±0.6	56.4±0.5
Forgetting measure				
TL	20.1±3.1	7.9±2.2	13.7±2.4	13.9±2.6
STL	-	-	-	-
MTL	-	-	-	-
CL-ER	**3.1±0.5**	**0.0±0.4**	**5.1±1.8**	**2.7±0.9**
CL-EWC	18.1±1.1	8.6±1.7	13.6±3.2	13.4±2.0
CL-Adapter	0.0±0.2	0.6±0.6	0.9±1.1	0.5±0.6

Table 16: Automatic Issue Labeling using different ML techniques for security-related requirements

Software AG evaluated the tool initially using publicly available GitHub issues. The initial model, trained on a small dataset of GitHub issues, demonstrated promising results with an accuracy of approximately 80% in predicting whether issues were security-relevant. This phase highlighted the model’s ability to transfer learning, leveraging pre-trained knowledge from similar tasks. However, the dataset differences between GitHub and Software AG's internal data prompted additional fine-tuning. We conducted multiple fine-tuning experiments using internal datasets annotated with reliable security labels. Fine-tuning the GitHub model on the smaller SAG dataset improved accuracy slightly to around 83%, though false positives remained a challenge due to differences in style and content between the datasets. A subsequent experiment pre-finetuned the model on a much larger dataset of Software AG issues with uncertain labels before fine-tuning on reliable ground truth data. This approach showed a very high accuracy of **98.4%** on the large data set. However, the uncertain quality of the labels probably increased these results.

AISA – Automatic Issue Similarity Analysis Tool:
We conducted an experimental evaluation with publicly available datasets such as Microsoft Visual Studio Code with information of issue duplicates. We have compared traditional statistical methods such as TF-IDF and advanced language models such as Sentence-BERT. The results showed that even simple word-based statistical methods can achieve good performance. However, the use of language models provided a slight

improvement in accuracy, and further gains were achieved through careful preprocessing of the issues.
Despite these improvements, a similarity-based approach cannot detect all duplicates, as some are expressed in entirely different ways. There is an inherent trade-off between increasing the detection rate and the effort required to review more issue pairs. For instance, recommending only the top result (k=1) correctly identifies the duplicate in 32% of cases. Expanding to the top two recommendations (k=2) increases the success rate to 37% but doubles the number of issue pairs to review. When providing the top five recommendations (k=5), the duplicate is included in 44% of cases, illustrating the balance between accuracy and efficiency in duplicate detection.

Method	Precision	Recall
TF-IDF	52.0	21.7
S-BERT all-MiniLM-L6-v2	63.2	20.9
S-BERT sentence-t5-base	62.3	20.9
S-BERT all-mpnet-base-v2	56.0	25.8

Table 17: Automatic Issue Similarity Analysis using different ML techniques

The tool was rigorously evaluated at Software AG, where a team conducted a manual review of many issue pairs, refining the filtering process to ensure that irrelevant issues are excluded from the similarity analysis. This refinement process has proven essential, revealing a consistent number of relevant duplicate issues that can be further leveraged to improve software quality and reduce technical debt.

Summary:
The Automatic Issue Labeling Tool (AILA) helps development and test teams prioritize requirements and monitor software quality by automatically classifying non-functional properties, such as security relevance. This automation streamlines processes, enabling teams to focus on critical tasks while maintaining high-quality standards. The tool also aligns with the SmartDelta Methodology by linking historical issues to current requirements, providing insights across software versions.
The Automatic Issue Similarity Analysis Tool (AISA) identifies and connects related issues from past and current code commits, uncovering patterns, reoccurring problems, and reusable fixes. By leveraging historical data, it helps teams proactively manage software stability and improve development efficiency. Both tools support smoother development cycles, better management of technical debt, and enhanced software quality, and are planned for release as Open Source solutions.

References:

[IFAK1] M. Binkhonain and L. Zhao, “A review of machine learning algorithms for identification and classification of non-functional requirements,” Expert Systems with Applications: X, vol. 1, p. 100001, 2019.
[IFAK2] J. M. Pérez-Verdejo, A. J. Sánchez-García, J. O. Ocharán-Hernández, E. Mezura-Montes, and K. Cortés-Verdín, “Requirements and GitHub issues: An

automated approach for quality requirements classification," Programming and Computer Software, vol. 47, no. 8, pp. 704–721, 2021
[IFAK3] K. Kaur and P. Kaur, "SABDM: A self-attention based bidirectional-RNN deep model for requirements classification," Journal of Software: Evolution and Process, 2022.
[IFAK4] Hey, T., Keim, J., Koziolek, A., & Tichy, W. F. (2020). NoRBERT: Transfer learning for requirements classification. In 2020 IEEE 28th International Requirements Engineering Conference (RE) (pp. 169-179). IEEE.
[IFAK5] W. Alhoshan, A. Ferrari, and L. Zhao, "Zero-shot learning for requirements classification: An exploratory study," arXiv:2302.04723, 2023.
[IFAK6] Zhang, T., Han, D., Vinayakarao, V., Irsan, I. C., Xu, B., Thung, F., LO, D. & Jiang, L. (2023). Duplicate bug report detection: How far are we?. ACM Transactions on Software Engineering and Methodology, 32(4), 1-32.

