SmartDelta

Automated Quality Assurance and Optimization in Incremental Industrial Software Systems Development

D1.6 - SmartDelta in Industrial Environments - Use Case Report

Submission date of deliverable: Feb 28, 2025

Edited by: Nicolas Bonnotte (Akkodis, Germany), Andreas Dreschinski (Akkodis, Germany), Martin Heß (Software AG, Germany), Robin Gröpler (ifak, Germany), Abhishek Shrestha (Fraunhofer FOKUS, Germany), Benedikt Dornauer (University of Innsbruck, Austria), Johannes Weinzerl (c.c.com, Austria), Mircea-Cristian Racasan (c.c.com, Austria), Mehrdad Saadatmand (RISE, Sweden), Muhammad Abbas (RISE, Sweden), Jean Malm (Mälardalen University, Sweden), Zulqarnain Haider (Alstom, Sweden), Muhammad Nouman Zafar (Mälardalen University, Sweden), Hakan Kilinc (NetRD, Turkey), Volkan Karsan (Kuveyt Türk, Turkey), Selahattin Furkan Karahan (Kuveyt Türk, Turkey), Merve Can Kuş (Kuveyt Türk, Turkey), Andrea Pabón-Guerrero (Universidad Carlos III de Madrid, Spain), Can Balcı (NetRD, Turkey), Yuriy Yevstihnyeyev (Vaadin, Finland), Ömercan Devran (Arcelik, Turkey), Baykal Mehmet Ucar (Arcelik, Turkey), Youhan Monsoon Fu (eCAMION, Canada), Emanuel Remneantu (TWT GmbH, Germany), Md Asif Khan (Ontario Tech University, Canada), Hyon Lee (Ontario Tech University, Canada)

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	WP1 Partners
Version number	V1.0

Work package	WP1
Work package leader	Nicolas Bonnotte (Akkodis)
Dissemination level	Public
Description	This deliverable reports the final evaluation results of WP1 on the application of the SmartDelta methodology and tools to the use cases.

Executive Summary

SmartDelta engages in collaborative research with a diverse range of industrial partners, encompassing sectors such as railway, e-mobility, telecommunications, finance and banking, enterprise software, logistics, personal mobility, and cybersecurity. The objective of WP1 is to integrate the disparate use cases into a unified approach that aligns their definition, development, and evaluation with the overarching objectives of SmartDelta.

In the initial phase of WP1, the specifications of the use cases were established, including the evaluation criteria, requirements, and the current state of existing systems (Year 1). Subsequently, a preliminary series of experiments is conducted via WP 2, 3, and 4, resulting in an intermediate evaluation and set of recommendations (Year 2). In the final year of the project, the present evaluation is performed.

This report presents the final evaluation of the tools developed and applied within the SmartDelta use cases, as well as their integration into the SmartDelta methodology. The objective was to provide both qualitative and quantitative feedback, evaluated against the project's key performance indicators (KPIs).

Each of the 11 use cases provides a comprehensive evaluation and recommendations for the implementation of SmartDelta in industry. This offers a detailed overview of the project's achievements and benefits across the various industrial domains involved.

Table of Contents

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)
ECU	Electronic Control Unit
EFP	Extra-Functional Property
FinTech	Financial Technology
FM	Feature Modelling
FR	Functional Requirement
FODA	Feature-Oriented Domain Analysis
IoT	Internet of Things
IPR	Intellectual property rights
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
V&V	Verification and Validation

1. Introduction

The SmartDelta project encompasses 11 use cases (see table 1 below), which span various industry domains. Each use case has been designed with the objective of deriving benefits from participation for the use-case provider. To ensure that the SmartDelta solutions can provide effective support for the selected industrial use cases, a requirements baseline has been established for each case, taking the technology, processes involved and business requirements into account (pre-conditions). The evaluation criteria and values have been set against the baseline values during WP1 and throughout the project as a whole. These criteria are aligned with the KPIs of the SmartDelta project, thereby ensuring that the evaluations are meaningful reflections of the project's performance.

The SmartDelta tools and methodology [1] have been developed with the objective of satisfying the criteria and achieving them. The methodology development is conducted in conjunction with continuous evaluation, a strategy that has been demonstrated to be an effective means of elaborating the SmartDelta methodology in a step-by-step manner (see Figure 1 below), thereby ensuring its suitability for industrial applications.

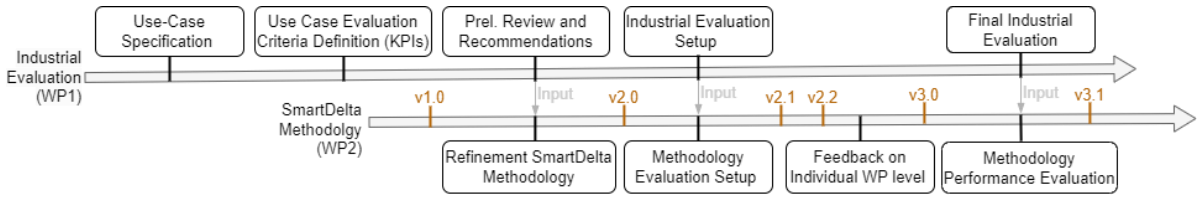


Figure 1: Step-by-step industrial evaluation setup and contribution to the Methodology

This document is organized into three main sections.

Section 2 outlines the evaluation framework and environment at the project level, with a particular focus on the SmartDelta Methodology and the metrics employed for the evaluations.

Sections 3 to 13 are use-case specific, presenting the use-case definition, tools and methodology integration, along with the evaluation results and recommendations for industrial domain adoption. An exhaustive list and basic use-cases description is provided in the table 1 below.

The document concludes with an overview of the project KPIs and an examination of their implications for the software industry in the sections 14 and 15.

Table 1: 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	Akkodis	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
UC9	Spain	Izertis	Enterprise Software	Semantic matchmaking
UC10	Finland	Vaadin	Software development platform	Continuous quality, security, and performance improvement in software development platform
UC11	Turkey	Arcelik	Home Appliances	Measure Software Product and Process Quality of Enterprise Solutions

2. Evaluation Objectives

a. SmartDelta Methodology

The SmartDelta approach entails the development of automated solutions for the assessment of product deltas within a continuous engineering context. To facilitate comprehension of the interconnections between specific processes and solutions, a software delta management concept, designated as the SmartDelta Methodology, has been created (see figure 2 below). The SmartDelta Methodology is presented in detail with examples in the SmartDelta D2.4 - SmartDelta Methodology Users and Developers Guidelines i

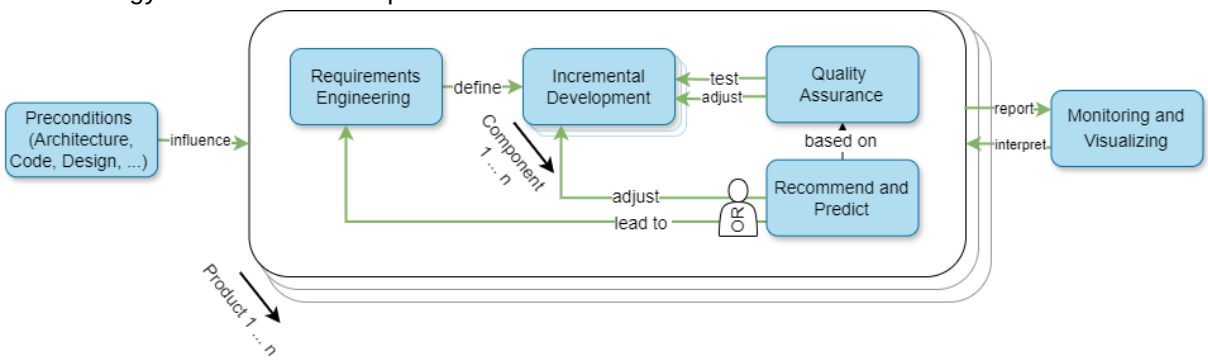


Figure 2: SmartDelta Methodology diagram

The present document offers a comprehensive evaluation of the SmartDelta methodology in a variety of use cases, with the objective of assessing the methodology's adaptability across multiple industrial domains.

b. Requirements, metrics and KPIs

SmartDelta's performance is measured through four main sets of KPIs (See Table 2 below).

Table 2: SmartDelta main sets of KPIs

KPI ID	Project KPI Family
1	Key Innovation related KPIs
2	Unique selling proposition KPIs
3	Progress on market access KPIs – exploitation and deployment
4	Progress on market access KPIs - dissemination

The Use-Cases Evaluations presented in this report are related to the ID 1 (Key Innovation) and ID 2 (Unique selling proposition) and can be found in the section 14.
The use-cases and the considered software are defined through functional and non-functional requirements. At their definition, each requirement has been mapped with the project KPIs. Therefore, each Use-Case performs an evaluation based on the requirements. The SmartDelta

performance is measured by considering all requirements contributing to the KPIs. One example of this linkage is shown in the Figure 3.

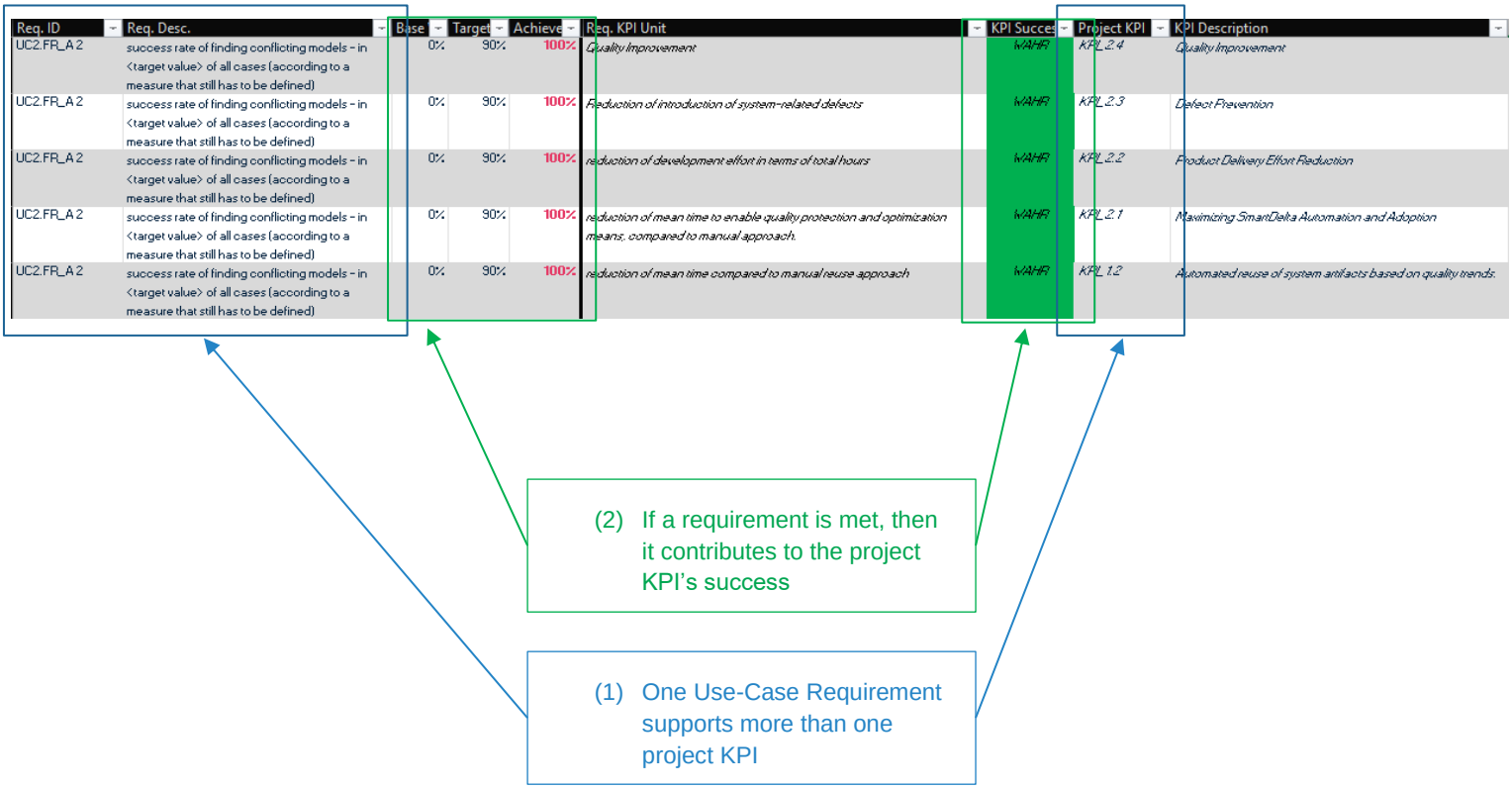


Figure 3: Use-Case requirement and project KPIs

3. Use-Case 1 from Alstom

a. Use-Case Description

Alstom is evolving mobility worldwide and making it easier for people to connect with one another. However, people choose to travel, they'll find an Alstom product ready to transport them. Our vast offering of public transit products (e.g., high speed trains) that're smarter than ever.

Alstom answers the call for more efficient, sustainable, and enjoyable transportation everywhere. Our vehicles, services and, most of all, our employees are what make us a global leader in transportation. We partner with customers, local organizations, and all stakeholders to help build communities and improve quality of life wherever we do business. Alstom is present in more than 70 countries, with 250 sites among them. Our 70,000 employees push mobility forward by creating rail transportation products adapted for the travellers of today and tomorrow. In the fiscal year ended 2020/2021, we posted sales of 14 billion Euro with an order backlog of 74.5 billion Euro.

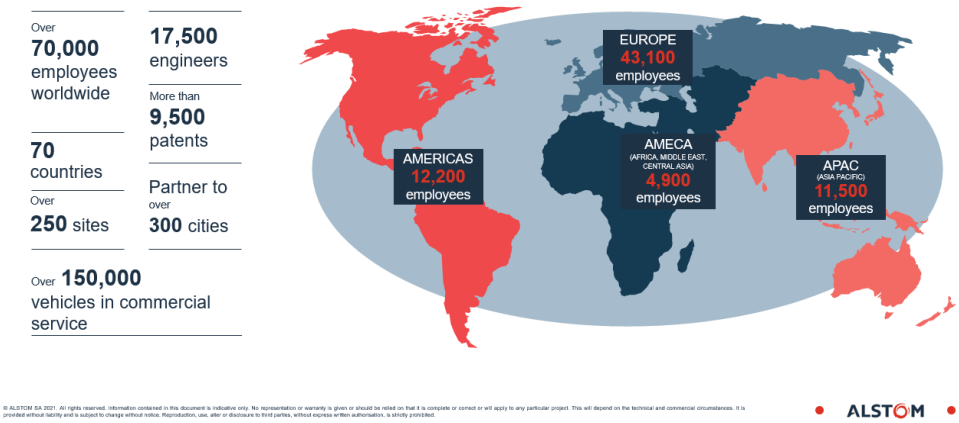


Figure 4: Alstom employee global presence

Challenges and motivation:

The Alstom use case is related to four main challenges that are structured into four user stories as follows.

Story A: Delta between product line features and customer needs

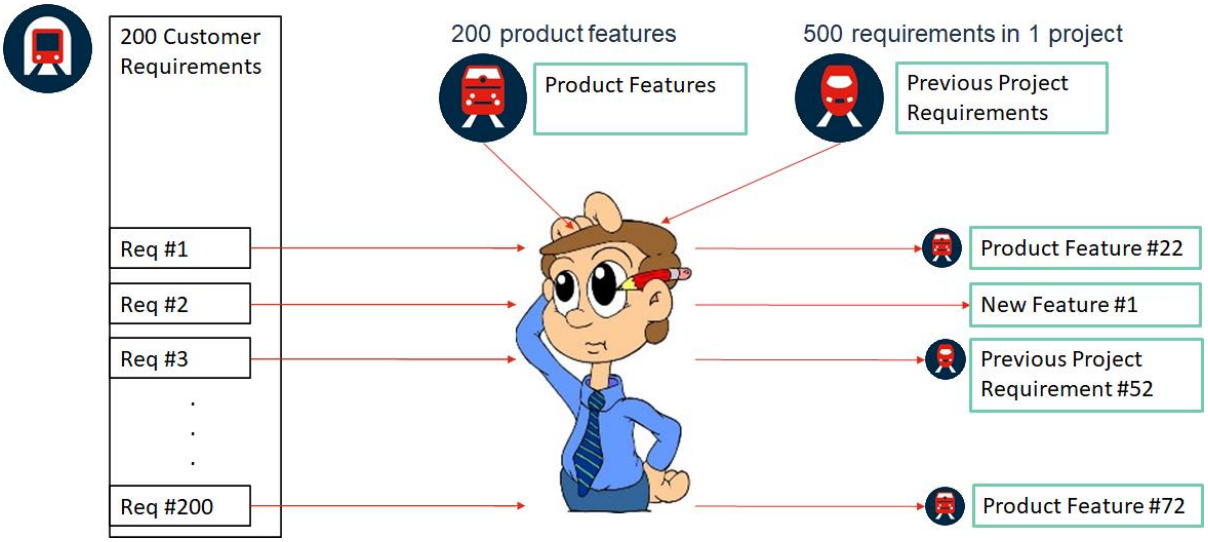


Figure 5: Customer requirements liked to produce features, past projects, or new features

Alstom in Västerås meet new customer requirements through modifying previous project solutions or modifying standardized “product” hardware and software solutions known. Controller firmware products provide an abstraction to support implementation of the train functions to fulfil the customer requirements. Standardized control algorithm products are adapted to suite hardware products been controlled. Both the hardware products and control algorithm products are usually modified to meet new customer requirements. Customer requirements are analyzed by advance engineers for correlation and differences with our standard product and past projects, sometimes within a very short timeframe due to bid submission and development deadlines. Bidding process is between 1 month to 12 months. Requirement analysis within projects takes 1 to 3 months. This process requires the system engineer to manually correlate all standard product features, past projects, and the current customer requirements. This process is extremely difficult, time consuming and error prone. Therefore, this user story requires innovative solutions that could aid the bidding and reuse process at Alstom.

Story B: Functional requirements quality and verifiability

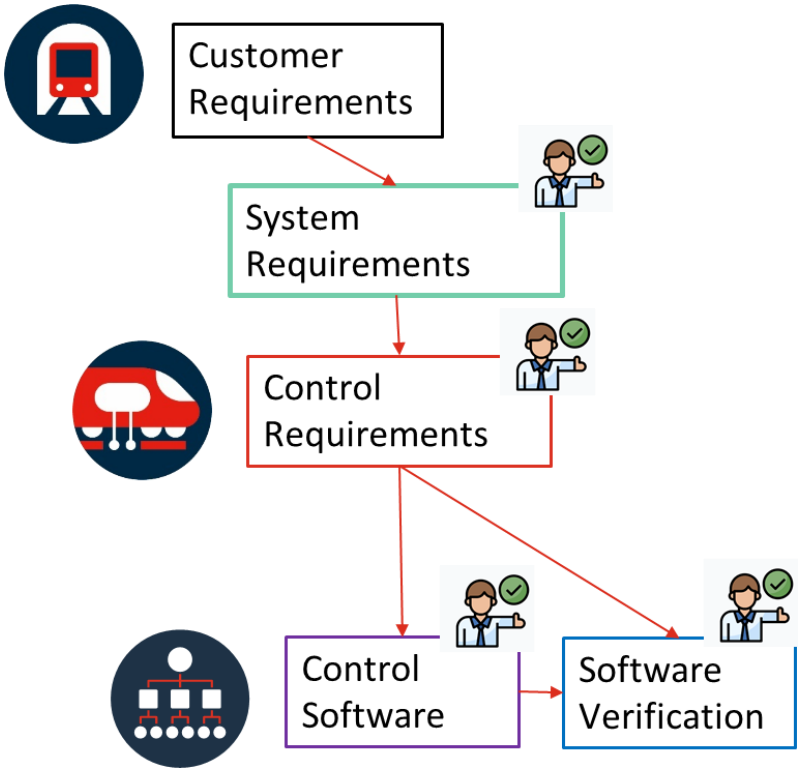


Figure 6: Customer requirements functionally verified manually

To ensure the customer requirements are verified, every customer requirement is broken down and linked to the relevant new and existing features. The features relevant to software control are broken down and linked to control requirements. All software requirements are linked to the relevant software implementation within the control software. To ensure the implemented software satisfies each requirement, at least one test is performed on the software, known as a functional test. The functional tests enable the verifier to demonstrate the software behaves as specified in the requirements. All links from the control requirements to the implemented software and functional tests must be traceable to satisfy European railway standard EN50657 up to safety integrity level 2 (SIL2). The functional tests are currently created and reviewed manually. Testers must review the requirements and create relevant test cases which will sufficiently test the control software. This use story attracts innovative solutions to enable semi-automated requirements quality evaluation and verification.

Story C: Code quality and Delta between manual and automatic test review, design review and code review

When designing software control solutions with a block diagram approach, the designs should be developed according to a specific set of rules and standards to ensure a higher reliability of the code generated and readability of the design. One approach used in Alstom is to use a functional block programming language using a design and coding standard enabling compliance to EN50128 or EN50657 up to safety integrity level 2 (SIL2). Depending on the designing tool used, the Alstom design standard is a combination of Alstom-specific designing guidelines and either:

1. Phoenix Contact Multiprog development environment based on IEC-1131
2. MATHWORKS Simulink

A combination of automatic and manual design and code reviews are performed to ensure the software artifacts have been developed against the Alstom software design and coding standard. Therefore, this user story will focus on bringing more automation to the code analysis and review process.

Story D: Incorporating product line updates into existing applications.

While the product line software evolves, the software artifacts evolve with each baseline and release which the projects using the product line need to consider. The evolution of these software artifacts between baselines could be either of the following:

- Clone – no changes to the software artifact.
- Change – changes performed on the software artifact.
- Removed – the software artifact is removed from the product.
- New – a software artifact is added to the product.

Within Alstom, the projects copy and own the product line at the start of their project. Once the project has copied the product line, they need to consider how to handle the product line to meet their project needs without affecting the product line. Within SmartDelta project, this user story aims at supporting the experts merge the desired product line changes into their project without negatively affecting the project application specific changes, that would be extremely beneficial

b. Link to SmartDelta Methodology

Alstom instantiates the overall SmartDelta methodology that spans across the in-house development life cycle for the customer delivery projects. Within the *Requirements Engineering*, the Alstom User Story A contributes to requirements extraction, allocation, similarity analysis, and quality. For *Incremental development,* requirements-driven reuse identification aids in the reuse of components to reduce the lead time of development and avoid redundant efforts. For *Quality Assurance*, User Story C contributes automated code review and automated test case generation from requirement models. Furthermore, User Story C also contributes quality assurance via automated design rule checking on the implementation models. Finally, User Story D focuses on the *product dimension* of the methodology by supporting product evolutions across product line variants. All four user stories contribute to visualization by visualizing output results. The four user stories are realised with various tools that are developed in SmartDelta project. Below, we briefly summarize each of the tools in relation with the four user stories.

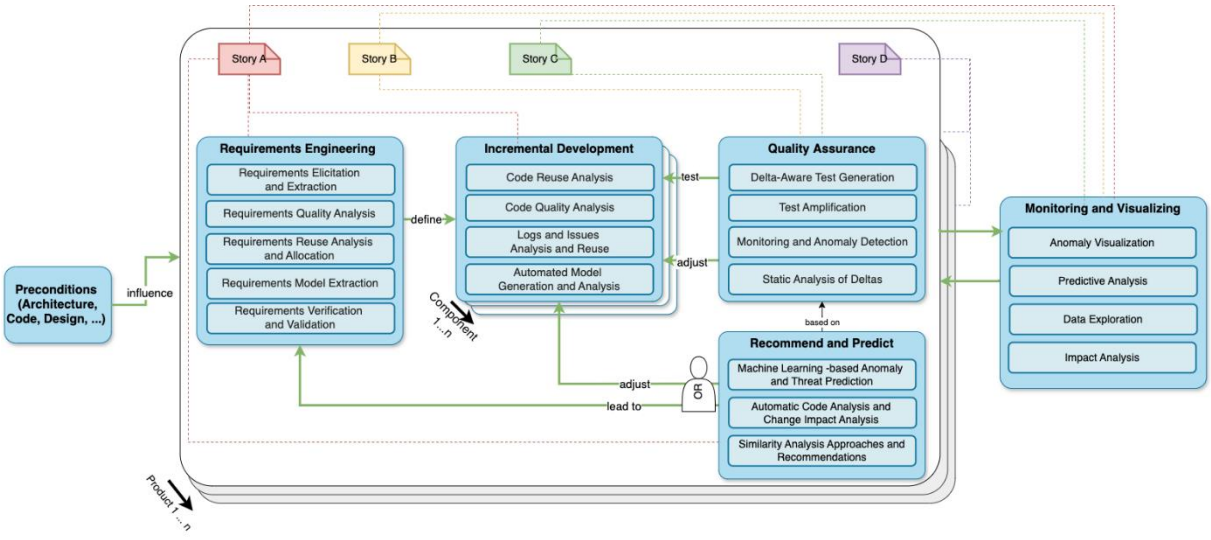


Figure 7: Mapping of the SmartDelta Methodology on the Alstom use-case (dotted lines shows mapping for Alstom).

c. Tools descriptions

To address the existing tool limitations described in the User Stories, the following solutions were proposed during the SmartDelta project. A description of each tool is included in the next section.

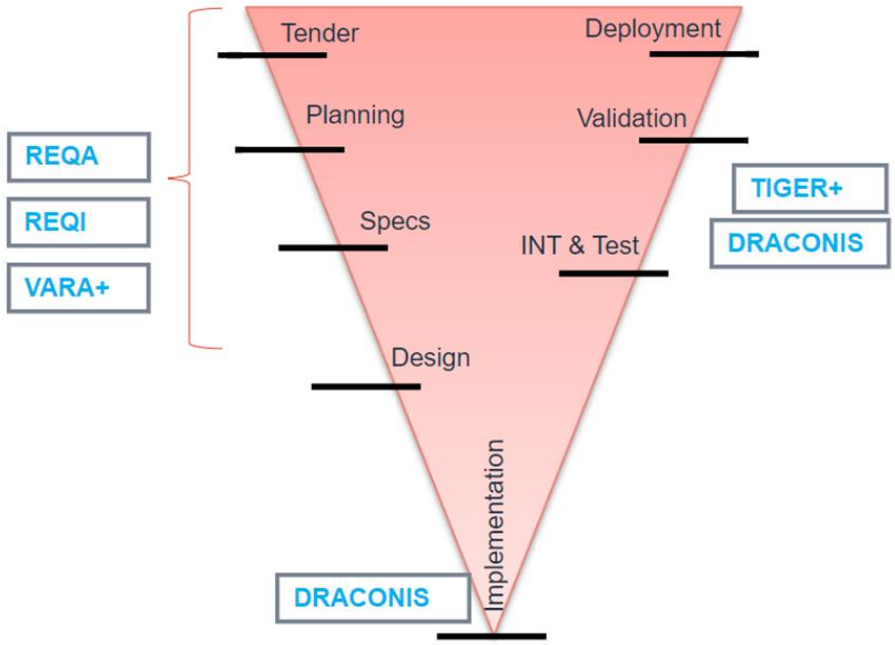


Figure 8: The SmartDelta Methodology for the Alstom use-case mapped to the V model

Key Tools and Their Roles in the Alstom Use-Case

REQ-I – a tool for customer needs identification for tender documents

ReqIdentifier (REQ-I) formulate the requirement identification problem as a binary text classification problem. It uses various state-of-the-art classifiers based on traditional machine learning, deep learning, and few-shot learning for requirements identification from large tender documents., the tool can process a text entry or a PDF document to extract text from it using Optical Character Recognition (OCR). Once, all the textual entries of the tender documents are available, a BERT language model-based classification pipeline is fine-tuned on the input text. In query mode, text or a PDF file can be given as input to the tool, and it outputs a PDF file with highlighted requirements.

REQA – a tool for allocation of requirements to teams for implementation

Once requirements are identified, they need to be allocated to various development and testing teams for implementation. The REQA tool combines traditional AI with deep learning to allocate requirements and generate supplementary information to support engineers in well-informed allocation. The REQA tool has two modules named Assigner and Augmenter:

The Assigner module uses large language models with statistical classification to recommend the allocation of the requirements to various teams that are likely to accept the allocation,

The Augmenter module uses lexical similarity-based clustering to generate cased based explanations to support the recommendations of Assigner and in turn, a well-informed allocation.

VARA+ – a tool for reuse identification

VARA is variability-aware requirements reuse analysis method which aims to automate product's assets reuse analysis and thus helps teams achieve quick and quality delivery of software systems. The tool uses state-of-the-art natural language processing and machine learning algorithms to predict existing product's assets that can be reused to realize the new customer requirements. In addition, VARA also compute various metrics (readability index, complexity and subjectivity etc.) on the quality of requirements that help in writing better requirements.
VARA takes all existing requirements and their links to software components implementing them, as input to fit/train a content-based recommender--- driven by clustering. In query mode (steps can be followed with blue arrows), unseen customer requirements are used as input to recommend reuse based on similarity with neighbouring existing requirements.

TIGER+ – a tool for test case generation and execution

TIGER uses the model-based testing concept to perform the concretization of abstract test cases and the generation of test scripts. It consists of three parts:

Modelling and Abstract Test Case Generator: GW (Graph Walker) takes as an input the model file in JSON/GraphML format and generates the abstract test cases by traversing through the model elements (i.e. states and transitions) based on a generator algorithm (such as random, quick random, Astar, etc.) and a stopping condition (such as edge coverage, vertex coverage, etc.).

Test Case Generator: The test case concretizer converts abstract test cases into concrete by mapping the logical signal names with their technical counterparts and corresponding values. Testers and developers use these logical names as initial names of the signals in the early phases of development. Later in the development, technical signal names became available that represent the actual signal names used by the SUT for its normal operations. Hence, the test case concretizer extracts the test data i.e. (variable names and their respective values) from the generated abstract test cases (available in a JSON file), extracts the required information about technical signal names from an XML file, and maps the logical signal names with technical signal names and their corresponding values based on defined mapping rules.

Test Script Generator: Once the abstract test cases are converted into concrete test cases, the test script generator generates the test script in C# language using the implementation details of the SUT (i.e. script format, libraries, and methods to be executed on the target test execution platform, SIL & HIL). The generated test script contains two types of steps for each test case, forcing the input signals and verifying the expected output signals, to validate the expected behaviour of the SUT.

DRACONIS – a framework for static analysis
DRACONIS is a static analysis framework. It is separated into three steps: Intermediate representation generation, analysis and reporting.
The intermediate representation is generated either directly through model transformations from the source models, or by converting the models to JSON, which is then parsed by the framework.
The analysis core supports analyses based on metrics or dataflow information. The tool supports design requirement checking by instantiating requirements as analysis rules (commonly called “checkers”) as a named combination of queries. Additional checks can be provided through configuration files, where the checker behaviour is defined using a subset of python. This allows multiple user configurations to be used, supporting for instance low-cost checks that may be run on every change to more extensive configurations which are part of the final validation work. After the analysis is performed, the tool stores the model instance and the analysis report in a database. In

cases where the model is then changed, a delta analysis is performed to recommend what analyses will need to be re-run.
The framework additionally supports upload of analysis results performed offline through JSON format, which will be added as *additional metrics*. Thus, some support for stand-alone tools such as https://github.com/jean-malm-mdh/fbd-complexity-tool exists.

DRACONIS can produce result outputs both in pure text-based format as well as a structured report, including a rendered image of the model. The web application provides functionality for reviewing and annotating the reports as well as viewing some statistics.

The usage of DRACONIS allows the automated generation of reports based on some existing design rules, which in turn accelerates starting the review process, and by automating the tedious parts manual validation effort can then be focused on cases where a human is most needed.

Implementation Details

- Python-based analysis backend. Developed against Python 3.9
- Adapters: PLC adapter uses ANTLR4 parser framework, Simulink adapter generates into JSON format.
- Django Web Application exposing report review interface and API functionality.
- Command Line Interface for batch uploading and generating textual reports.

The implementation of the tool can be found at https://github.com/jean-malm-mdh/draconis

d. Visualization

DRACONIS. Most of DRACONIS' visualisation facilities are accessed through the web application. Figure 9 shows the report view for a block model. Checks are first grouped by whether they pass or fail, to ensure the user can quickly get an overview of the status. Users may then add review the different reports by adding comments. Additionally, a graphical rendering of the model is shown to the right of the reports, giving the developer contextual information such as comment blocks. In Figure 10 we see the remaining information: Variable information, metrics, as well as dependency chains of endpoint variables.

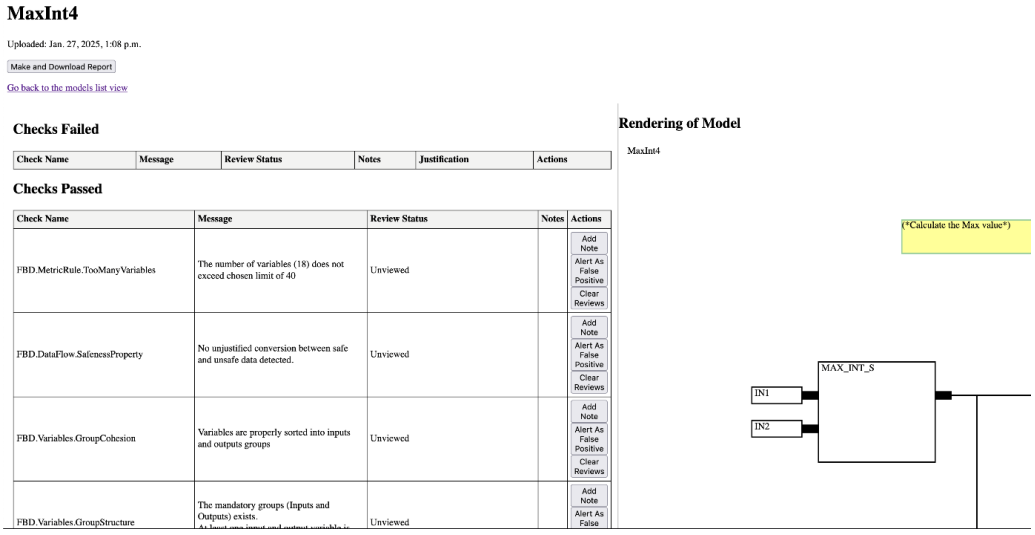


Figure 9: Top half of DRACONIS' model report view. To the left is the report side, and a rendering of the model can be seen to the right (continuing off-screen). The user may the buttons in the action column to leave comments or classify reports.

Variable Info

Variable name	Variable Type	Data Type	Initialized Value	Description
IN1	InputVar	SAFEINT	UNINIT	Value in 1
IN2	InputVar	SAFEINT	UNINIT	Value in 2
IN3	InputVar	SAFEINT	UNINIT	Value in 3
IN4	InputVar	SAFEINT	UNINIT	Value in 4
OUT1	OutputVar	SAFEINT	UNINIT	Value OUT 1 Max Value
OUT2	OutputVar	SAFEINT	UNINIT	Value OUT 2 Second Max Value
C_Invalid	InternalVar	SAFEINT	-32767	Constant: Invalid
C_Id1	InternalVar	SAFEUSINT	SAFEUSINT#1	Constant: Index 1
C_Id2	InternalVar	SAFEUSINT	SAFEUSINT#2	Constant: Index 2
C_Id3	InternalVar	SAFEUSINT	SAFEUSINT#3	Constant: Index 3
C_Id4	InternalVar	SAFEUSINT	SAFEUSINT#4	Constant: Index 4
Max_Id	InternalVar	SAFEUSINT	UNINIT	Id of the Maximum Value
MAX_INT_S_1	InternalVar	CUSTOM_FBD	UNINIT	None
MAX_INT_S_2	InternalVar	CUSTOM_FBD	UNINIT	None
MAX_INT_S_3	InternalVar	CUSTOM_FBD	UNINIT	None
MAX_INT_S_4	InternalVar	CUSTOM_FBD	UNINIT	None
MAX_INT_S_5	InternalVar	CUSTOM_FBD	UNINIT	None
MAX_INT_S_6	InternalVar	CUSTOM_FBD	UNINIT	None

Core Metrics

Name	Value
NrOfVariables	18
NrOfFuncBlocks	20
NrInputVariables	4
NrOutputVariables	2
VariableTypeComplexity	132
IsPotentiallyImpure	True

Additional Metrics

Name	Value

Dependency Analysis

Network Outputs	Potentially dependent Input(s)
Max_Id	['IN1', 'IN2', 'IN3', 'IN4', 'IN4', 'C_Id4', 'IN1', 'IN2', 'IN3', 'IN3', 'C_Id3', 'IN1', 'IN2', 'IN2', 'C_Id2', 'C_Id1']
OUT2	['Max_Id', 'C_Id1', 'C_Invalid', 'IN1', 'Max_Id', 'C_Id2', 'C_Invalid', 'IN2', 'Max_Id', 'C_Id3', 'C_Invalid', 'IN3', 'Max_Id', 'C_Id4', 'C_Invalid', 'IN4']
OUT1	['IN1', 'IN2', 'IN3', 'IN4']

Figure 10: Bottom half of DRACONIS' model report view displaying data in tabular form. The full set of variables is sorted and printed. Core and additional metrics are displayed separately. The dependency analysis shows the dependencies from function block network endpoints (internal feedback variables, and outputs) backwards.

Users may also download an excel report, which contains a summary of checker reports, their feedback, the metrics and the rendering. This allows the analysis results to be disseminated in a document-driven way-of-working process.

DRACONIS can also perform a diff analysis of two models. If differences are detected, a summary of the differences is written out in plain text, and a visual comparison is highlighted by overlaying the different model renderings, where a red colour marking the graphical differences detected. Figure 11 and 12 show these two views for a small illustrative example.

Diff Report for the Selected Programs

Summary

The following variables have changed some property between the versions:
Var(UINT Result_Even: OutputVar = 0; Description: 'Result if the input is an even number') ->
Var(SAFEUINT Result_Even: OutputVar = 0; Description: 'Result if the input is an even number')

Expression in constant block UINT#2 has changed to SAFEUINT#2 - rerun IO tests

Figure 11: Textual summary of a diff analysis between two variants of the same base model, where the newer model has been rewritten to use the SAFE variant of unsigned int datatypes.