4.3.5. LLM-based indexing for advanced semantic artefacts search in corpus-based reuse use case (Akkodis)

Synopsis:

This section explores the implementation of Large Language Model (LLM)-based indexing techniques to enhance semantic search capabilities for artifacts in a corpus-based reuse context. By leveraging advanced natural language processing and embedding models, the proposed approach aims to improve the retrieval of relevant artifacts—such as requirements, models, and code—thereby facilitating more efficient reuse and modification of existing resources.

Related Works:

The use of retrievers in Retrieval-Augmented Generation (RAG) pipelines presents numerous opportunities for improvement and optimization, spanning from the indexing phase to the augmentation phase. Various techniques can be applied at each step to enhance document parsing, indexing, storage, retrieval processes, and prompt augmentation using retrieved data, culminating in multi-stage and multi-agent answer distillation. Many RAG optimization methods focus on refining the retrieval process and the execution of prompts using the retrieved data [AKK1] [AKK2].
In the context of software artifact retrieval, the data differs significantly from natural language documents, introducing new challenges. For instance, [AKK3] explores the use of LLM-based metadata for filtering during the retrieval process. However, the generation of LLM-supported metadata for optimized indexing of software artifacts remains a largely unexplored area.

Methodology:

The proposed methodology involves the following steps:

1. Data Collection: Gather a diverse corpus of artifacts, including requirements, models, and code files.
2. LLM Integration: Employ a Large Language Model to generate a set of descriptive labels and comprehensive descriptions for each artifact. This enhances the metadata associated with the artifacts, making them more searchable and contextually relevant. This step brings all types of artifacts into a common space, which would not be achieved during embedding calculation due to the different characteristics of each artifact type.
3. Embedding Generation: Utilize embedding models to create a dense embedding vector for each artifact. This process transforms the artifacts into a numerical format that captures their semantic meaning.
4. Data Storage: Store all artifacts, along with their embedding vectors and generated metadata, in a PostgreSQL database. This database supports full-text search capabilities using tsvector and tsquery.
5. Search Implementation: Develop a user interface that supports both keyword-based and full-text semantic searches, enabling users to input queries in natural language or as specific keywords.

Results:
Results indicate that using direct embedding vector calculations for artifacts presents two main challenges:
Embedding models often lack a robust understanding of certain technical input data and file formats, leading to imprecise semantic representations.
Different file formats (natural language for requirements, specific formats for encoding UML state machines as models, and C++ code) result in varied representations in the embedding space, making it difficult for a single query to retrieve all relevant types of artifacts/files.
By using label lists and descriptions generated by an LLM, a shared language is introduced, allowing for the creation of a unified embedding space for all types of artifacts. Initial results are showing that the different artifacts are distributed and not separated in the space. This can be seen in Figure 5, where the distribution of labels is visualized by applying TD-IDF as a vectorizer, K-means for clustering, and PCA to bring data to a 2-dimensional space.
Figure 5 Visualization of artifact distribution in 2-dimensional space showing no separation of file types

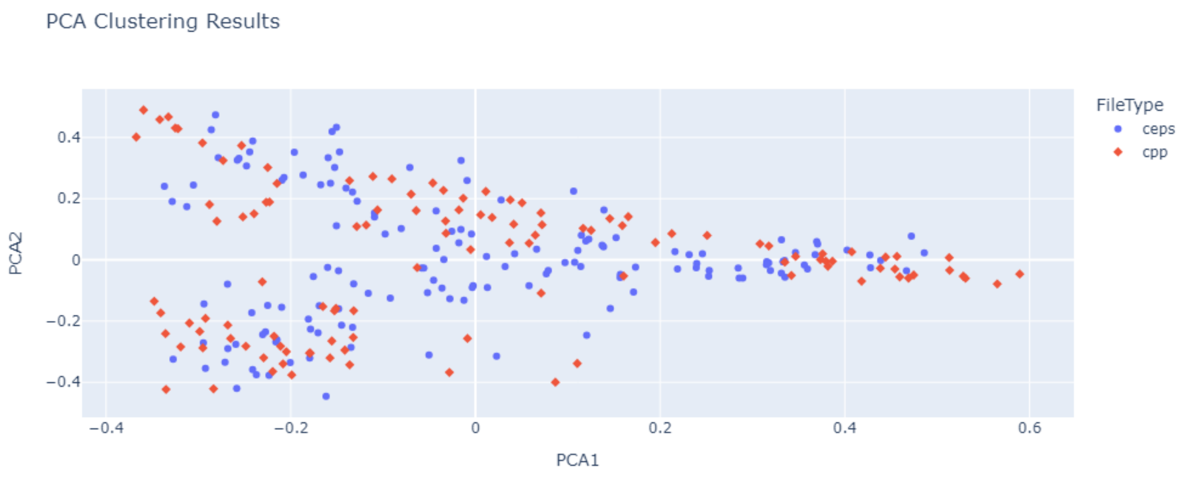


Figure 36: Visualization of artefact distribution in 2-dim space showing no separation of file types