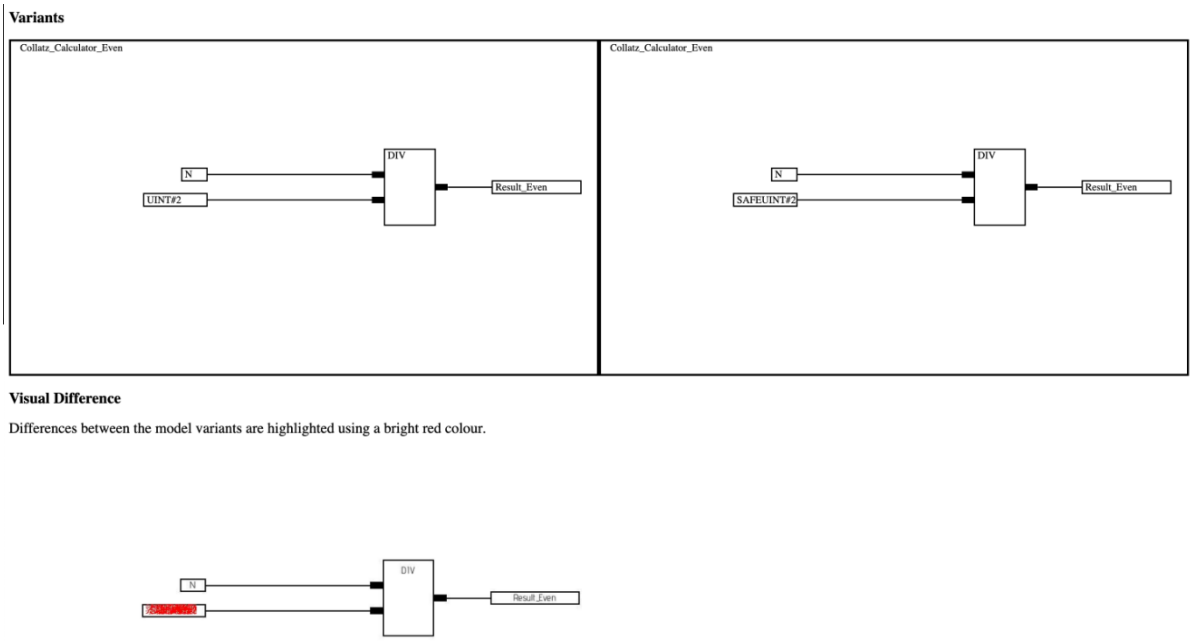


Figure 12: The graphical differences highlighting the changed input port properties. The original variants are presented alongside the diff to allow the user to investigate the change in the same view.

TIGER+. The TIGER framework uses visualizations that display the finite state machine model (from Graphwalker as shown in Figure 13) and the generated test. It highlights the states, transitions, and guard conditions derived from the system requirements, showing traceability from requirements to test elements.

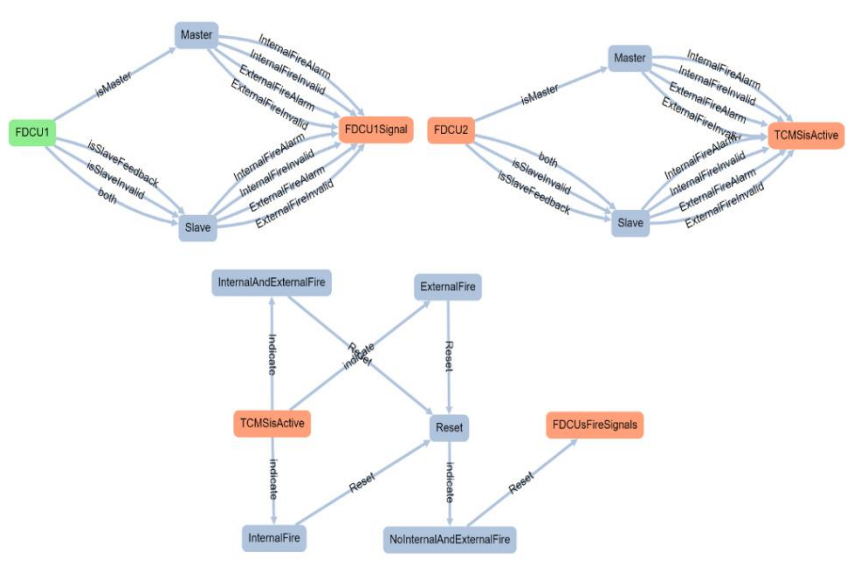


Figure 13. An example test model shown in Graphwalker within the TIGER framework.

Each model element (node or edge) is linked to the corresponding requirement so engineers can track coverage and confirm that all essential behaviour is represented. The visualization also shows the status of generation and optimization runs.

```
{
  "modelName": "VarA",
  "data": [
    { "varD": "true" },
    { "varC": "true" },
    { "varB": "true" },
    { "varA": "true" }
  ],
  "currentElementID": "1eacd4d5-3d6c-4624-92f4-65bc5e6d9a5c",
  "currentElementName": "A",
  "properties": [
    { "x": 221.04553986268414 },
    { "y": 326.0896089137364 }
  ]
} {
  "modelName": "VarA",
  "data": [
    { "varD": "true" },
    { "varC": "true" },
    { "varB": "true" },
    { "varA": "true" }
  ],
  "currentElementID": "19e19385-9a02-45f7-b780-162e3c53f7a5",
  "currentElementName": "No",
  "actions": [ { "Action": "varA=false;" } ],
  "properties": []
} {
```


Figure 14. A representation of an abstract test case and its actions.

(i)Dashboard Solution

DRACONIS. As it is primarily an analysis framework, DRACONIS' internal dashboard is focused on showing an overview of the ongoing review status and is built into the web application. This allows stakeholders to get an overview of the code quality on a checker level and the review effort. These views are shown in Figures 15. and 16 respectively.

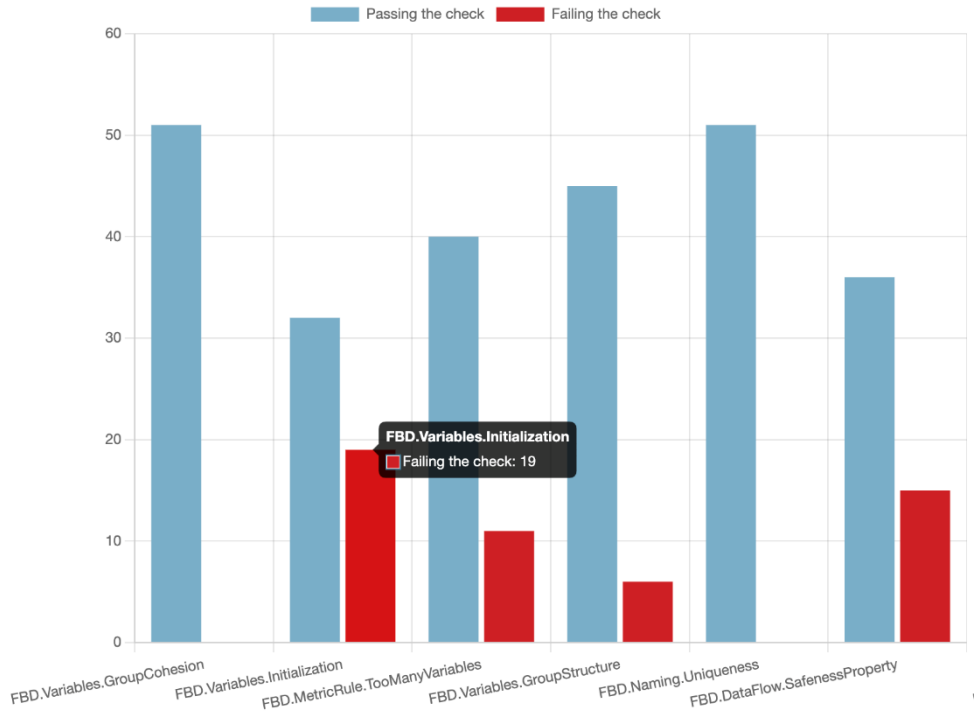


Figure 15: A graph of the number of models that have passed and failed the specific checks respectively. Basic interactivity for filtering the data based on labels exists.

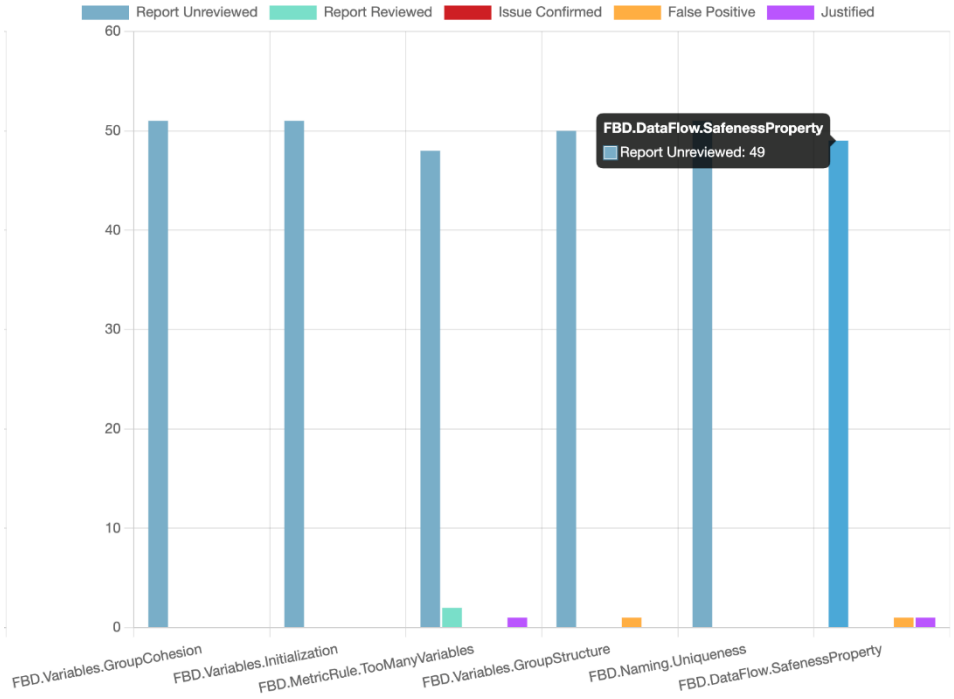


Figure 16: Graph showing the number of reports that have been attended to by a user. All reports will start with the status unreviewed, and then get other statuses upon being interacted with.

Visualisation requirements

TIGER+.

In industrial practice, especially for safety-critical systems like the TCMS system, these visualizations in TIGER+ ensure testers, developers, and validators understand which requirement is tested and when it transitions in the model. This is particularly useful during audits or certification steps, where visual confirmation of requirement coverage helps verify compliance with standards such as EN 50128 or EN 50657.

e. Use-Case evaluation Setup

Each tool included in the Alstom use case was evaluated within SmartDelta. The tools are evaluated against technical KPIs.

REQ-I.

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 17: Considered data from REQ-I evaluation

As shown in Figure 17, 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.

REQA.

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 18, 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.

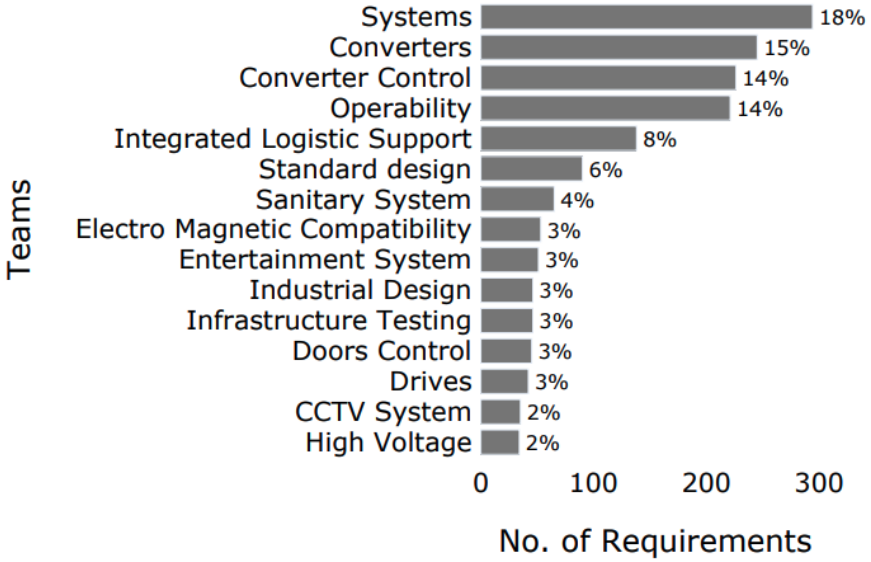


Figure 18: Considered data for REQA evaluation

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.

VARA

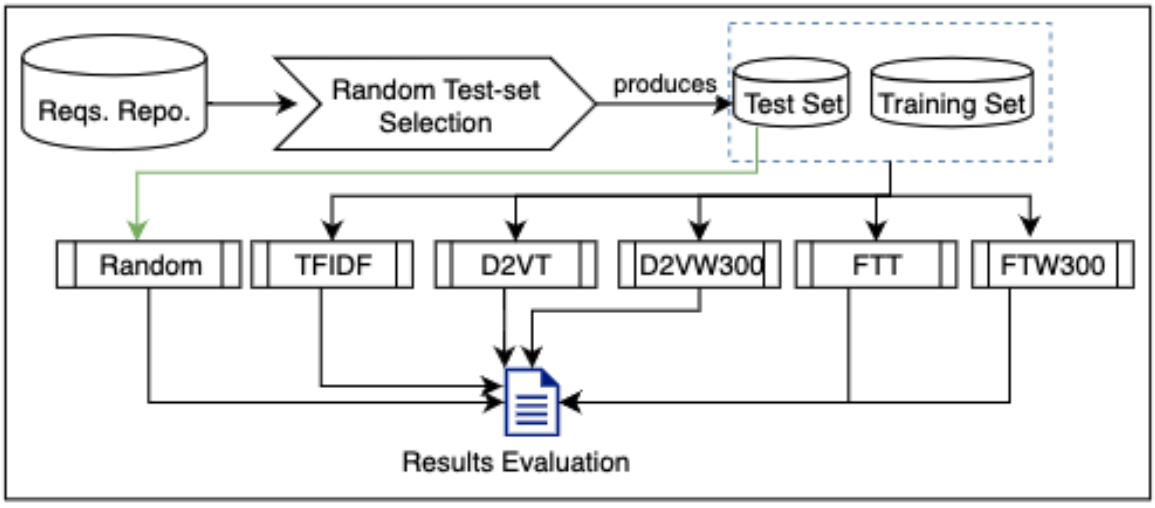


Figure 19: VARA evaluation procedure

Old Evaluation: As shown in Figure 19, the evaluation process considered a requirements repository originating from two projects and containing 188 high-level requirements linked to software components implementing them. The data was randomly split to consider 75% of the requirements for training. Clustering-based Content-based recommenders are then trained on the training set and evaluated on the test set. The considered recommendation pipelines consider a random, the VARA approach (TF-

IDF), Doc2Vec trained (D2VT), Doc2Vec pre-trained (D2VW300), FastText trained (FTT), and FastText pre-trained (FTW300).

New Evaluation: After the old evaluation, resulting in around 78% average accuracy for the VARA approach, the tool is also exploited at another department at Alstom. In this case, the tool was used for in-project reuse identification. The evaluation considers two projects with varying numbers of requirements among which 80% of requirements were used for training and 20% for testing and validation.

We used the standard metric accuracy (A) and exact match ratio (E) for the evaluation of our pipelines. A recommendation is correct if the recommendations generated by the pipeline contains the ground truth. In our case, accuracy is calculated as the ratio between the total number of correct recommendations and total instances in the test set. In addition, we use a stricter evaluation metric (i.e., exact match percentage). This is calculated using the ratio between the number of exactly correct recommendations (where the ground truth is ranked on the top of the list of recommendations) and the total number of instances in the test set.

DRACONIS. For evaluating the speedup of the review process, the tool was applied on a set of real function block diagrams provided by Alstom, with the full analysis and upload time sampled 5 times per model. This was evaluated against the current way-of-working, where a developer would produce the same information from the development tool.

TIGER+. The evaluation framework for TIGER+ follows a model-based testing process integrated with the GraphWalker tool for test case generation and execution. The process starts with defining system requirements. These requirements are translated into a model representing the system under test. Abstract test cases are generated from the model and optimized by TIGER+ to remove redundancies while preserving fault detection effectiveness. The requirements for the evaluation setup are as follows: generate MC/DC-adequate test suites while ensuring efficient coverage of system requirements, achieve high requirement coverage with a minimized number of test cases, and maintain or improve the fault detection rate after test suite optimization.

f. Evaluation results

REQ-I. Results from the evaluation on the Alstom use-case show that the tool could identify requirements in large documents with an average F1 score of 0.82%. Our results also confirm that few-shot classifiers can achieve comparable results with an average F1 score of 0.76 on significantly lower samples, i.e., only 20% of the data.

REQA. Results from the evaluation show that REQA can allocate the requirements to different teams with a 76% F1 score when considering requirements allocation to the most frequent teams. Information augmentation provides potentially useful indications in 76% of the cases.

VARA. Evaluation of VARA+ shows that the tool can recommend reuse with an average accuracy of around 82% and can reduce the lead time of the propulsion software system. In addition, the qualitative evaluation also shows that the recommendations produced by the tool are valuable and insightful.

TIGER+. TIGER+ improved test execution efficiency in the Alstom use case by reducing the size of the model-based test suite while maintaining high fault detection effectiveness. TIGER+ identified and removed redundant test cases, reducing the test suite size by approximately 85% to 92%. The optimized test suites maintained a fault detection rate of 95% to 100%, comparable to manually created test suites, while lowering execution costs.

DRACONIS. With respect to the review process speedup, the evaluation was performed on a dataset of 51 function block diagrams of varying complexity, with a total number of over 1100 function blocks. Using the command line interface, the entire analysis and upload to the web app process took 17.89s (sampled over 5 uploads) with a single user action. Once uploaded, generating an excel report out of the results and rendered model can be done in four clicks, from the starting point of the application.

In comparison, for the manual process we observed generation times of between 5 and 10 minutes per model, depending on the complexity and size of the model. This included performing the same reviews, computing/finding the metrics and dependency information and putting the information into a report form.

Table 3: KPIs overview table

Requirements	Tool	Solution partner	KPI Definition	KPI Base Values	KPI Target Values	KPI achieved Values
UC1.FR1	TIGER+	MDU	The success is to reduce the cost of translating input requirements into architecture/Design and functional tests. In Alstom today, majority of this process is manual. This solution shall conform to safety standards EN50657&EN50128 if a human is not kept in the loop.	100%	40%	70%
UC1.FR2	TIGER+	MDU	The success is to reduce the cost of verifying a software package through a smart verification tool/processes ensuring redundant testing is removed. Today, a full set of tests are performed on the software package, no matter the number or types of changes. This solution shall conform to safety standards EN50657&EN50128 if a human is not kept in the loop.	100%	80%	72%
UC1.FR3	DRACONIS	MDU	The success is to reduce the cost of identifying the similarities and differences between two software packages. The software packages are usually very similar. Key metrics on the types of differences between software packages is very important. The current process today is to manually review the changes between two software packages and record if the changes affects interfaces, functionality, documentation or testing. This solution shall conform to safety standards EN50657&EN50128 if a human is not kept in the loop.	functionality identical models 10 minutes per model. identical models & code takes 1 minute per package	functionality identical models 20 seconds per model. identical models & code takes 1 minute per package	functionality identical models less than 5 seconds per model.

UC1.FR4	VARA+ REQA REQI	RISE	The success is to reduce the cost of identifying the similarities and differences between two requirement sets. Key metrics on the types of differences between two sets of requirements is very important. Today, a manual review of the two requirement sets is performed. This solution shall conform to safety standards EN50657&EN50128 if a human is not kept in the loop.	100%	20%	20%
UC1.FR5	DRACONIS	MDU	The success is to reduce the cost of identifying the gaps in the conformance and compliance when verifying software packages. Key metrics on the types of gaps is very beneficial. Today, a manual review of the activities performed vs the activities requested is performed highlighting the gaps in different tools, e.g. excel. This solution shall conform to safety standards EN50657&EN50128 if a human is not kept in the loop.	20 seconds to 20 hours depending on task.	a minute to clearly see which compliance gap exists in the code/models.	less than 1 second
UC1.FR6	DRACONIS	MDU	The success is to process software packages which can generate around 1 million lines of code.	N/A, waiting for tool.	1 million lines of code and/or 300 models	300+ models
UC1.FR7	All	All	The success is to reduce the time (hence cost) a defect is discovered in the whole software design process. Defects can be created through all development steps from requirement mistakes to coding errors. Alstom detects different defects are different stages of the development process, but generally in the design review or testing steps. This solution shall conform to safety standards EN50657&EN50128 if a human is not kept in the loop.	100%	50%	33%

C1.FR1: The KPI measures the reduction of the cost associated with the input requirements translation to architecture models and functional tests. We have measured 30% reduction of cost, which was mainly contributed by the functional tests generation from requirements. However, the former step of translating requirements to architectural models has not been fully achieved, which potentially could contribute to further cost reduction to achieve the targeted 60% reduction. The measurement is carried out by selecting a set of requirements which has been manually translated into models and functional tests by experienced engineers in contrast with the activities/steps carried out by the tool to generate the functional tests.

UC1.FR2: The KPI measures the reduction of the cost associated with the verification of software package. We have achieved 28% reduction in cost thanks to TIGER+ tool. Similar to the UC1.FR1, the measurement is carried out by selecting a set of requirements, for which the experienced testers performed the test case design, implementing test case scripts and executing the test cases to verify the software package. The cost (measured in terms of time) is compared with to the time taken by the TIGER+ tool to verify and generate test reports for the same data set.

UC1.FR3: The KPI measures the reduction of cost associated with identifying the similarities between two software packages mainly to identify the differences between two models. This delta identification is essential to enable the reuse of software with ease in the projects. Moreover, this supports the identification of the change impact for example in iterative development a clear visualisation of the delta (such as interfaces, functionality etc.) between two versions of same models results efficient planning of change impact. We have measured that this delta is highlighted and can be followed with DRACONIS tool in less than 5 seconds for the diverse models of complexity.

UC1.FR4: The KPI measures the reduction of cost associated with identifying the similarities between two requirements set. For measurement, we have selected a set of requirements which has been estimated by experience manager for identification of similar requirements in the existing project base. The VARA+ tool chain is applied on the same set of requirements which provides the results in few mins. However, these results need to be manually reviewed (human in the loop) by experience manager for conformance. Considering both the review time and execution time of VARA+, we have achieved 80% cost reduction in contrast with the manual approach.

UC1.FR5: The KPI measures the reduction of cost associated with identification of compliance gaps in the software package mainly related to code review. For measurement, we have selected a sample of ten models (of diverse complexity) which are manually reviewed by the experienced verifier and recorded the cost associated to this activity. The same models are then fed to the DRACONIS tool, where we have measured that the results can be visualised in less than 1 second for each model.

UC1.FR6: The KPI measures the scalability of the solution. We have measured that the DRACONIS tool can practically process and generate the code review results for 300+ models with a single input.

UC1.FR7: The KPI measures the overall cost reduction in identifying the defects spanning across different phases of the software development. To measure this, we have considered the other KPIs which are addressing the reduction in three stages of the over development cycle i.e., requirements engineering, implementation and verification respectively. The reduction measured on each of these stages is then summed – we have measured an overall reduction of 67%.

g. Recommendation for industry adoption

Even for experienced engineers, it is difficult and time-consuming to manually identify the specific product feature or previous products which complies with new customer requirements, since each standard product has over a hundred features and Alstom has over a thousand solutions within our past projects linked to 100 to 10000 requirements in each project (as highlighted in Story A and D). Tools like REQA, REQ-I and VARA, can analyse and bridge the gap between existing

product line features and customer needs. The evaluation results show that these tools can provide, in many cases, a good requirements identification and mapping that can aid the engineers in their work.

When a project starts and it takes a product or similar project solutions as a base for development, they also inherit all the associated test cases and verification results. Since each project or product iteration has a unique set of customer requirements, the links from customer requirements to implementation need to be created and verification activities need to be performed again (as highlighted in Story B). Currently, these tests are created and reviewed manually. Tools like TIGER+ can support with the test-case generation and execution and the evaluation results show that the tool can speed-up the testing activities by reducing the number of tests while maintaining a high fault detection rate. To be adopted in the industry, the evaluation suggests that organizations should ensure that the requirements are well-structured and traceable, as the effectiveness of TIGER+ relies heavily on the quality of input requirements and models. Investing in training for engineering teams to create accurate models and properly utilize the Gherkin-like DSL will improve the use of TIGER+. It is recommended that companies validate the generated FSM models for syntactic and semantic correctness to prevent the propagation of modeling errors. Regular reviews of requirement traceability and coverage metrics through the TIGER+ visualizations will help monitor test suite effectiveness. While adopting TIGER+, companies should avoid relying solely on automated test suite optimization without proper manual review. Although TIGER+ significantly reduces test suite size, some context-specific test cases may be necessary to cover edge cases.

The control software is developed according to a specific set of rules and standards to ensure a high readability of the design and reliability of the generated code. To check that the block diagrams used to produce the software are developed according to the coding standard, a set of design and code reviews are performed (as highlighted in Story C). While many of these check require manual review, tools like DRACONIS have supported the automatic checking through static analysis of different design requirements.

Overall, the constellation of tools included in the SmartDelta methodology effectively addresses the company’s needs as outlined in the use case. However, further evaluation is required before their industrial adoption.

4. Use-Case 2 from Akkodis

a. Use-Case Description

Challenges and Motivation

Starting Point: The Challenges of Developing the EvaCharge Product

The EvaCharge product operates in a highly competitive and fast-paced environment, specifically within the rapidly evolving market for electric vehicle (EV) charging. EvaCharge is an ISO 15118-compliant controller designed for both electric vehicles and charging stations (see figure). Its development faces unique challenges driven by the complexity and dynamism of the EV charging ecosystem. Since its launch in 2014, EvaCharge has established a strong foothold in the market, with over 170 customers and more than 35,000 installations across 35 countries on 5 continents. This extensive adoption underscores the product's versatility and reliability in various charging scenarios, from individual users to large-scale commercial applications (see figure 20)

Figure 20: EvaCharge worldwide footprint

The Evolving Market of Electric Vehicle Charging

The EV charging market is still relatively young, but it is growing and transforming at an extraordinary pace. This evolution is fuelled by advancements in technology and the increasing adoption of electric vehicles across various sectors, including passenger cars, buses, trucks, and even specialized vehicles like agricultural machinery. Several key trends are shaping the market:

1. Higher Power Rates:
   The industry is moving towards ultra-fast charging solutions capable of delivering higher power rates. This is critical to reducing charging times and enhancing the overall user experience, particularly for long-distance travellers and commercial fleet operators.
2. Simplified Authentication and Billing:
   Modern EV charging systems prioritize seamless user experiences, offering easy authentication methods such as Plug & Charge (enabled by ISO 15118) and streamlined billing processes. These innovations make charging as convenient as traditional refuelling.
3. Dynamic Scheduling and Pricing:
   Smart charging infrastructure enables dynamic scheduling and pricing models. This not only maximizes resource utilization but also provides users with flexible options tailored to their specific needs and energy availability.
4. Convenience and Flexibility:
   The market demands charging solutions that are convenient and adaptable to various user scenarios. Whether it's home charging, public fast-charging stations, or depot charging for commercial fleets, the infrastructure must accommodate diverse needs.
5. Smarter Charging Planning:
   Advanced algorithms and AI are increasingly employed to optimize charging schedules. This ensures vehicles are charged efficiently while minimizing stress on the electrical grid and leveraging periods of low electricity costs.
6. Grid-Friendly Charging:
   As the adoption of EVs increases, their collective impact on the grid becomes significant. Charging solutions are now designed to support grid stability through load balancing, vehicle-to-grid (V2G) technologies, and integration with renewable energy sources.
7. Diverse Use Cases:
   The market is expanding beyond passenger vehicles. Solutions are being developed for

specialized applications, such as pantograph charging for buses and trucks, wireless charging for convenience, and even solar-integrated systems for off-grid scenarios.

8. High Connectivity and Service Integration:
   Modern charging solutions emphasize connectivity, allowing integration with a wide range of services. These include navigation systems, fleet management software, energy management systems, and even smart home devices. This connectivity supports smarter, more autonomous charging behaviour.

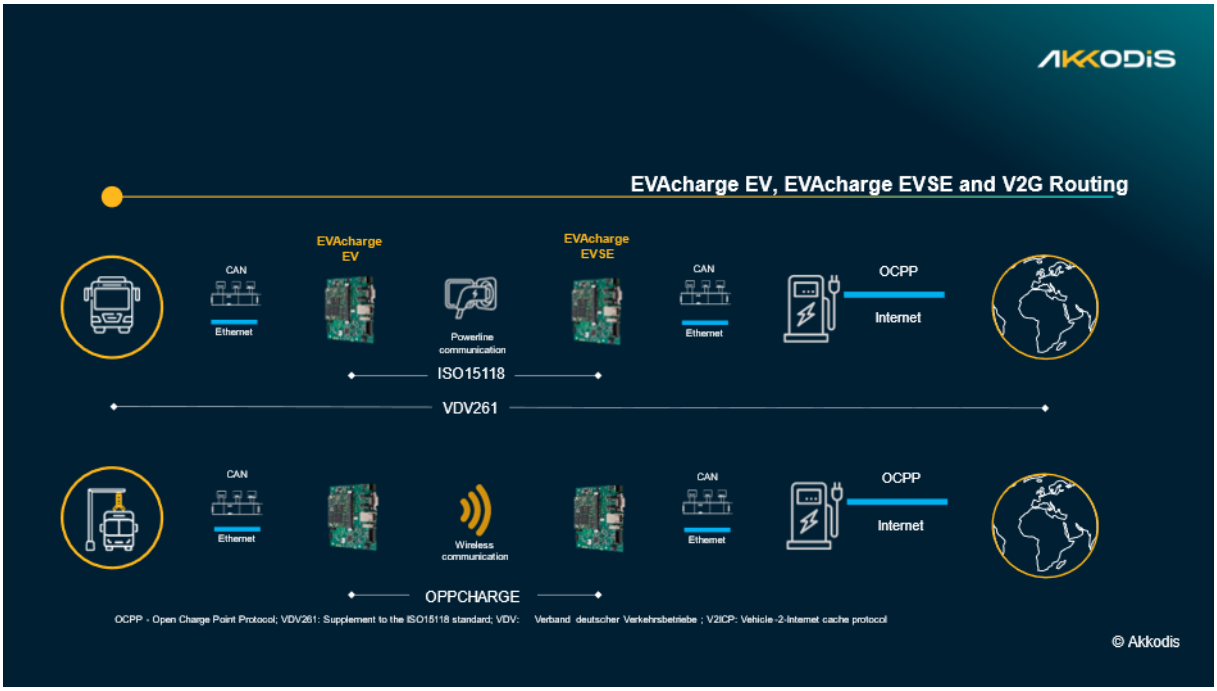


Figure 21: EvaCharge ecosystem

The figure 21 illustrates the evolving ecosystem of the EvaCharge product, highlighting new use cases and the integration of additional components and standards. As the ISO 15118 standard evolves beyond its initial implementation, new features like Vehicle-to-Grid (V2G) routing, advanced charging protocols, and enhanced communication methods such as OppCharge and VDV261 are introduced. These developments build on older standards like DIN 70121, increasing the overall system complexity.
Key elements shown include:

- EvaCharge EV (onboard vehicle controller) and EvaCharge EVSE (charging station controller), both supporting powerline and wireless communication.
- Compatibility with multiple communication layers, including CAN and Ethernet, for seamless interaction between vehicles, chargers, and the grid.
- Integration with OCPP (Open Charge Point Protocol) for global connectivity and remote management.
- Advanced use cases like pantograph charging for buses, underlining the system's versatility across different vehicle types and applications.

This interconnected setup demonstrates the growing sophistication of EV charging infrastructure, aiming to provide smarter, more efficient, and grid-friendly charging solutions.

However, the challenges EvaCharge faces are not confined to the EV charging domain alone. Broader difficulties stem from the general software development field, which is currently grappling with a well-documented shortage of skilled software developers. This industry-wide constraint

exacerbates the task of developing high-quality, feature-rich software under tight deadlines. Combined with the ever-increasing demand for faster development cycles and more sophisticated product capabilities, these factors place significant pressure on development teams to do more with less.

This dynamic environment presents several significant challenges:

1. Limited Software Development Resources:
   A critical constraint in our context is the shortage of skilled software developers. Given the high demand for rapid product innovation, this limitation hampers our ability to scale efficiently and meet market expectations.
2. Accelerated Development Timelines:
   To remain competitive in this disruptive market, new features and product variants must be developed and released at an increasingly rapid pace. Traditional development methods struggle to meet these demanding timelines.
3. High Variability Across Product Lines:
   The EvaCharge product portfolio includes numerous variants to cater to diverse customer segments and comply with different regulatory environments. Managing this level of complexity within constrained timeframes adds significant pressure to our development processes.
4. Necessity for Agile Development Practices:
   In such a dynamic market, agility is essential. We must quickly adapt to changing requirements and market conditions. However, our existing practices do not fully exploit the potential of agile methodologies to streamline development.

Motivation: The Need for an Advanced Development Framework

These challenges underscore the need for innovative tools and methodologies that can optimize resource utilization, accelerate development cycles, and manage product variability effectively. Addressing these goals is critical to maintaining our competitiveness and ensuring efficient scaling of our development efforts.

Goals: Applying Advanced Tools in a Real-World Use Case

To tackle these challenges, we are working with research partners who are developing advanced tools based on a corpus-based development approach with AI support. Our role is to provide a real-world use case that involves defining specific requirements, supplying relevant data, and engaging in productive discussions to ensure the tools are tailored to address our needs. The core principles of corpus-based development include:

1. Comprehensive Storage of Software Artifacts:
   The approach emphasizes organizing and storing all development artifacts—such as source code, requirements, specifications, and models—in a structured and accessible format within Git repositories. This practice ensures that each artifact is readily available for reference and reuse.
2. Avoidance of Binary Files:
   Binary files are avoided wherever possible, favoring text-based formats instead. This enables efficient version control, making it easier to track, diff, and merge changes across the development lifecycle.
3. Direct Integration of Requirements and Specifications:
   Requirements and specifications, often derived from external standards like ISO 15118, are directly incorporated into the repositories alongside code. This integration enhances traceability and enables a consistent view of the project’s scope and details.

The corpus used in our EvaCharge use case comprises:

- Requirements and Specifications sourced from the ISO 15118 standard, which governs electric vehicle charging communication protocols.
- C++ Source Code implementing the core functionality of EvaCharge.
- UML State Machine Models that enable simulation and testing of complex system behaviors.

Through this structured corpus, developers gain efficient access to information, enabling the AI tools to deliver meaningful recommendations for reuse, code generation, and impact analysis.

Applying AI to Enhance Corpus-Based Development

The tools developed by our research partners target specific goals that address the challenges in our use case:

1. Efficient Reuse of Existing Software Artifacts:
   The tools create a repository of reusable software artifacts. Using AI, they can identify and recommend existing components, reducing redundant development efforts and accelerating project timelines.
2. Automated Generation of Software Artifacts from Requirements:
   The tools leverage AI to transform high-level requirements into functional software components, streamlining early development phases and helping ensure alignment between requirements and implementation.
3. Streamlined Change Management:
   The tools support seamless planning and integration of changes by automatically identifying affected components within the repositories, providing optimized strategies for their modification.
4. Incorporation of Software Metrics in Decision-Making:
   The tools use metrics such as code complexity, test coverage, and maintainability to guide changes in ways that maintain or improve system quality.
5. Visualization of Changes at the Model Level:
   Through visual representations, the tools enable stakeholders to grasp the scope and implications of modifications quickly, which enhances communication within agile teams.

Conclusion

In this project, our role is to provide a comprehensive use case for testing and refining these advanced tools. Through the application of corpus-based development and AI-powered capabilities, we aim to evaluate the tools' effectiveness in solving real-world challenges associated with the EvaCharge product. This collaboration not only promises to improve our own development practices but also contributes valuable insights that will advance the ongoing development and refinement of these cutting-edge tools.

b. Link to SmartDelta Methodology

The Akkodis use case and its associated developments exemplify the methodology outlined in the SmartDelta framework, specifically within the context of incremental development, as illustrated in the Figure 22:

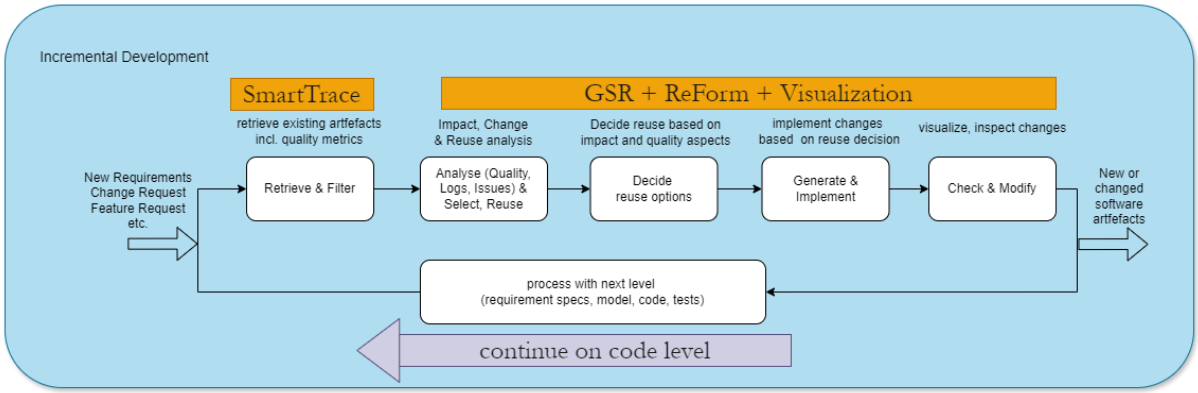


Figure 12: Akkodis Use Case: Mapping to Methodology

Overview of the Incremental Development Approach

The incremental development approach implemented in the SmartDelta framework utilizes tools and methodologies to enhance software development processes. This approach is designed to help developers manage the increasing complexity of modern software systems, especially in dynamic and rapidly evolving domains such as electric vehicle charging.

Initial Query and Artefact Retrieval

The process begins with a manually written query. This query—comprising a topic, keywords, or a full sentence derived from a requirement or specification—serves as the starting point for retrieving relevant software artefacts from the project corpus. These artefacts include:

- **Requirements and Specifications**
- **Models** (e.g., UML state machine models)
- **Source Code**

The **SmartTrace** tool acts as the retriever, searching the structured project corpus (e.g., Git repositories) for artefacts that align with the query. SmartTrace returns artefacts deemed relevant to the new requirement or feature under development.

Iterative Review and Refinement

Once retrieved, the artefacts are reviewed by developers to ensure their relevance to the current task. If necessary, developers can refine their query to retrieve additional or more precise artefacts. This iterative process ensures that a comprehensive and contextually relevant set of artefacts is gathered, providing a solid foundation for the subsequent development stages.

Contextual Model Generation

After identifying and validating relevant artefacts, the identified models are utilized to provide critical context for the **model generation process** conducted by Reform tool. This process supports the creation of new artefacts or updates existing ones, ensuring alignment with project goals and adherence to established requirements and specifications.

Tools and Workflow

The Figure 23 outlines the tools and workflow underpinning this incremental development approach as part of the SmartDelta methodology.