The search results, while filtering and re-ranking the results from multi-stream retrievers (vector search, full-text, BM25, etc.), present many opportunities for further improvement. Thus, additional research and detailed evaluation are needed to stabilize and optimize search results across various software repositories.
The search results can reveal related file artifacts; however, this is not a recommendation for any kind of changes. It is up to the human user to assess these results, decide whether to use them as a basis for software changes, or provide this context to any AI-based software agent.

Summary:
In summary, the integration of LLM-based indexing for semantic artifact search represents a promising advancement in the field of corpus-based reuse. By enabling more context-aware and nuanced searches, this approach not only enhances the efficiency of artifact retrieval but also supports better decision-making in software development processes. Future work will focus on further refining the model and exploring additional applications of LLMs in software engineering contexts.

References

[AKK1] Gao, Y., et al. (2023). Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997.
[AKK2] Sawarkar, K., et al. (August 2024). Blended RAG: Improving RAG (Retriever-Augmented Generation) Accuracy with Semantic Search and Hybrid Query-Based Retrievers. arXiv preprint arXiv:2404.07220.
[AKK3] Poliakov, M., et al. (August 2024). Multi-Meta-RAG: Improving RAG for Multi-Hop Queries using Database Filtering with LLM-Extracted Metadata. arXiv preprint arXiv:2406.13213.

4.3.6. Predicting commercial charging station energy usage (eCamion and OntarioTechU)

Synopsis:

This section explores the implementation of Random Forest Regressor- based machine learning model to predict the hourly energy usage by a commercial charging station. Predicting the magnitude of energy used and the predicted peak usage hours bring additional values to charging station management and analysis.

Related works:

The charging station load prediction can be predicted using two data sources, using the customer profiles or through the station measurement. In Majidpour et al. [EC1], the two data sources were compared for their relation to load prediction accuracy using four different machine learning models (TWDP-NN, MPSF, SVR and RF), revealing that the datasets have no significant difference and viable for predictions. Using an application protocol Open Charge Point Protocol (OCPP) to collect station measurements from multiple charging stations, Renata et al. [EC2] predicted the daily energy use of commercial charging stations in Indonesia. Using features extracted from the charging records, the team compared four different machine learning models (RF, SVR, XGBoost and MLP) and evaluated them. Using the K-fold cross validation for evaluation, the result showed that Multilayer Perceptron (MLP) method had, and Random Forest Regression (RF) yielded the highest R2 and lowest error values respectively.

Methodology:

1. Data collection: Historical data of charging station transactions and meter values
2. Development of OCPP compliant systems: OCPP compliant charging station and Charging Station Management System (CSMS) are developed and implemented functionalities are tested using OCPP Compliance Testing Tool
3. Training and Prediction: Features are analyzed and extracted from the charging station transaction and meter value data. Due to lacking historical OCPP data collected, historical data was provided by eCamion for model training.
4. Model selection: Compare and evaluate the prediction from different machine learning models
5. Visualization: The predicted energy load is displayed on management dashboard, along with other real-time charging station readings from the OCPP for charging station analysis

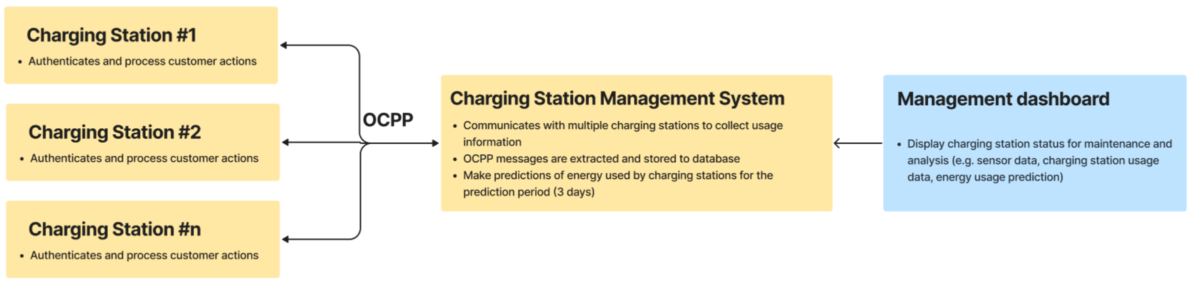

Charging Station #1
Authenticates and process customer actions
Charging Station #2
Authenticates and process customer actions
Charging Station #n
Authenticates and process customer actions
OCPP
Charging Station Management System
Communicates with multiple charging stations to collect usage information
OCPP messages are extracted and stored to database
Make predictions of energy used by charging stations for the prediction period (3 days)
Management dashboard
Display charging station status for maintenance and analysis (e.g. sensor data, charging station usage data, energy usage prediction)