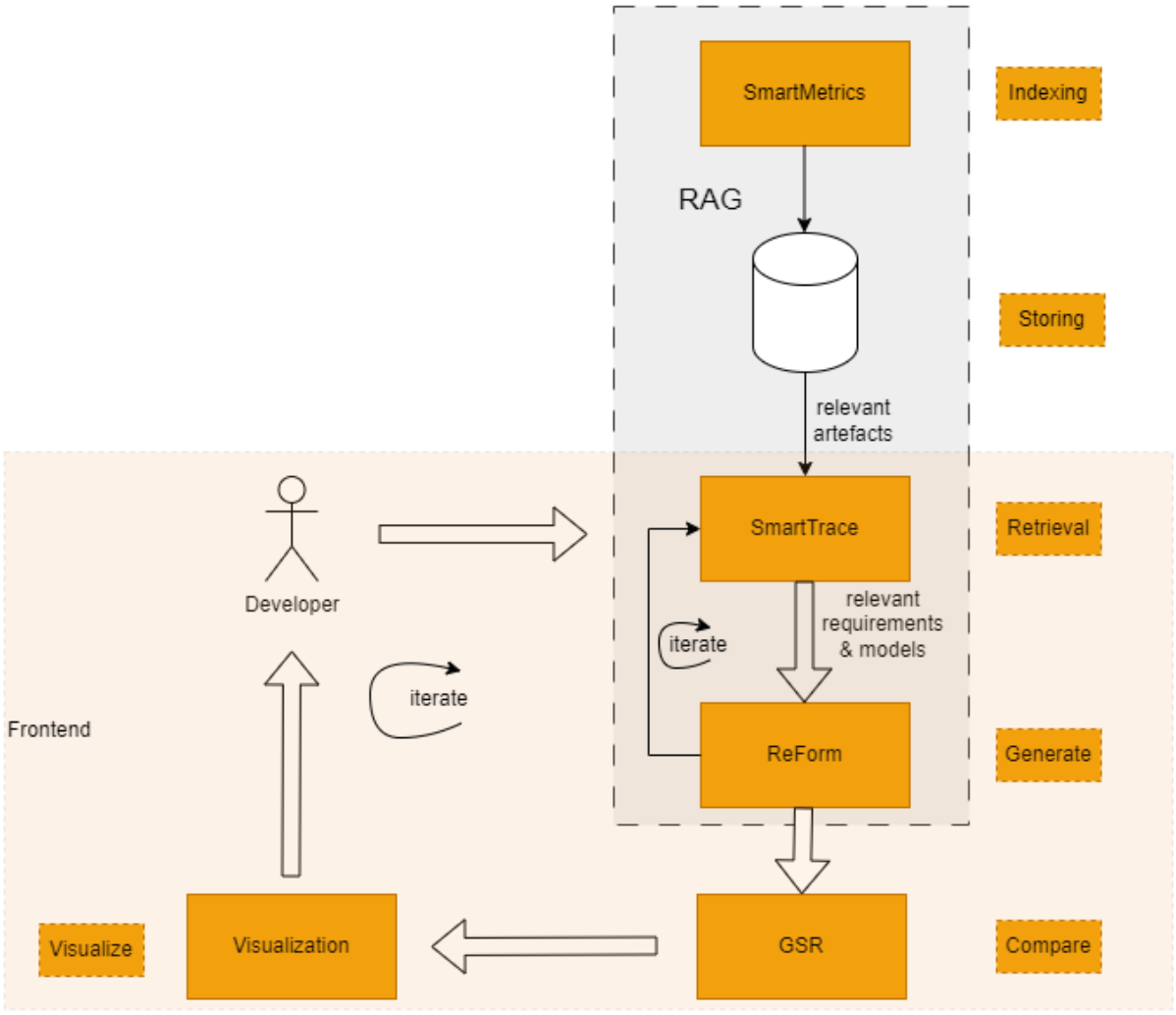


Figure 22: Example Workflow and corresponding tools

Key Tools and Their Roles

SmartMetrics: Responsible for indexing artefacts into a central repository for efficient retrieval. This includes embedding generation, labeling, and storing software metrics.

SmartTrace: Retrieves artefacts from the repository based on developer queries. SmartTrace ensures the retrieval of specifications, models, and code essential for implementing new features or changes.

ReForm: Leverages retrieved models to generate or adapt artefacts. This tool ensures the consistency of newly created or updated artefacts with overarching project goals and requirements. With this step, the Retrieval-Augmented Generation (RAG) pipeline is completed.

GSR: Provides a comparison between two models by calculating deltas at the branch or module (subgraph) level and measuring similarity. These features help developers analyze changes, assess their impact, and plan subsequent code-level development.

Visualization: Supports developers with visualization and interaction capabilities. Its intuitive frontend highlights deltas identified by TWT and displays software metrics to facilitate decisions based on quality and architectural considerations.

Conclusion
This integrated, tool-supported methodology enables a structured and efficient approach to incremental development. By automating the retrieval and generation of artefacts and providing robust tools for analysis and validation, the SmartDelta framework significantly improves developer productivity and ensures high-quality outcomes in complex software projects.

c. Tools descriptions

Four tools have been developed from scratch.

GSR – Graph Similarity Recommender (TWT-Tool)

The GSR tool facilitates the automatized comparison of State Machines with transitions defined by ISO standards, quantifying their similarity and identifying deltas. This analysis is essential to track the evolution of normative requirements over time or to recognize reuse opportunities. Manual comparison of State Machines, especially across numerous instances, is labor-intensive and impractical. By leveraging hierarchical decomposition based on ISO standards, GSR improves efficiency and enhances analysis quality.

The tool is implemented in python and the input for the tool consists of two State Machines in Json format, as well as a hierarchical modularisation of the ISO standard. This modularization is used to decompose the State Machines into submodules that are being compared. This reduces the runtime of the comparison. Currently, the modularization must be prepared once as a preprocessing step. Additionally, a converter makes it possible to also use State Machines in ceps format. However, the tool does not support nested State Machines as input, meaning State Machines containing states that are themselves State Machines.

The output of the tool is also a Json file that contains the decompositions of the input State Machines. It also contains mappings between the states and transitions of the State Machines, indicating the deltas. Finally, the output Json contains similarity values between the submodules of the State Machines, including the overall state machines. Note that the tool does not consider guards and actions yet. The tool version used for the evaluation is GSR V1.0.

For more information on the GSR Tool, see chapter 4.3 of D4.5.

SmartSearch (SmartMetrics & SmartTrace)

SmartMetrics and SmartTrace are two Python-based tools that work together to offer a smart search functionality for project users. SmartMetrics scans and indexes the Git repositories of a project and stores all calculated data in a PostgreSQL database, enhanced by vector search functionality using the pgvector extension. The scanning process loops over all commits, branches and files, allowing for a comprehensive analysis of the entire history of each repository. This analysis can be used to detect trends and degradations. Each compatible file is analyzed depending on its type and configuration. It is possible to calculate software metrics, dense vector embeddings using sentence transformers, and semantic data generated by large language models (LLMs). Based on this data, diagrams can be generated using dashboard technologies like Plotly to create Python-based interfaces that run in a browser, or by connecting to common tools like Grafana and performing SQL queries on the data.
Using an LLM to create tags and a short paragraph to describe the file content introduces a common representation that enables the retrieval of all kinds of file types with a single query. The query itself is performed by the tool SmartTrace, which is a command-line-based tool that has also been integrated into the browser-based interface developed by Fraunhofer (see next section). SmartTrace runs 5 search streams consisting of vectors searches (cosine, L2, dot product) and the PostgreSQL full text searches named *tsvector* and *tsquery*. The results will be merged

according to the score of each result. The following Figures 24 and 25 provide an overview of SmartMetrics and SmartTrace and their connections.

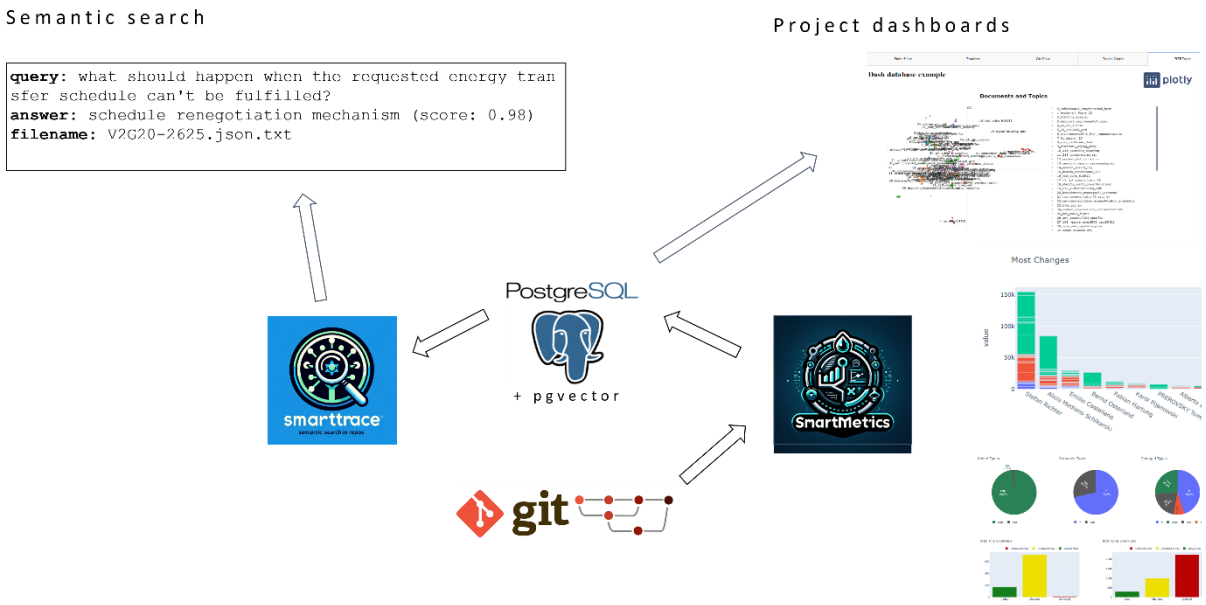


Figure 23: SmartMetrics and SmartTrace Architecture

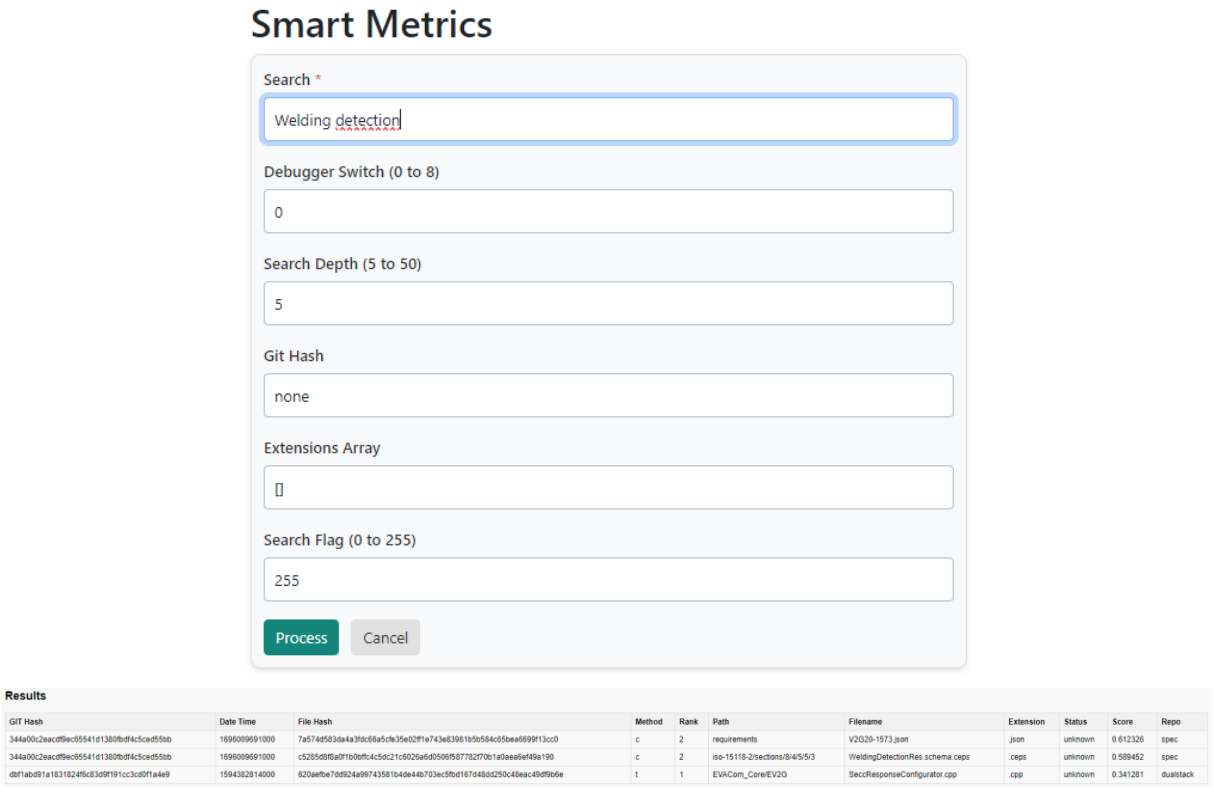


Figure 25: SmartTrace integrated in UI

SmartMetrics

Input: git repositories, configuration

Output: data stored in database

Tech: Python, Sentence Transformer Embedding Models, LLM (GPT-4o, Llama3)

Limitations: TRL 3, only basic software metrics as example implemented, semantic indexing needs improvement for more accurate search

SmartTrace
Input: smartmetrics database
Output: search results based on query and settings
Tech: Python, PostgreSQL, Hybrid search
Limitations: TRL 3

Architecture Analysis and Visualization Tool (Fraunhofer)
The Architecture Analysis and Visualization Tool can be used to extract various architectural views from a variety of input file types. Specifically, the tool analyzes .ceps and log files to produce three unique views: the execution flow state diagram, the log similarity matrix, and the event flow diagram.

- The **Execution Flow State Diagram** analyzes individual log files within the input folder and generates state transitions captured in the log file.
- The **Log Similarity Matrix** is a heatmap that depicts the similarity among log files within a folder.
- Finally, the **Event Flow Diagram** shows the relationship among various state machines described in the .ceps file.

The general workflow is as depicted in the Figure 26 below:

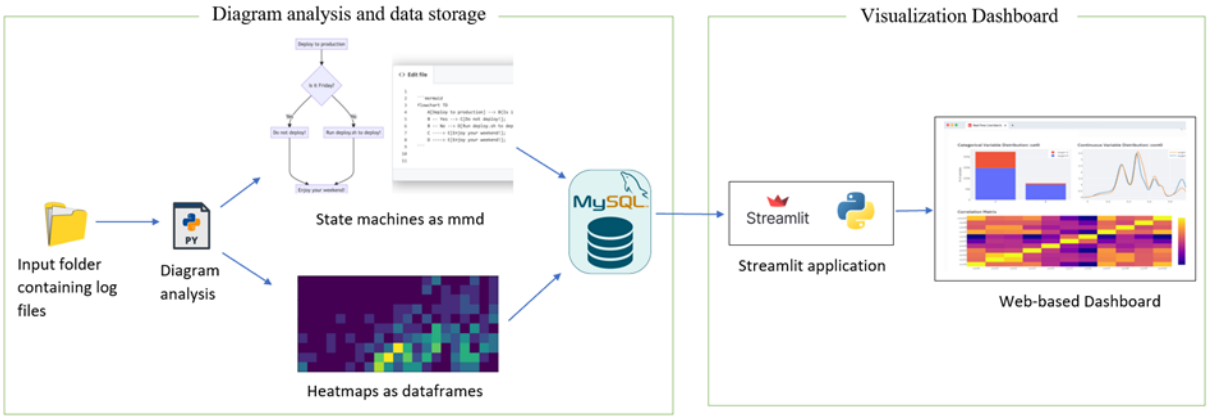


Figure 26: General Workflow of the Architecture Visualization Tool.

As shown in Figure 16, the architecture views are initially computed in a textual format (except for the Events Map Diagram) through the analysis module as markdown-like syntax and data frames and are stored in a database for persistent access. The graphical representations of the computed diagrams are then available through a web-based visualization dashboard.
The tool is publicly available on GitHub at:
https://github.com/SmartDeltaFraunhoferFOKUS/Architecture_Visualization_Tool. The steps to install and use the tool, along with the technologies used, are explained in detail in the README and Wiki.

ReForm Tool - Automated Requirements Formalization (IFAK)
ReForm is a tool that automatically transforms textual requirements into structured state machine models. It processes requirements from ISO 15118 and similar standards, ensuring formalized and machine-readable representations. The tool uses language models (sentence-transformers) to generate vector embeddings for all stored requirements. These embeddings are stored in a FAISS vector database, enabling fast and efficient similarity searches. When a new requirement is provided, ReForm retrieves the most similar existing requirements and their state machine models.

This retrieval-augmented generation (RAG) pipeline provides context to generate a new state machine. It converts the new requirement into a JSON-based state machine model using a recent large language model (Llama 3). The model consists of states, transitions, events, guards, and actions, ensuring a complete representation. The newly generated state machine is then merged into the existing system model.

The tool supports customization of embedding models and inference models, making it adaptable for different domains and use cases. The tool improves requirement traceability, consistency, and automation in system modeling. It reduces manual effort in formalizing requirements, increasing efficiency and accuracy. ReForm is useful for engineers, researchers, and developers working on requirement-based modeling. Its structured approach makes requirement analysis and system design more scalable and automated.