Figure 37: Overview of the approach to data collection and prediction

Results:

During the feature extraction stage, it was discovered that the selecting the hourly interval had influence over model accuracy. The hourly interval was therefore selected to maximize the model's accuracy while providing useful and relevant data for management.

Based on the transaction records from multiple charging stations, charging stations can have different charging patterns based on variety of different factors including location and customer behaviours. Due to the difficulties of capturing factors outside of reporting functionalities of OCPP, the training data was added a label to classify the charging pattern based on time, which was found to increase accuracy of the models.

For model selection, four machine learning models, Random Forest, SVM, LSTM and Prophet were used to compare and their outcomes evaluated, where Random Forest Regression was found to yield highest accuracy.

Table 16: Models tested for energy load prediction and their evaluation of rolling cross validation

	MAE	RMSE	R2
Random Forest Regression	0.0101	0.0134	0.3278
SVR	0.0509	0.0537	-10.5937
LSTM	0.0342	0.0455	0.09445
Prophet	0.005	0.0666	0.0636

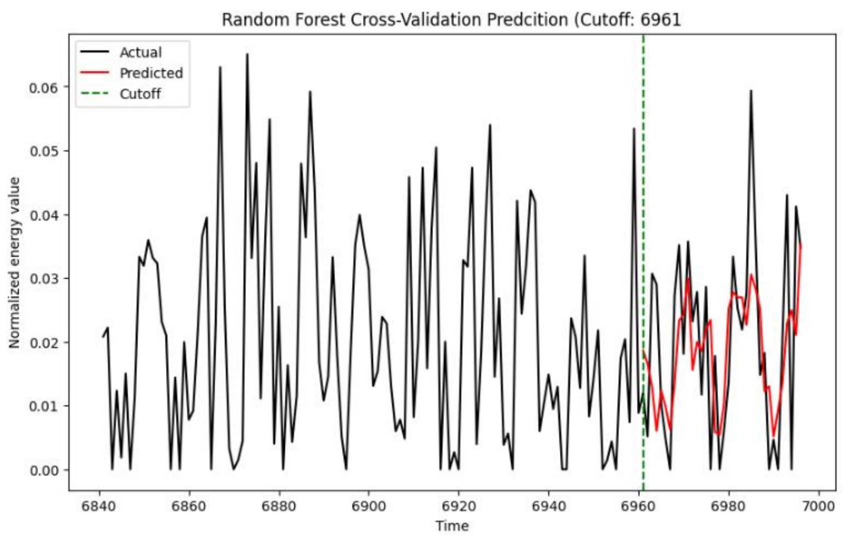


Figure 38: Example of prediction cycle made using Random Forest Regression during cross-validation process

Summary:

Using historical charging station's measurement data, energy consumption of a charging station can be predicted by the selected hourly interval. To increase the accuracy of the model, additional features were engineered including labels based on charging pattern.

The outcome of the prediction system adds additional insight for the management and lays ground for potential work for cost-saving solutions. Using the historical charging station's measurements.

References

[EC1] Majidpour, Mostafa, et al. "Forecasting the EV charging load based on customer profile or station measurement?." *Applied energy* 163 (2016): 134-141.
[EC2] Renata, Dionysius A., et al. "Modelling of electric vehicle charging energy consumption using machine learning." *2021 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. IEEE, 2021.

5. Tools/Technologies Developed in WP4

SoHist is an open-source tool developed by the University of Innsbruck and c.c.com to manage technical debt through retrospective code analysis. It extends SonarQube by providing historical insights into technical debt evolution, offering comprehensive GIT history quality analysis and visualizations. Developers can analyze the long-term impact of their decisions on maintenance costs and risks.

The **EPS Cybersecurity Anomaly Detector**, created by Glasshouse Systems and Ontario Tech University, integrates with QRadar to identify network anomalies using unsupervised machine learning models. By analyzing event-per-second (EPS) data, this closed-source tool provides graphical visualizations of anomalies to enhance the efficiency of SOC operations.

The **Offense Prioritization App**, also developed by Glasshouse Systems and Ontario Tech University, leverages machine learning to prioritize critical security threats. Integrated with SIEM solutions, it ranks offenses based on risk levels, enabling SOC analysts to address high-priority issues more effectively.