Input: requirement (JSON)
Output: state machine model (JSON)
Tech: pytorch, sentence-transformers, langchain, faiss, llama-cpp-python, Llama 3
Limitations: TRL 3, the accuracy depends on the provided examples (RAG) and language model

The tool will be made publicly available on Github:
https://github.com/ifak-prototypes/nlp_reform

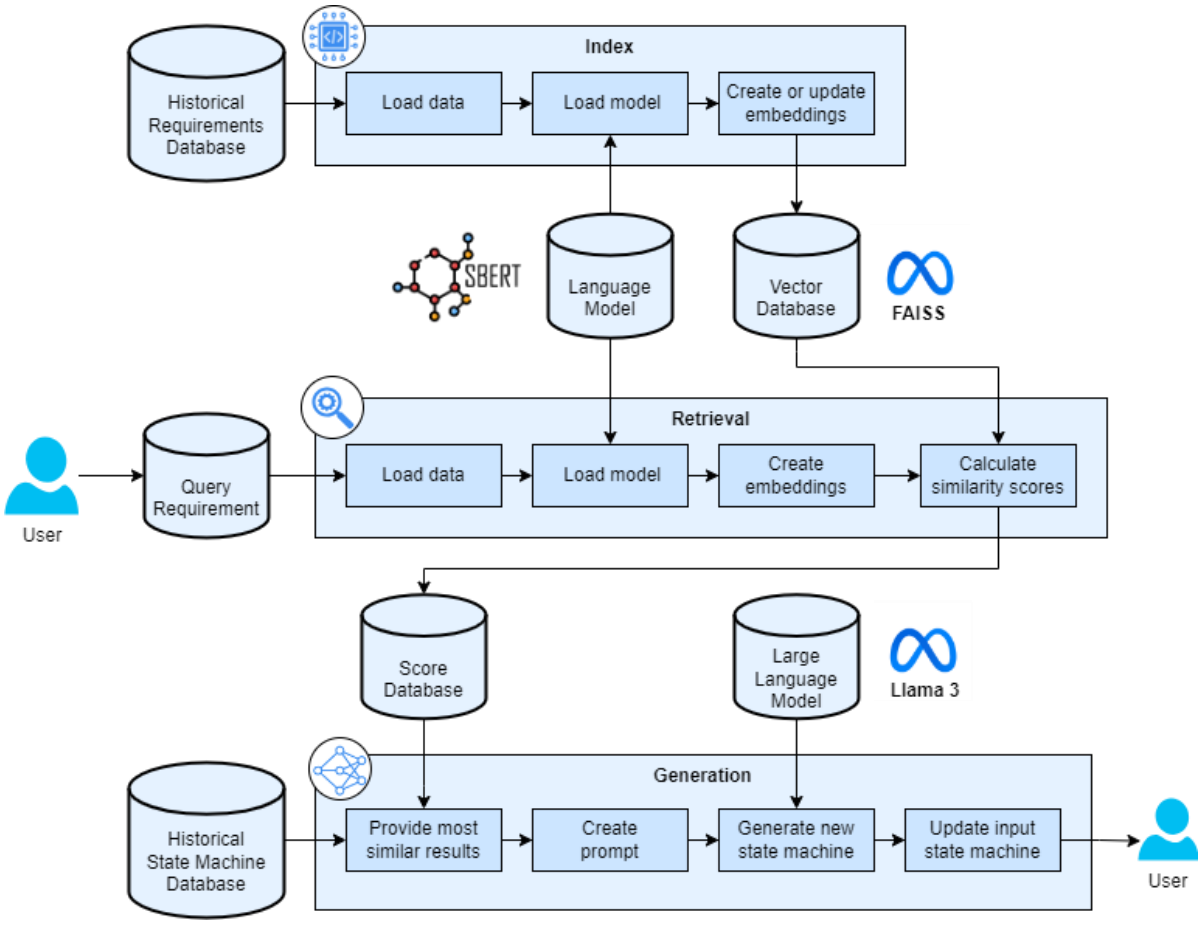


Figure 27: Pipepline of ReForm tool for automated model generation.

d. Visualization

The use case brings together a range of tools developed within the SmartDelta project to address various requirements, including resolving architectures from textual data, identifying similarities or deltas among models, and generating models from requirements. The web-based integrated environment, which provides a graphical interface for easier execution of these tools, also offers a range of visualization options to make the results more accessible. The visualization options available within the integrated environment are discussed below:

1. **Visualizing the input models:** Tools like GSR require users to input state machines in JSON format. These model definitions can become quite complex, especially where there are large number of states and transitions. To help users easily verify the input state machines, the web-based tool offers visualization options. It leverages PlantUML[1] to generate visual representations of the state machines defined in the input JSON files (Figure 28).

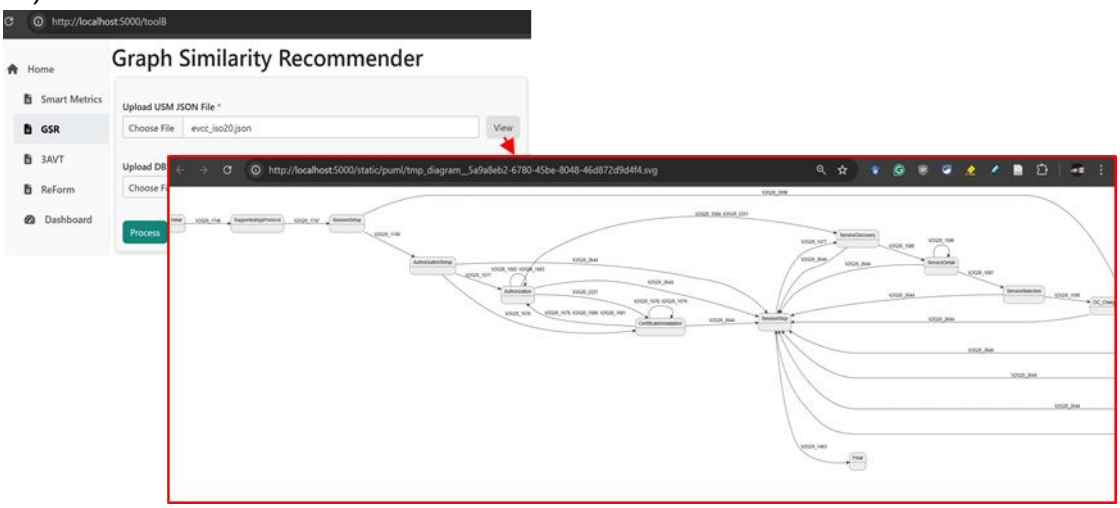


Figure 28. Visualize input state machine.

2. **Delta Visualization:**
   Full View: The GSR tool generates a JSON formatted output that details the similarities between the input models. This JSON is highly expressive, allowing users to identify specific states and transitions that have been added, removed, or changed between state machines being compared. This information is then utilized to create a visualization of the delta between the state machines. States are segmented in accordance to the corresponding modules defined within the GSR tool.

[1] https://plantuml.com/

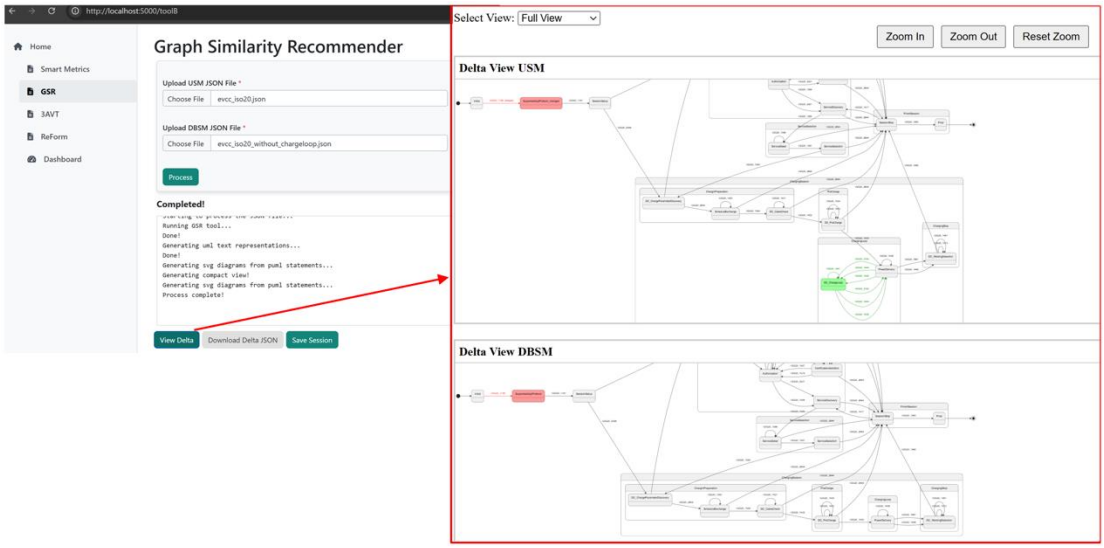


Figure 29: Delta visualization showing "Full View".

As illustrated in Figure 29, the visualization displays all states within their respective modules, providing a detailed view. States and transitions present in one state machine but absent in the other are highlighted in green, while those differing between the two models are highlighted in red. This colour coding also applies to edges. The tool leverages PlantUML to generate these visualizations.

Compact View: The delta visualization window also offers an option to collapse the visualization, displaying only the modules while hiding all states within them. Incoming and outgoing edges to and from the modules are still plotted, but edges that connect states solely within a module are hidden.

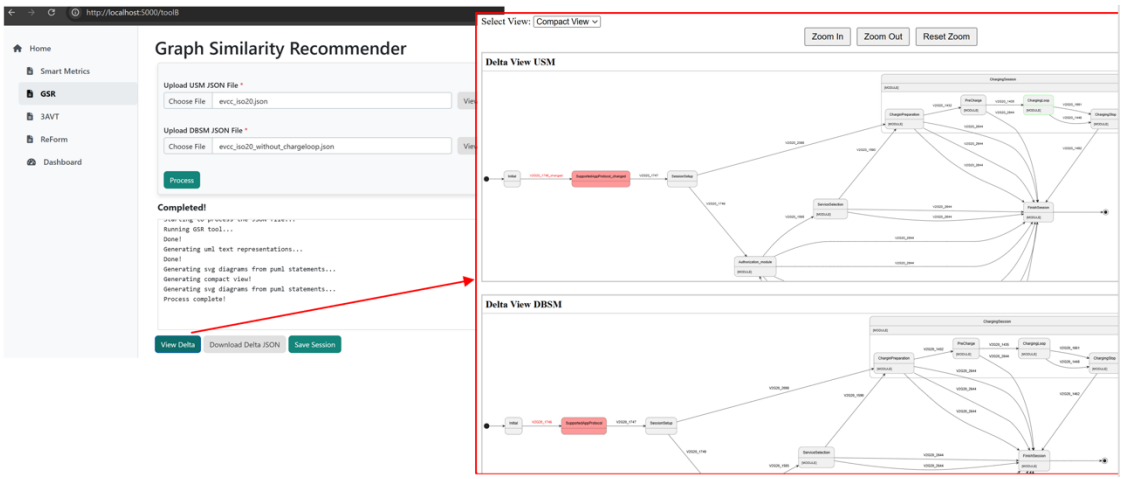


Figure 30: Delta visualization showing "Compact View".

Modules containing exclusively green or red states are highlighted accordingly (see Figure 29 and Figure 30). Modules containing both green and red states are coloured red. Collapsing the view to show only modules simplifies the visualization, making it easier to interpret differences when both input state machines have a large number of states and transitions, which can otherwise make the diagram difficult to navigate.

3. **The Visualization Dashboard:** The visualization dashboard is a Streamlit[2] web application that attaches to a MySql database with a pre-populated data. The data is inserted during the execution of the diagram analysis and storage (see Figure 31) process. The dashboard offers a tabbed view, enabling intuitive navigation across various visualization components and configuration options.

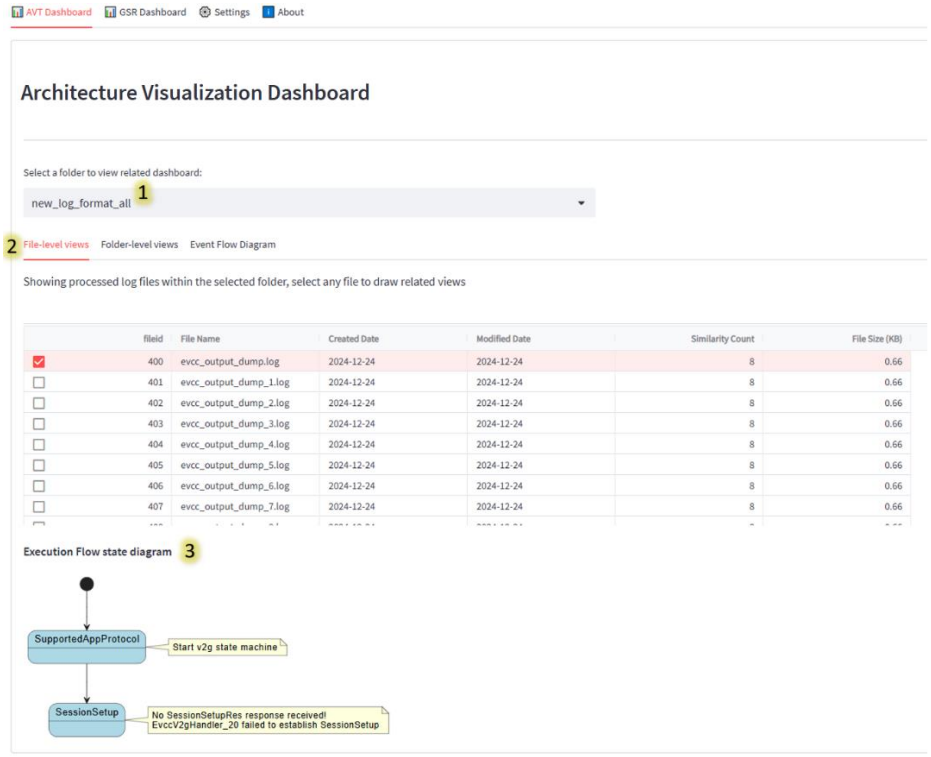


Figure 31: Architecture Visualization Dashboard – Execution Flow State Diagram.

As shown in Figure 8, the "file-level view" (2) enables users to select an individual processed file from within a processed folder (1) and view the **Execution Flow State Diagram** (3) that captures the runtime behaviour of a model as captured in that log file. Activities logged during the execution of a state are displayed alongside the corresponding states. For example, *Start v2g state machine* is depicted as an activity that occurred during the execution of the `SupportedAppProtocol` state.

[2] https://streamlit.io/

Figure 32: Architecture Visualization Dashboard – Log Similarity Matrix.

Similarly, "Folder-level view" includes the **Log Similarity Matrix** (4), a heatmap that depicts similarity among ingested log files (Figure 32) within the selected folder (1).

Finally, the **Event Flow Diagram** (5) in Figure 33 illustrates the relationship among state machines based on the shared events. Users can drag and drop multiple .ceps files, that define state machines, into the input widget in the UI. These files are then processed to extract both outgoing and incoming events associated with the state machines defined by each .ceps file. The resulting diagram displays state machines as nodes and uses event connections as edges.

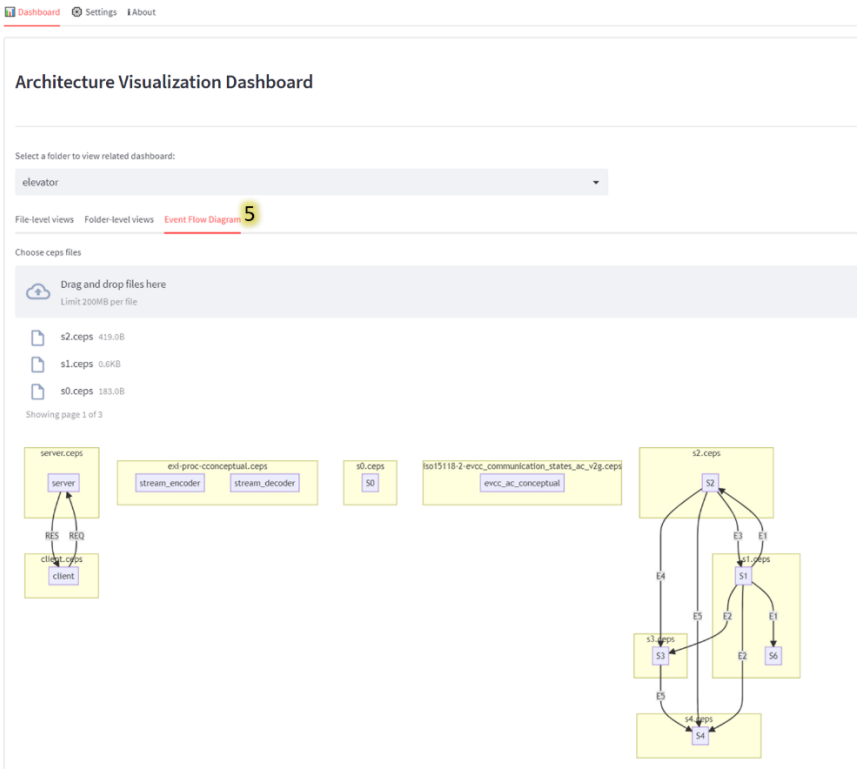


Figure 33: Architecture Visualization Dashboard – Events Flow Diagram.

The generated architectural views are helpful for developers and first level QC in fault diagnosis. For instance, when a fault occurs, the corresponding log file can be analyzed, allowing users to visually observe the sequence of events via the execution flow state diagram. Additionally, the similarity matrix facilitates the identification of similar faults from previous logs, aiding in the discovery of similar faults. As a result, these diagrams provide valuable insights for initial quality control, aiding in localizing faults and supporting the diagnostic process.

e. Evaluation setup

The tools are evaluated using technical KPIs (refer to the next section, *Evaluation Results*), which measure specific functionalities. However, while these technical KPIs provide deep insights into performance, they do not fully reflect the tools' effectiveness in real-world scenarios. To address this, the following diagram illustrates potential tool integrations and workflows for achieving a common goal: incorporating and implementing new requirements into an existing corpus.

The process consists of the following steps:

1. **Artefact Retrieval:**
    - The first step is to retrieve existing artefacts relevant to implementing the new requirement. Specifically, the focus is on identifying all relevant state machine models within the corpus.
    - This task can be performed manually by the developer or automated using **SmartTrace**, which efficiently retrieves artefacts based on a query.
    - **Remark:** In many cases, the retriever (e.g., SmartTrace) will identify multiple relevant models. It is up to the developer to decide which model to use.
2. **Model Modification**
    - The retrieved artefacts, along with the new requirement, are passed to the **ReForm** tool, which outputs a modified model.
    - This process can be iterative, allowing developers to use different input models or make small adjustments to the requirement until an acceptable output model is achieved.
3. **Visualization and Review**
    - The graphical user interface developed by **FOKUS** displays the model and highlights the modifications.
    - While direct intervention in this workflow is limited, developers can modify the model externally if needed.
    - Once a requirement is added and the resulting model is accepted, developers can either continue adding new requirements to the updated model or switch to another available model.
4. **Optional: Model Comparison**
    - At this stage, the **GSR** tool can be employed to compare models and assist in selecting one based on similarity metrics.
    - However, as similarity is not always the sole criterion, automatic selection based solely on similarity is not feasible.
5. **Delta Analysis**
    - After iteratively adding requirements, developers can use **GSR** to compare the original and final output models, analyzing cumulative deltas to understand the impact of changes.

Besides this core workflow, the following steps are available:

- **Dashboard Integration**

- The *Dashboard is* independent of this workflow but can monitor specific project aspects. It also serves as an extension for other dashboards, such as those in the **Jira** toolchain (Atlassian)

- **Visualization Enhancement**
  - The **AVT** tool provides an alternative method for visualizing models and their interconnections. This is particularly useful for inspecting models retrieved by SmartTrace.
- **Model Conversion**
  Before processing, models must be converted into the appropriate format.
  - The **GSR** and **ReForm** tools require input in JSON format, while the corpus and AVT operate in the original CEPS format.
  - A converter tool developed by **TWT** facilitates this transformation.

The experiences, observations, and results obtained through this evaluation setup are detailed in the next section.

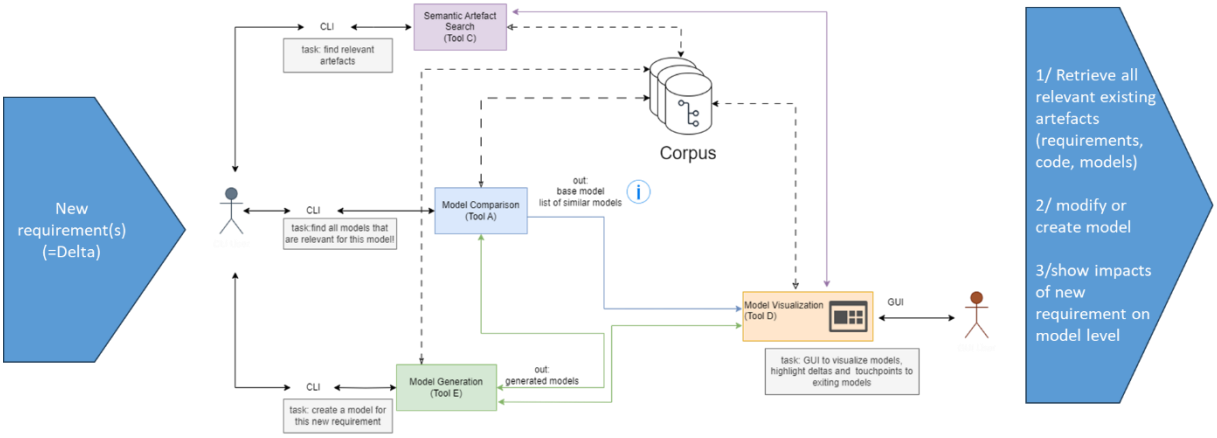


Figure 34: Evaluate tools in context of adding new requirements to existing corpus

f. Evaluation results

UC2.FR_A:

KPI1: Alignment of Similarity Values with User-Defined Similarity

The similarity value of two State Machines computed by the tool is a percentage value between 0% and 100%, 100% meaning that the State Machines are identical except for possibly different State names and 0% meaning that in order to obtain one State Machine from the other one, all transitions have to be deleted or relabelled.