INIMASU, an open-source tool by Fraunhofer FOKUS, supports intelligent issue management by optimizing scheduling and classification. It processes data from Git repositories and configuration files to produce management reports and predictions, aiding in decision-making and resource allocation.

ReqIdentifier simplifies requirement identification in large tender documents using machine learning classifiers. This partially open-source tool processes PDF or CSV files, highlighting requirements for better scalability and accuracy in bidding processes.

DIA4M, an open-source tool by NetRD, focuses on detecting faults and anomalies in microservice interactions. By processing logs from CSV or ElasticSearch, it generates visualized reports to aid DevOps engineers in quality assurance and fault detection.

YATAP, a licensed tool by Erste, performs comprehensive change impact analysis by integrating data from Jira, GitHub, SonarQube, and other sources. It outputs data in ElasticSearch or PostgreSQL formats to help assess the effects of changes on systems.

AILA, developed by IFAK and Software AG, automates issue labeling using fine-tuned BERT models. This planned open-source tool classifies requirements or issues based on descriptions, aiding in prioritization and team assignments.

AISA, developed by IFAK and Software AG, automates issue similarity analysis using language models such as Sentence-BERT. This planned open-source tool provides similar requirements or issues based on descriptions, aiding in code and test reuse recommendations.

VARA+, a closed-source tool by RISE, automates asset reuse analysis and assesses requirements quality. By analyzing CSV files, it predicts reusable assets and computes metrics to enhance efficiency in software projects.

SmartTrace, a closed-source tool by Akkodis Research, enables semantic searches to locate and analyze reusable artifacts. By processing natural language or keyword queries, it provides a list of related artifacts for better reuse and efficiency.

ReqAllocator (REQA) automates requirements allocation and classification using machine learning and deep learning techniques. This closed-source tool processes CSV files to recommend allocations and augmentations, facilitating efficient team assignments.

RADICLE, a closed-source tool by RISE, leverages LLMs to detect ambiguities in requirements and provide rational explanations. It processes CSV files to identify ambiguous requirements and their rationale, improving clarity and quality.

Telemetry Anomaly Analyzer by Hoxhunt detects telemetry anomalies in distributed systems using OpenTelemetry-compatible data. This closed-source tool generates anomaly heatmaps and dashboards, enabling efficient monitoring and troubleshooting.

RAG-based QA Chatbot uses retrieval-augmented generation and LLMs to answer requirement-related queries. This closed-source tool processes text-based inputs and produces comprehensive answers, aiding engineers in understanding software releases.

DETANGLE, by Cape of Good Code, provides dashboards and visualizations to analyze technical debt and its impact. This closed-source tool processes data from issue trackers, code repositories, and test coverage reports to support root cause analysis and quality improvements.

Modernization Toolkit by Vaadin analyzes Java code for compatibility with Vaadin and applies transformations to update source code. This closed-source tool generates summaries of transformation coverage and transformed code for efficient modernization.

SONATA leverages ontologies and knowledge graphs to recommend test cases for new code. This closed-source tool processes code repositories and outputs tailored test case recommendations, improving software quality management.

Code Similarity Investigator (CSI), by TWT, provides automated code reuse suggestions using Code Property Graphs. It processes source code to identify similar sections and suggests improvements, enhancing efficiency and maintainability.

Graph Similarity Recommender (GSR) compares state machines to identify similarities and recommend comparable ones. This closed-source tool processes state machine data and outputs similarity values and delta paths to streamline analysis.

Team Eagle QA Tool analyzes software quality metrics for cloud-based software hosted on Microsoft Azure. This closed-source tool provides quality assurance metrics to help engineers monitor and improve software quality.

ReqIdentifier (RADICLE) uses LLMs to detect ambiguous requirements in CSV files, providing classifications and rationales. This planned open-source tool improves requirements clarity and supports quality assurance efforts.

These tools collectively address critical challenges in software engineering, offering innovative solutions to improve quality, efficiency, and maintainability across various domains.

6. Conclusions

This report highlights the significant contributions made in Work Package 4, focusing on advancing the state of the art in software quality trend analysis and prediction, similarity analysis and reuse recommendation, and change impact analysis and prediction. Key achievements include the development of novel machine learning methodologies for identifying and analysing quality trends, enabling predictive insights across various domains. These methodologies have enabled automated detection of quality improvements and degradations, streamlining continuous engineering workflows.

In the area of anomaly detection and offense prioritization, advanced ML techniques were employed to identify anomalies in complex systems and prioritize cybersecurity threats effectively. These solutions have demonstrated significant improvements in operational stability, reducing detection and response times, and ensuring the robustness of live systems in domains such as micro-service architectures and telemetric data analysis. In similarity analysis and reuse recommendation, innovative graph-based and ML-driven techniques were developed to identify reusable components and design artifacts, significantly enhancing efficiency in software evolution. Additionally, the work package introduced advanced change impact analysis tools that predict and evaluate the effects of software changes, providing actionable recommendations for maintaining and improving system quality.