This KPI evaluates whether the similarity values computed by the tool aligns with the similarity defined by a user. To assess this, we prepared 12 examples of state machines with varying degrees of similarity. From these, we created four test sets, each consisting of one state machine compared against eight samples selected from the remaining 11 state machines.

After visualizing the state machines, we manually categorized the pairs into three similarity levels: **High Similarity**, **Medium Similarity**, and **Low Similarity**. We then applied the GSR tool to these pairs to compute similarity values, categorizing them as follows based on the computed percentages:

- **High Similarity:** ≥ 85%
- **Medium Similarity:** ≥ 60% and < 85%
- **Low Similarity:** < 60%

The computed categories were compared to the subjective categories. For 29 out of 32 comparisons (90.62%), the computed category matched the subjective assessment.

KPI 2: Accurate Allocation of Changes and Conflict Identification

This KPI assesses the tool's ability to accurately allocate changes and identify conflicts. We generated 10,000 test examples by applying random modifications to a base test example. The possible modifications included the following:

- Deleting a random edge or node
- Adding a new random edge or node
- Relabelling a random edge or node

We evaluated the tool's performance in two areas:

1. Change Allocation: Verifying if the similarity value of a module is exactly 1 when no changes were applied to that module.
2. Conflict Identification: Accurately detecting deletions or relabelling applied to any module.

The tool successfully determined these aspects in 99.81% of cases.

KPI 3: Correct Determination of Deltas Between State Machines

This KPI evaluates the accuracy of determining deltas between two state machines. Using the same 10,000 test examples generated for KPI 2, we assessed the deltas between the base state machine and the modified examples.

The process involved identifying the delta path from the base state machine to the modified example, then applying the determined deltas back to the base state machine. The resulting state machine was then compared to the modified example to check for equivalence. This process resulted in successful matches in 99.98% of cases.

Detailed evaluation results for single requirements and the corresponding KPI are shown in the table below.

UC2.FR_C:

KPI4: Semantic search functionality covering different artefacts types

The tools offer search functionality for three artifact types: requirements (JSON), models (CEPS format), and code (C++). All types can be retrieved with a single query, as illustrated in the figure in the tools description section above. However, it is important to note that our current evaluation does not provide a measure of the quality of the search results. Due to the absence of a ground truth, the results obtained cannot be considered definitive. Since users can query using entire sentences, such as requirements from ISO 15118, the results should not be interpreted as development recommendations. Different developers may have varying expectations for the results, making the definition of a ground truth challenging. Additionally, there are many design alternatives and optimization options to consider, but choosing the right direction remains an open research question that needs to be addressed in an upcoming project. The current tools serve as a starting point for further exploration and analysis.

KPI5: Number of Valuable Technical Insights into the Project Repository, Including Semantic Insights and a Delta View

Diagrams were counted if they provide new technical insights into the repositories and cannot be generated by other tools. The value or helpfulness of a diagram depends on the project role and personal preferences. There is one delta view (showing the difference between two commits) and several diagrams based on semantic information.

By selecting two repositories, two branches, and two commits, the Delta View offers several diagrams in a side-by-side comparison, including:

- Number of commits per time slot, where a time slot may represent a sprint in an agile development process.
- The types of files that have been worked on.
- The types of changes (added, modified, or deleted lines of code) per user.
- A table showing average values, such as average cyclomatic complexity.
- A time plot showing the evolution of average cyclomatic complexity over time.
- Three diagrams illustrating the evolution of:
    - Added lines of code
    - Changed lines of code
    - Deleted lines of code

Having a list of AI-generated labels and descriptions for model and code files allows for the creation of the following diagrams (as examples):

- Labels that have been worked on per time slot (sprint), which can be used to identify topics and semantic trends.
- Evolution of activity rates per label, e.g., identifying the top X labels per time slot and plotting their evolution over time. This is helpful for tracking the evolution of labels and topics.
- Labels per user, which allows for the identification of experts, single points of knowledge, and other relationships between labels and users.

During the project, new approaches for visualizing the semantic structure of complex documents, such as the different parts of the ISO 15118 standard, were explored and tested.

The following figure 35 shows an interactive visualization of the semantic document structure of the ISO 15118-2 standard after extracting and processing the raw data from the PDF file.

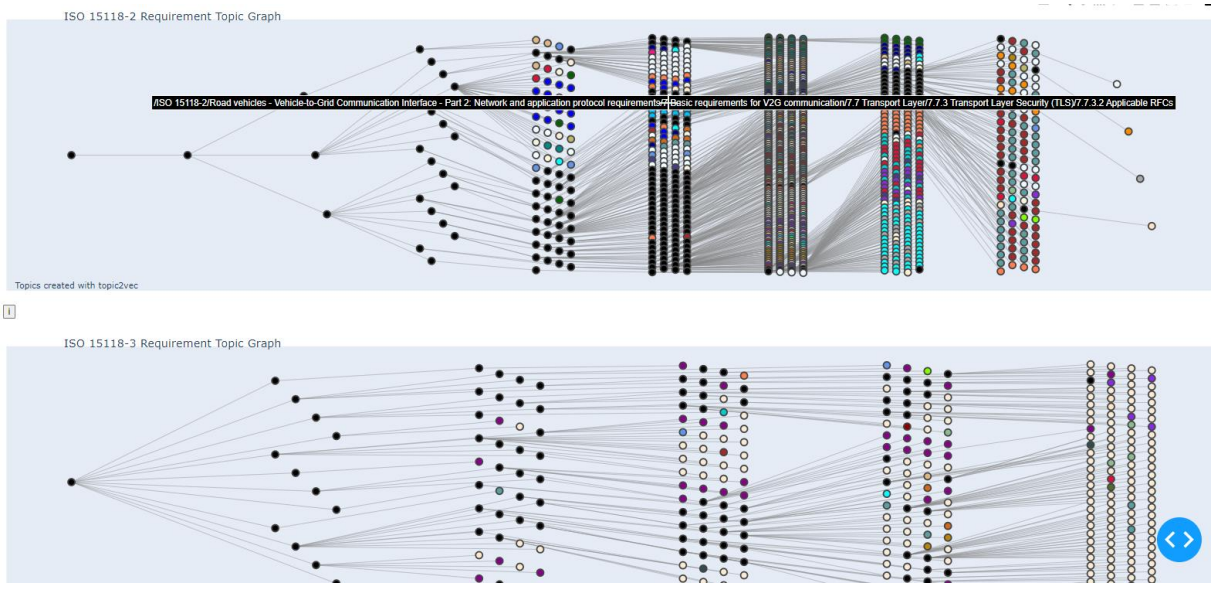


Figure 35: interactive visualization of the semantic document structure of the ISO 15118-2 standard

It is important to note that many aspects were explored during the project, and our experiences with AI have led to significant progress. Many of the ideas, approaches, and implemented tools were part of our journey to explore the new possibilities of generative AI in the context of software development. The functional prototypes currently have a low Technology Readiness Level (TRL) of 3 and are not intended for use in developing software solutions for production environments. Substantial redesigns and improved concepts that leverage the latest findings from global AI research are necessary to

achieve a level of quality, stability, scalability, and effectiveness. See more in next section “Recommendations for industry adoption”.

UC2.FR_D:

KPI6: Number of generated /up-to-date architectural views for a variant.

The Architectural Visualization Dashboard from FOKUS currently provides 3 unique architectural views.

a. Execution flow diagram (Figure 8) uses log files to visualize sequence of events during model execution.
b. Log similarity matrix (Figure 9) shows similarity among ingested logs in a heatmap.
c. Events flow diagram (Figure 10) visualizes how multiple models are related based on shared events. The input models are written in *.ceps* format.

Apart from these, the delta visualizations, that include full view (Figure 6) and a compact view (Figure 7), are available from the integrated environment. These diagrams use the output from the GSR tool developed by TWT. Thus, a total of 5 different architecture views are offering various levels of information and insights.

KPI7: Automation of visualizing model changes.

The KPI measures the extent to which the visualization of delta between models is automated across various tools. The Model changes can currently be visualized in two scenarios:

1. Using the GSR Tool: Users provide models as JSON files, which serve as inputs to the tool. Once processing is complete, they can visualize the delta between the two inputs.
2. Using the ReForm Tool: Similarly, when executing the ReForm tool from the integrated environment, users can view the delta between the input and output state machines. They can also modify requirements iteratively, and after each iteration, the delta between the initial input and the updated output state machine becomes available for visualization.

In both cases, the visualization module computes delta between models automatically once the tool execution (GSR or ReForm) is complete.

However, when using the ReForm tool, users can replace the initial input state machine with the current output state machine and rerun the tool to generate another output state machine. In this scenario, manual intervention is required. Users must save the final output state machine to disk, open the GSR tool from the integrated environment, and execute it by providing the initial state machine and the final output state machine as inputs to view the delta. This step is currently missing from the automated toolchain. Thus, about 90% of the automation goal has been met, with the final step requiring user intervention.

UC2.FR_E:

KPI8: Accuracy of generated models.

We evaluated the accuracy of automatically generated state machine models for 72 requirements from ISO 15118 Part 2 and Part 20, comparing them against manually created models as the ground truth. Our evaluation used a macro-average accuracy metric, incorporating Levenshtein similarity across key model properties (source/target state, event, guard, and action), allowing for partial correctness.

We tested different LLMs and selected the 6 currently best-performing models for detailed analysis. Each model was tested under Zero-Shot (without RAG) and Few-Shot (with RAG) prompting. To ensure robustness, we conducted three independent runs per model, both with and without RAG, averaging the results to account for statistical deviations.

The accuracy results revealed significant performance differences. Mistral Small 3 achieved the highest accuracy at **83.3%**, followed by Phi-4 (80.3%), surpassing the target threshold of 80% accuracy. The other models performed slightly lower but still demonstrated improvements with RAG-based prompting.

The Levenshtein-based metric provided a more flexible and granular evaluation, tolerating minor variations in naming, structure, and syntax while still rewarding overall correctness.

Notably, RAG-based prompting consistently improved accuracy, highlighting the benefits of incorporating relevant context for structured model generation. These findings confirm that our developed method is effective and generalizable, opening opportunities for further fine-tuning, optimized retrieval strategies, and hybrid approaches to enhance performance in real-world applications.

Language Model			Prompt	Average Accuracy					
Name	**Size**	**Quant.**	**Use RAG**	**Source State**	**Target State**	**Event**	**Guard**	**Action**	**Total**
Code Llama	7B	F16	No	72.3%	57.3%	59.3%	51.5%	39.3%	55.9%
			Yes	64.9%	47.6%	79.1%	72.2%	39.2%	60.6%
DeepSeek Coder	7B	F16	No	82.7%	67.1%	65.0%	60.8%	43.3%	63.8%
			Yes	72.8%	54.5%	84.2%	76.0%	44.9%	66.5%
Code Gemma	7B	F16	No	87.5%	70.9%	70.7%	55.7%	43.7%	65.7%
			Yes	73.8%	60.1%	88.6%	79.8%	46.1%	69.7%
Qwen2.5 Coder	7B	F16	No	80.9%	68.5%	65.8%	59.7%	46.5%	64.3%
			Yes	75.3%	70.3%	90.7%	77.9%	**52.1%**	73.3%
Phi-4	14B	Q8	No	**94.7%**	71.9%	67.3%	67.4%	45.4%	69.3%
			Yes	88.3%	82.7%	93.6%	85.1%	51.8%	80.3%
Mistral Small 3	24B	Q8	No	93.6%	75.5%	77.2%	65.7%	47.6%	71.9%
			Yes	94.6%	**91.3%**	**93.8%**	**86.6%**	50.2%	**83.3%**

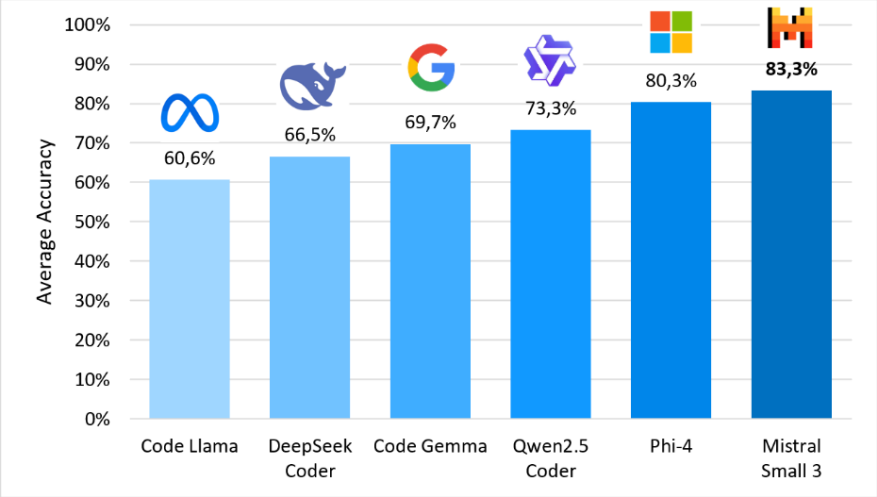


Figure 36: RAG-based prompting accuracy

KPI9: Compatibility with existing models.

We evaluated the compatibility of the LLM-generated state machine models with existing models by integrating them into comprehensive system-wide models. Our deterministic approach ensured that newly generated sub-models seamlessly merged with the existing model structure.

In all test runs, the integration was **100%** successful: the new states were incorporated correctly, and the LLM consistently recognized and reused pre-existing states from the modeling process. This confirms that our approach fully meets the targeted compatibility goal, demonstrating the method's reliability for extending and refining existing system models.

Table 4: KPIs overview table

Requirements	Tool	Solution partner	KPI Definition	KPI Base Values	KPI Target Values	KPI achieved Value
UC2.FR_A	GSR	TWT	[tech]- success rate of finding similar models - in <target value> of all cases (according to a similarity measure that still has to be defined)	0	90%	90,62%
	GSR:	TWT	[tech] - success rate of finding conflicting models - in <target value> of all cases (according to a measure that still has to be defined)	0	90%	99,81%
	GSR:	TWT	[tech] - success rate of finding all impacts -in <target value> of all cases	0	90%	99,98%
UC2.FR_C	Smart-Trace	Akkodis	Semantic search functionality covering different artefacts types	0	2	3
	Smart-Metrics	Akkodis	Number of valuable technical insights into the project repository including at least semantical insights and a delta view	0	8	8
UC2.FR_D	Visualization	Fraunhofer FOKUS	[tech] number of generated /up-to-date architectural views for a variant	0	4	5
	Visualization	Fraunhofer FOKUS	[tech] automation of visualizing model changes	0%	90%	90%
UC2.FR_E	ReForm	IFAK	[tech] correct model generation in <target value> of all cases	0%	70%	83.3%
	ReForm	IFAK	[tech] high compatibility with existing models in <target value> of all cases	0%	80%	100%

g. Recommendation for industry adoption

The tools developed during this project present significant potential for improving development efficiency, velocity, and quality in software engineering. Key benefits include:

- **Enhanced Efficiency in Artifact Retrieval**: The tools facilitate a more efficient process for finding relevant artifacts across different repositories, reducing the effort required for developers to access necessary information.
- **Automated Model Comparison and Visualization**: The ability to automatically compare models, along with integrated visualization features, streamlines the analysis process and aids in understanding differences and similarities between various design elements.
- **Informed Design Decisions**: By reusing and modifying design decisions based on software quality metrics, teams can make more informed choices that enhance the overall quality of the software.
- **Improved Planning Capabilities**: The tools support better planning by providing insights into quality trends and semantic overviews, such as the distribution of employees across topics/keywords and domain expertise within the project.

While the methodology shows promise, it is important to recognize that some aspects are use-case specific, particularly regarding the types of models employed (e.g., UML state machines in CEPS or JSON format). However, the underlying pipeline is generally applicable to various software development contexts.

Challenges and Limitations

Despite the advantages, several challenges must be addressed to enable productive usage and facilitate broader industry adoption:

GSR: Limitations in Model Comparison: The current model comparison for delta calculations has inherent limitations, particularly regarding the assumptions required for effective model comparison, such as a common namespace for states and transitions. This may hinder the ability to achieve accurate comparisons. Additionally, the model comparison tool requires a module definition that is not readily available, limiting its automatic use. A solution for automatic module calculation is currently lacking, which restricts usability to prepared environments where module definitions exist.

SmartMetrics and SmartTrace: Quality of Semantic Indexing: The effectiveness of semantic indexing using large language models (LLMs) heavily depends on the quality of the models and artefacts being analyzed. Poor code quality, insufficient comments, and outdated documentation can lead to suboptimal semantic representations, such as embedding vectors, labels, and descriptions. However, as these challenges are common in many real-world use cases, it is crucial for new approaches to address and adapt to such constraints effectively.

While the current era of AI-based tools is still in its early stages, the potential for improvement is immense. Future advancements in AI and machine learning promise significant enhancements to semantic indexing and retrieval capabilities. Nevertheless, good engineering practices remain essential to harness this potential and develop tools with production-level maturity that meet real-world demands.

Challenges in Semantic Search

Semantic search capabilities face several challenges that hinder accuracy and relevance:

1. **Merging Multiple Search Streams**
    a. The system employs five distinct search streams—three vector-based searches and two text-based searches—which are merged to produce the final results.

   b. These streams often fail to meet user expectations due to the inherent difficulty of reranking results across different modalities. For example, comparing vector-based and text-based search scores is complex and may not align with the user's personal preferences.
2. **Unified Queries for Multiple Artefact Types**
   a. Conducting a single search query across multiple artefact types (e.g., requirements, models, and code) adds further complexity.
   b. Identifying optimal strategies for retrieving artefacts across these diverse domains remains an unresolved challenge.
3. **Indexing Consistency**
   a. Tags generated by LLMs for artefacts are often inconsistent or too generic, particularly when applied across different artefact types or files.
   b. This lack of specificity reduces the utility of the indexing process and hampers effective retrieval.

Areas for Improvement

To enhance the performance of SmartMetrics and SmartTrace, two primary areas require further development:

1. **Improved Search Quality**
   a. Efforts should focus on refining the semantic indexing and search ranking algorithms to deliver more accurate and relevant results.
   b. Addressing inconsistencies in tagging and improving the merging logic for multiple search streams will be critical.
2. **Reduced Latency**
   a. Minimizing latency during retrieval processes is essential for ensuring a smoother user experience.

ReForm: Artifact Generation Challenges: The ambitious goal of automating requirement-based change implementation, starting at the model level, presents several challenges that limit the tool's practical usage in its current form.

Current Limitations

1. One-to-One Mapping Assumption
   a. The ReForm tool assumes a one-to-one mapping between individual requirements and small state machine models as input for the generation process. However, this scenario rarely exists in practice.
   b. While individual requirements are typically available, existing models are often larger and represent a set of interconnected requirements, making it difficult to isolate specific mappings.
2. Sequential Requirement Addition
   a. Requirements must be added one by one, which slows the process and limits scalability.
   b. Additionally, the tool's scope is currently restricted to the model level, without incorporating knowledge of the broader codebases or technical environment.