These contributions collectively establish a strong foundation for improving software quality across industrial domains, offering scalable, intelligent solutions that address critical challenges in modern software engineering. The methodologies and tools developed in this work package pave the way for further innovation and integration into diverse software development environments.

7. References

1. Kaner, C. (2004). Software engineering metrics: What do they measure and how do we know? In In METRICS 2004. IEEE CS.
2. Galin, D. (2018). Software quality: concepts and practice. John Wiley & Sons.
3. Dlamini, G., Ergasheva, S., Kholmatova, Z., Kruglov, A., Sadovykh, A., Succi, G., ... & Zouev, E. (2022). Metrics for Software Process Quality Assessment in the Late Phases of SDLC. In Science and Information Conference (pp. 639-655). Springer, Cham.
4. Masuda, S., Ono, K., Yasue, T., & Hosokawa, N. (2018, April). A survey of software quality for machine learning applications. In 2018 IEEE International conference on software testing, verification and validation workshops (ICSTW) (pp. 279-284). IEEE.
5. Reddivari, S., & Raman, J. (2019, July). Software quality prediction: an investigation based on machine learning. In 2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI) (pp. 115-122). IEEE.
6. Karunanithi, N., Malaiya, Y. K., & Whitley, L. D. (1991, May). Prediction of software reliability using neural networks. In ISSRE (pp. 124-130).
7. Dragicevic, S., Celar, S., & Turic, M. (2017). Bayesian network model for task effort estimation in agile software development. Journal of systems and software, 127, 109-119.
8. Ruk, S. A., Khan, M. F., Khan, S. G., & Zia, S. M. (2019, December). A survey on adopting agile software development: issues & its impact on software quality. In 2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICES) (pp. 1-5). IEEE.
9. Mahfuz, A. S. (2016). Software Quality Assurance: Integrating Testing, Security, and Audit. CRC Press.
10. Najafabadi, M. M., Villanustre, F., Khoshgoftaar, T. M., Seliya, N., Wald, R., & Muharemagic, E. (2015). Deep learning applications and challenges in big data analytics. Journal of big data, 2(1), 1-21.

11. Rashid, E., Patnaik, S., & Bhattacherjee, V. (2012). Software quality estimation using machine learning: Case-Based reasoning technique. International Journal of Computer Applications, 58(14).
12. Puri, A., & Singh, H. (2014). Genetic algorithm based approach for finding faulty modules in open source software systems. International Journal of Computer Science and Engineering Survey, 5(3), 29.
13. Abouelela, M., & Benedicenti, L. (2010). Bayesian network based XP process modelling. arXiv preprint arXiv:1007.5115.
14. Ebert, C., Cain, J., Antoniol, G., Counsell, S., & Laplante, P. (2016). Cyclomatic complexity. IEEE Software, 33(6), 27-29.
15. Ahmed, M. A., & Al-Jamimi, H. A. (2013). Machine learning approaches for predicting software maintainability: a fuzzy-based transparent model. IET software, 7(6), 317-326.
16. M. Kretsou, E.M. Arvanitou, A. Ampatzoglou, I. Deligiannis, and V. C. Gerogiannis, “Change impact analysis: A systematic mapping study”, *Journal of Systems and Software*, vol. 174, Dec. 2020.
17. S. A. Bohner, “Impact analysis in the software change process: A year 2000 perspective,” *Proceedings of International Conference on Software Maintenance ICSM-96*, pp. 42–51, Aug. 2002.
18. B. Li, X. Sun, H. Leung, and S. Zhang, “A survey of code-based Change Impact Analysis Techniques,” *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, Apr. 2012.
19. S. A. Bohner, “Software change impacts-an evolving perspective,” *International Conference on Software Maintenance, 2002. Proceedings.*, pp. 263–271, Jan. 2003.
20. A. De Lucia, F. Fasano, and R. Oliveto, “Traceability management for Impact Analysis,” *2008 Frontiers of Software Maintenance*, pp. 21–30, Nov. 2008.
21. R. S. Arnold and S. A. Bohner, “Impact analysis-towards a framework for comparison,” *1993 Conference on Software Maintenance*, pp. 292–301, Sep. 1993.
22. L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Dam, “On the precision and accuracy of Impact Analysis Techniques,” *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, pp. 513–518, May 2008.
23. X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, “Change impact analysis based on a taxonomy of change types,” *2010 IEEE 34th Annual Computer Software and Applications Conference*, pp. 373–382, Jul. 2010.
24. B. Riberio-Neto and R. Baeza-Yates, *Modern Information Retrieval*. Boston: Addison–Wesley Longman Publishing Co., 1999.
25. S. Jiang, C. McMillan, and R. Santelices, “Do programmers do change impact analysis in debugging?” *Empirical Software Engineering*, vol. 22, no. 2, pp. 631–669, Jul. 2016.
26. A. Dhamija and S. Sikka, “A systematic review of feature location techniques under software change impact analysis,” *International Journal of Computer Sciences and Engineering*, vol. 7, no. 3, pp. 184–192, Mar. 2019.
27. M. Shahid and S. Ibrahim, “Change impact analysis with a software traceability approach to support software maintenance,” *2016 13th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pp. 391–396, Jan. 2016.
28. S. Kugele and D. Antkowiak, “Visualization of trace links and change impact analysis,” *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, pp. 165–169, Sep. 2016.
29. B. Dit, “Configuring and assembling information retrieval based solutions for software engineering tasks,” *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 641–646, Jan. 2017.

30. D. Kchaou, N. Bouassida, and H. Ben-Abdallah, “UML models change impact analysis using a text similarity technique,” *IET Software*, vol. 11, no. 1, pp. 27–37, Oct. 2016.
31. S. Nejati, M. Sabetzadeh, C. Arora, L. C. Briand, and F. Mandoux, “Automated change impact analysis between SysML models of requirements and design,” *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 242–253, Nov. 2016.
32. A. Ghabi and A. Egyed, “Exploiting traceability uncertainty among artifacts and code,” *Journal of Systems and Software*, vol. 108, pp. 178–192, Jun. 2015.
33. N. Almasri, L. Tahat, and B. Korel, “Toward automatically quantifying the impact of a change in Systems,” *Software Quality Journal*, vol. 25, no. 3, pp. 601–640, May 2016.
34. T. Sharma and G. Suryanarayana, “Augur: Incorporating hidden dependencies and variable granularity in Change Impact Analysis,” *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 73–78, Oct. 2016.
35. T. Rolfsnes, S. Di Alesio, R. Behjati, L. Moonen, and D. W. Binkley, “Generalizing the analysis of evolutionary coupling for software change impact analysis,” *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 201–212, May 2016.
36. V. Musco, M. Monperrus, and P. Preux, “A large-scale study of call graph-based impact prediction using mutation testing,” *Software Quality Journal*, vol. 25, no. 3, pp. 921–950, Jul. 2016.
37. H. Cai and R. Santelices, “A framework for cost-effective dependence-based dynamic impact analysis,” *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2015.
38. B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vásquez, D. Poshyvanyk, and H. Kagdi, “Impactminer: A tool for change impact analysis,” *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 540–543, May 2014.
39. A. Rajan and D. Kroening, “Measuring change impact on program behaviour,” *Validation of Evolving Software*, pp. 125–145, Jul. 2015.
40. H. Cai and R. Santelices, “A comprehensive study of the predictive accuracy of dynamic change-impact analysis,” *Journal of Systems and Software*, vol. 103, no. C, pp. 248–265, May 2015.
41. A. Goknil, R. van Domburg, I. Kurtev, K. van den Berg, and F. Wijnhoven, “Experimental evaluation of a tool for change impact prediction in requirements models: Design, results, and lessons learned,” *2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE)*, pp. 57–66, Aug. 2014.
42. A. Goknil, I. Kurtev, K. van den Berg, and W. Spijkerman, “Change impact analysis for requirements: A metamodeling approach,” *Information and Software Technology*, vol. 56, no. 8, pp. 950–972, Mar. 2014.
43. M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, “Making refactoring safer through impact analysis,” *Science of Computer Programming*, vol. 93, pp. 39–64, Nov. 2014.
44. M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, “Integrated Impact Analysis for managing software changes,” *2012 34th International Conference on Software Engineering (ICSE)*, pp. 430–440, Jun. 2012.
45. G. J. Holzmann, “Cobra: A light-weight tool for static and dynamic program analysis,” *Innovations in Systems and Software Engineering*, vol. 13, no. 1, pp. 35–49, Jun. 2016.
46. S. Basri, N. Kama, R. Ibrahim, and S. A. Ismail, “A change impact analysis tool for software development phase,” *International Journal of Software Engineering and Its Applications*, vol. 9, no. 9, pp. 245–256, Sep. 2015.

47. S. L. Pfleeger and S. A. Bohner, “A framework for software maintenance metrics,” *Proceedings. Conference on Software Maintenance 1990*, pp. 320–327, Nov. 1990.
48. K. H. Bennett and V. T. Rajlich, “Software maintenance and evolution,” *Proceedings of the conference on The future of Software engineering - ICSE '00*, May 2000.
49. H. Cai and D. Thain, “Distia: A cost-effective dynamic impact analysis for distributed programs,” *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Aug. 2016.
50. L. Zhang, M. Kim, and S. Khurshid, “FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs,” *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, pp. 1–4, Nov. 2012.
51. T. Savage, M. Revelle, and D. Poshyvanyk, “FLAT3: feature location and textual tracing tool,”*Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, pp. 255–258, May 2010.
52. J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, “JRipples: A tool for program comprehension during Incremental Change,” *13th International Workshop on Program Comprehension (IWPC'05)*, May 2005.
53. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, “Chianti: a tool for change impact analysis of java programs,” *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 432–448, Oct. 2004.
54. M. Kretsou, E.-M. Arvanitou, A. Ampatzoglou, I. Deligiannis, and V. C. Gerogiannis, “Change impact analysis: A systematic mapping study,”*Journal of Systems and Software*, vol. 174, Dec. 2020.
55. D. Binkley, N. Gold, and M. Harman, “An empirical study of static program slice size,” *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 2, p. 8, Apr. 2007.
56. M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip, “Finding failure-inducing changes in java programs using change classification,” *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14*, pp. 57–68, Nov. 2006.
57. S. P. Day Galbo,“A Survey of Impact Analysis Tools for Effective Code Evolution,”M.S. thesis, Florida Ins. Tech., Central Florida Univ., Melbourne, Apr. 2017. [Online]. Available: https://repository.lib.fit.edu/handle/11141/1438
58. T. W. Aung, H. Huo, and Y. Sui, “A literature review of automatic traceability links recovery for software change impact analysis”, *Proceedings of the 28th International Conference on Program Comprehension*, pp. 14–24, Jul. 2020.
59. T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes”, *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, Jul. 2004.
60. T. Apiwattanapong, A. Orso, and M. J. Harrold, “Efficient and precise dynamic impact analysis using execute-after sequences”, *Proceedings of the 27th international conference on Software engineering - ICSE '05*, May 2005.
61. M. Shahid and S. Ibrahim, “Change impact analysis with a software traceability approach to support software maintenance,” *2016 13th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pp. 391–396, Jan. 2016.
62. Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J., and Morari, A., “Learning to map source code to software vulnerability using code-as-a-graph”, arXiv e-prints, 2020. doi:10.48550/arXiv.2006.08614.

63. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.
64. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.
65. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
66. Ukkonen, Esko (1985). "Algorithms for approximate string matching". Information and Control. 64 (1–3): 100–118.
67. Abbas, M., Saadatmand, M., Enoiu, E., Sundamark, D., & Lindskog, C. (2020). Automated reuse recommendation of product line assets based on natural language requirements. In International Conference on Software and Software Reuse (pp. 173-189). Springer, Cham.
68. Borg, M., Wnuk, K., Regnell, B., Runeson, P.: Supporting change impact analysis using a recommendation system: An industrial case study in a safety-critical context. IEEE Transactions on Software Engineering 43(7), 675–700 (2016)
69. Abbas, M., Ferrari, A., Shatnawi, A., Enoiu, E. P., & Saadatmand, M. (2021). Is requirements similarity a good proxy for software similarity? an empirical investigation in industry. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 3-18). Springer, Cham.
70. Abbas, M., Ferrari, A., Shatnawi, A., Enoiu, E., Saadatmand, M., & Sundmark, D. (2022). On the relationship between similar requirements and similar software. Requirements Engineering, 1-25.
71. Walker, A., Cerny, T., Song, E.: Open-source tools and benchmarks for code-clone detection: past, present, and future trends. ACM SIGAPP Applied Computing Review 19(4), 28–39 (2020)
72. Kaindl, H., & Mannion, M. (2015). A feature-similarity model for product line engineering. In International Conference on Software Reuse (pp. 34-41). Springer, Cham.
73. Davril, J. M., Delfosse, E., Hariri, N., Acher, M., Cleland-Huang, J., & Heymans, P. (2013, August). Feature model extraction from large collections of informal product descriptions. In proceedings of the 2013 9th joint meeting on foundations of software engineering (pp. 290-300).
74. OMA compliant API, http://www.openmobilealliance.org/wp/API_Inventory.html
75. M.-C. Lee, "Software Quality Factors and Software Quality Metrics to Enhance Software Quality Assurance," Br J Appl Sci Technol, vol. 4, no. 21, pp. 3069–3095, Jan. 2014, doi: 10.9734/BJAST/2014/10548
76. Benedikt Dornauer, Michael Felderer, Johannes Weinzerl, Mircea-Cristian Racasan, and Martin Hess. 2023. SoHist: A Tool for Managing Technical Debt through Retro Perspective Code Analysis. In Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE '23). Association for Computing Machinery, New York, NY, USA, 184–187. https://doi.org/10.1145/3593434.3593460
77. https://www.splunk.com/en_us/blog/learn/incident-response-metrics.html