Broader Challenges in AI-Based Generation

These limitations highlight a more general challenge for AI-based artifact generation tools: achieving better integration of the technical context and environment to produce results aligned with real-world conditions. Key issues include:

1. Pre-Processing of Requirements

   a. Effective pre-processing of requirements is essential to clarify ambiguities and provide the tool with improved context awareness.
   b. This pre-processing must account for both the explicit requirements and the implicit assumptions present in the project environment.
2. Comprehensive Context Integration
   a. The context for generation spans the entire project environment, including:
      i. Requirements
      ii. System architecture
      iii. Subsystem and interface specifications
      iv. Codebases and libraries
      v. The overall technical environment
   b. Integrating this context effectively remains an open research question. A more sophisticated approach is needed to align generated artifacts with the preconditions, constraints, and dependencies inherent to the project.
3. Human-in-the-Loop Processes
   a. Maintaining human oversight and involvement is critical to ensuring the generated artifacts are both relevant and aligned with user expectations. Developing workflows that balance automation with human input is a key area for further exploration.

Path Forward

To make tools like ReForm practical and scalable, future development should focus on:

- Addressing the limitations of one-to-one requirement mapping assumptions.
- Exploring methods to incorporate broader project context into the generation process.
- Developing pre-processing techniques that improve the quality and specificity of requirements before they are fed into the generation tool.

By addressing these challenges, ReForm and similar tools could better support real-world software development scenarios while aligning with the complexities of modern technical environments.

General Workflow: Incorporating Verification and Feedback: In complex software development projects, ensuring alignment between generated artifacts and project requirements demands a structured workflow. Incorporating mechanisms for verification, human feedback, context awareness, refinements on different levels. These elements play a pivotal role in improving both the reliability of generated artifacts and the overall effectiveness of the development process.

Verification and Design-Level Alignment

Verification is fundamental to ensuring that generated artifacts adhere to global architectural requirements and specifications. This process should occur at multiple levels, including:

1. Design-Level Verification:
   a. Ensures compliance with system-wide requirements and architectural constraints.
   b. Verifies alignment with established interfaces, subsystems, and overall project goals.
2. Artifact-Level Verification:
   a. Validates generated models or code against specific requirements.
   b. Uses formal methods or automated checks to confirm consistency and correctness.

By embedding verification mechanisms throughout the workflow, the risk of inconsistencies or misaligned outputs is significantly reduced, particularly in dynamic and complex environments.

Human Feedback: A Critical Loop

Human input is essential for refining the workflow and addressing limitations in automated systems. Feedback loops enable developers and domain experts to:

- Provide design decisions that clarify ambiguities in requirements or outputs.
- Identify opportunities for refactoring, improving maintainability and aligning artifacts with long-term project goals.
- Resolve open questions about trade-offs, constraints, or alternative approaches.

Incorporating verification and feedback mechanisms into the development workflow is essential for ensuring that design artifacts align with global architectural requirements and specifications. By integrating human feedback—such as design decisions, addressing open questions, and identifying opportunities for refactoring—developers can enhance design choices and improve the quality of artifacts.

Iterative Multi-Stage and Multi-Agent Approaches

Implementing iterative multi-stage or multi-agent strategies can significantly enhance design decisions and artifact quality. For instance, the "Cocoa: Co-Planning and Co-Execution with AI Agents" [2] system introduces interactive plans that allow users to collaborate with AI agents on complex, multi-step tasks within a document editor. This approach harmonizes human and AI efforts, enabling flexible delegation through co-planning and co-execution phases.

By adopting such collaborative frameworks, development processes can become more adaptive and responsive to the dynamic needs of software projects, leading to more robust and well-aligned outcomes.

1. Iterative Workflow:
    a. Incorporates cycles of generation, verification, feedback, and refinement.
    b. Promotes gradual improvement of artifacts by systematically addressing errors, ambiguities, or omissions in each cycle.
2. Multi-Agent Setup:
    a. Specialized Agents: Different agents can be designed to perform distinct tasks, such as semantic analysis, artifact generation, verification, and optimization.
    b. Collaboration and Coordination: Agents can exchange intermediate outputs, sharing context and updates to refine the results collectively. For instance:
        i. One agent generates models based on requirements.
        ii. Another verifies the consistency of generated models with system architecture.
        iii. A third agent suggests optimizations or refactoring opportunities.
3. Dynamic Adaptation:
    a. Agents adapt based on feedback from verification and human reviewers, ensuring the workflow remains responsive to evolving requirements or project conditions.
    b. Facilitates exploration of multiple alternatives for changes, allowing the system to converge on the most suitable solution.

Maturity: The tools developed within the SmartDelta project are currently in prototype form, exhibiting limited functionality and a Technology Readiness Level (TRL) of approximately 3. This indicates that they have been validated in a laboratory environment but require significant advancements to reach production-level maturity. To enhance these tools, it is recommended to develop production-ready versions from scratch, leveraging insights gained from the research prototypes rather than building

upon the existing codebase. This approach will facilitate the introduction of new features, improve usability, and enable deeper integration into existing development environments.

Usability: To fully realize the potential of these tools, a comprehensive and general-purpose user interface is essential. This interface should streamline the entire artifact generation workflow while fostering effective collaboration between AI agents and human developers.

Key Features of the User Interface

1. Workflow Management
    a. The interface should support all stages of the artifact generation process, including:
        i. Requirements Clarification: Allow users to refine and clarify requirements, ensuring they are well-defined for the generation process.
        ii. Specification and Design Management: Facilitate the organization and visualization of specifications and design elements, maintaining alignment with overarching project goals.
        iii. Final Artifact Generation: Provide a seamless transition from design to the generation of models or code.
2. Change Impact Visualization
    - The interface should dynamically display deltas at all levels of the project, including:
        - Specifications: Highlight modifications to requirements or functional descriptions.
        - Documentation: Identify changes in textual artifacts.
        - Models: Visualize differences in state machine or architectural models.
        - Code: Show the effects of changes at the implementation level.
    - These visualizations will help developers assess the implications of new or modified requirements, facilitating informed decision-making.
3. Collaboration Between AI Agents and Humans
    a. To support the complex cooperation between AI agents and human developers, the interface should:
        i. Enable real-time feedback and adjustments.
        ii. Offer clear explanations for AI-generated outputs, enhancing trust and transparency.
        iii. Allow for flexible delegation of tasks, enabling humans to step in or modify outputs as needed.

These challenges highlight the need for continued investment in research and development to create a user interface that not only enhances usability but also transforms the way developers interact with AI-driven tools.

Summary

The current results should be viewed as initial steps toward harnessing generative AI-based technologies to address the complexities of software development. Generative AI presents promising capabilities, with many applications currently focused on simpler use cases, such as code snippet generation and chat assistance, which are functioning effectively at scale. However, the AI revolution is just beginning, and its ultimate trajectory remains uncertain. What is clear, though, is that it has the potential to revolutionize the way we develop software-based systems—a transformation that is urgently needed, given the challenges our existing approaches face in managing increasing complexity.

In the automotive industry, for example, software-related issues have resulted in years of delays, billions of euros in losses, and a decline in market revenue due to ongoing software problems.

Emerging innovations in generative AI are set to create better opportunities, fundamentally altering how we manage software projects and develop software.

The activities undertaken in the Akkodis use case are crucial to this transformation. Identifying relevant existing resources, such as software artifacts, is the first step toward effective reuse. The ability to compare, rate, visualize, and utilize retrieved artifacts for generation represents essential process steps in this journey.

Our research conducted within the SmartDelta framework (in the context of Akkodis) will be instrumental in facilitating future advancements and ensuring our participation in the global AI race. However, it is important to recognize that the implementation of generative AI-based tools in software development will differ from the SmartDelta approach. This distinction arises primarily from the rapid evolution of technologies, which necessitates the swift development of prototypes to keep pace. By learning from these prototypes and applying that knowledge and experience, we can create solutions that are well-suited for productive environments.

5. Use-Case 3 from eCAMION

a. Use-Case Description

Electric Vehicles have become more prevalent on the roads in recent years, and EV charging infrastructures have been growing rapidly to support the growth of EV cars on the road. EV charging infrastructure is a developing area of research and development, with IoT technology enabling real-time communication between the EV charging unit and the Charging Station Management System (CSMS).

At eCamion, real-time data collected from the charging stations are used in three different ways to provide additional insight to the charging station operators.

1. Charging Station Health Monitoring

Using sensor data collected from the charging station and its battery, operators can monitor the availability, anomalies and battery's health down to the cell level.

2. Charging Station Usage Monitoring

The dashboard offers an overview of charging station usage, presenting key metrics and insights to help estimate the station's popularity.

3. Energy Consumption Rate Prediction

Using the load profile of each charging station, the dashboard provides the predicted energy consumption rate at hourly intervals.

As the EV charging industry rapidly evolves, eCamion's charging stations are expected to undergo continuous development to enhance communication, energy efficiency, and functionality. Specifically, in collaboration with SmartDelta, we have considered the development of a Charging Station Management System (CSMS) and a charging station analysis dashboard.

The CSMS serves as a central server that communicates with charging stations and collects real-time performance metrics. Its development is fast-paced, driven by an active industry community and ongoing enhancements from eCamion's development team. Through our collaboration with SmartDelta, we are also evaluating the improvements introduced with each CSMS implementation to assess their impact and effectiveness.

b. Link to SmartDelta Methodology

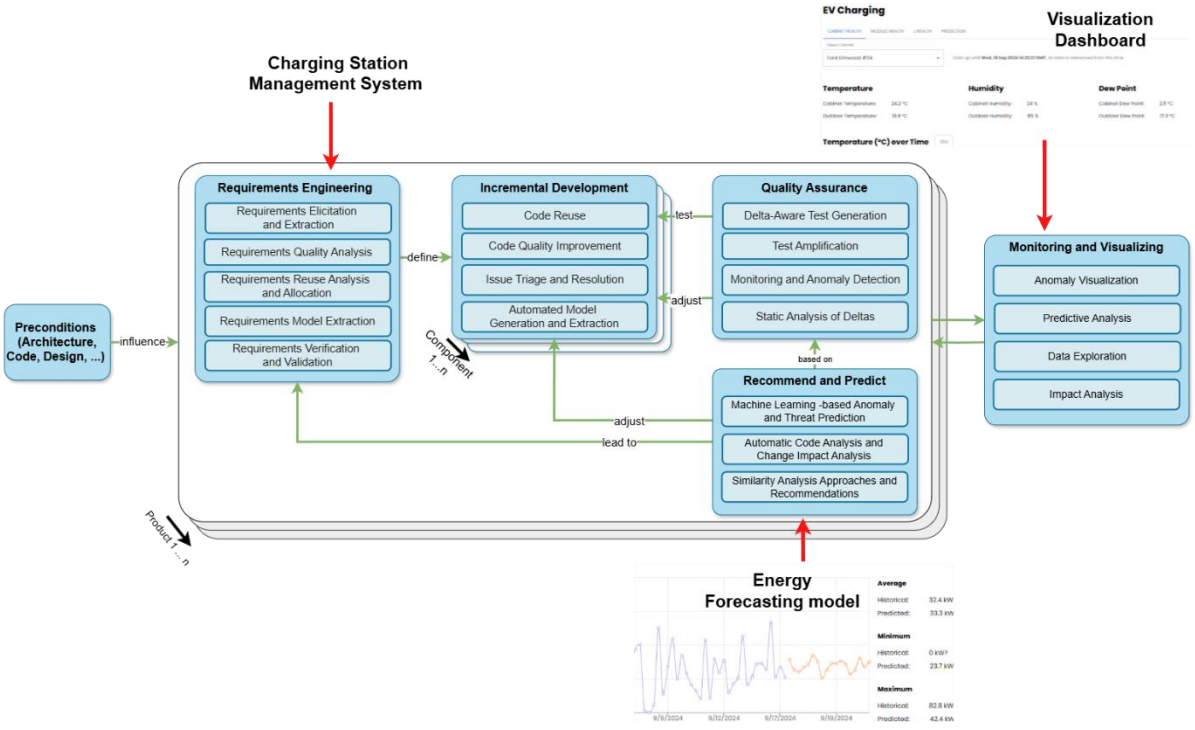


Figure 37: eCamion and the SmartDelta Methodology

Using SmartDelta methodology, we have established KPIs to track the quality of subsequent product variations. Following the SmartDelta methodology, our development focuses on recommendation and predict where our ML model is applied to predict the energy load of a charging station. Based on the evaluation KPI of the model, suggestions can be made to apply different models. Lastly, the predicted output from the model, along with other charging station usage and sensor data are visualized using the management dashboard.

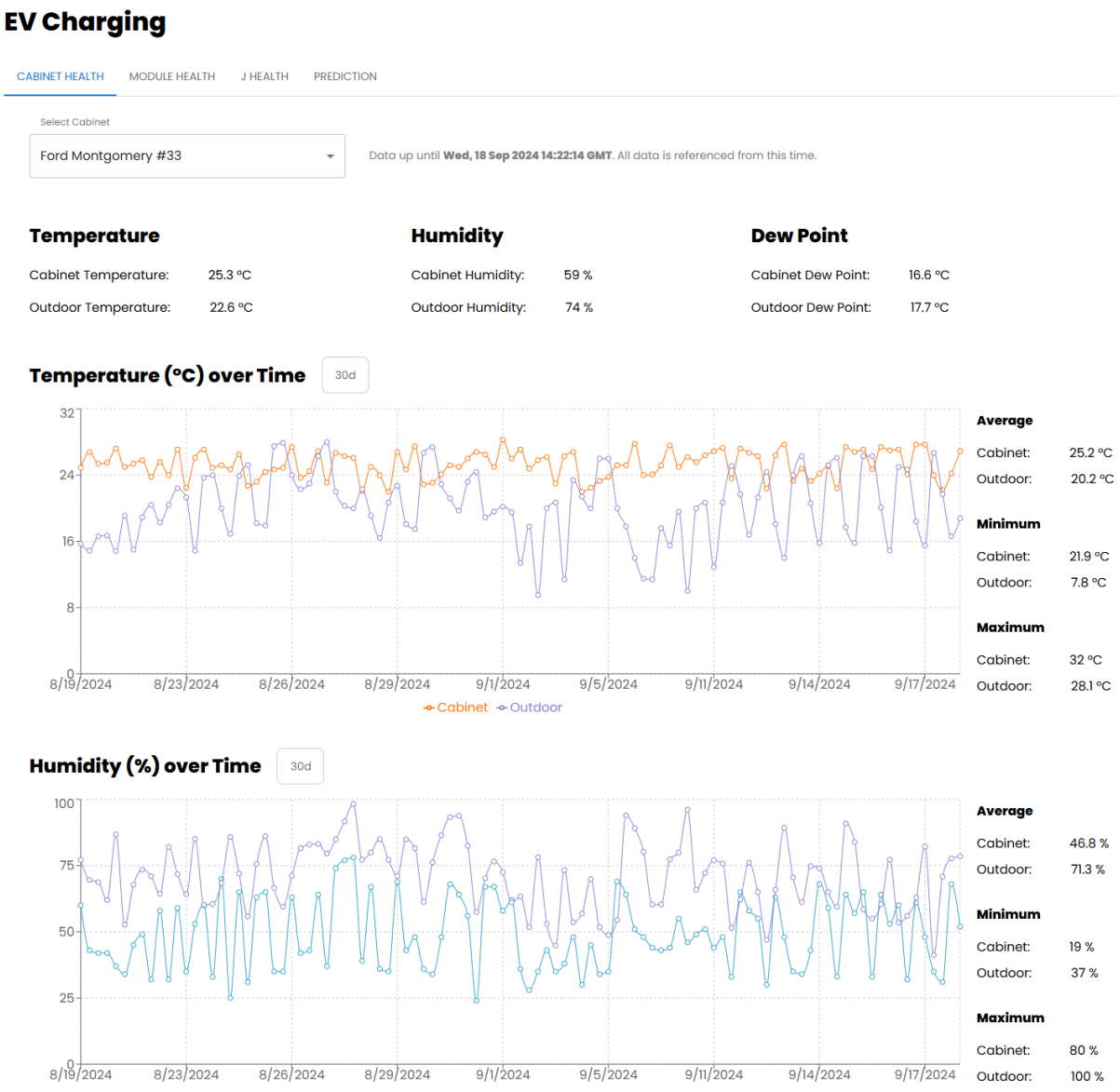
c. Tools descriptions

Our management dashboard analyzes the health and performance of charging stations. Commercial charging stations are often installed outdoors, exposed to weather and other environmental factors. Swift action in response to anomalies is crucial for effective maintenance. Using real-time data from the station's sensors, owners are notified of temperature or voltage anomalies, enabling proactive inspections and preventing failures.
Tracking charging station performance is essential for future development and cost-saving measures. The dashboard provides insights such as station popularity, usage patterns, and predicted trends, helping owners make informed decisions.

d. Visualization

The visualization features planned for the tool aim to present a comprehensive view of system health through historical data and detailed summary statistics. This includes measurements of key physical parameters at charging sites and stations, such as component temperatures, humidity, dew points, voltages, and power delivery. Additionally, outdoor readings at corresponding site locations are incorporated to evaluate the impact of external environmental factors on the system's physical characteristics.

In its current stage of development, the Figure 38 below represents a few examples of the visualizations intended for inclusion in the final version.



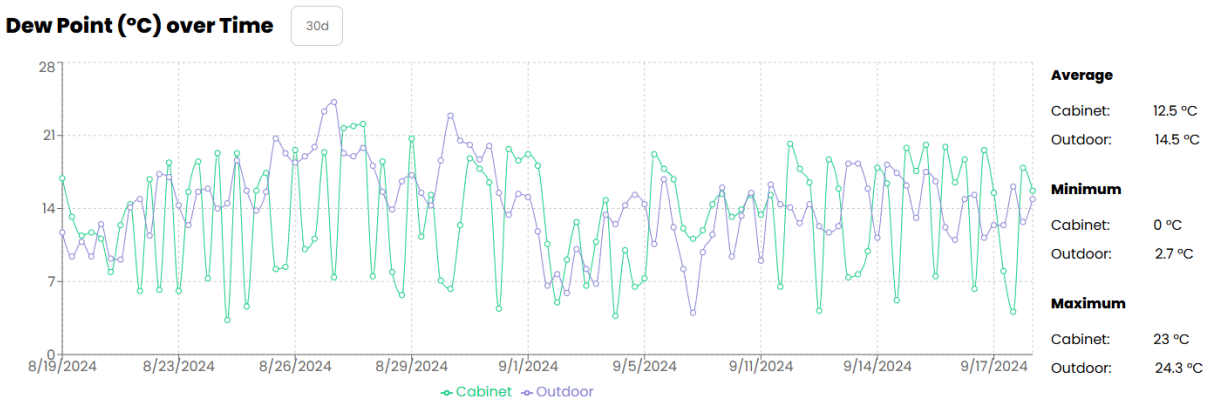


Figure 38: eCamion's visualization examples

Dashboard solution

The data collected to characterize various system health parameters is stored within a PostgreSQL database. Raw data from the database tables are queried, processed, and prepared for use by a suite of Python scripts and modules. These scripts are utilized by a Python Flask server, which functions as the system's backend for the dashboard tool.

The dashboard's frontend is built in NodeJS and leverages the ReactJS framework. Several JavaScript libraries have been employed for visualization components, including MaterialUI, ReCharts, and ChartJS, with additional libraries expected to be integrated as development progresses.

Visualisation requirements

The dashboard visualization requirements focus on providing clear, actionable insights across various aspects of system performance and usage. Key elements include:

Forecasted Demand:

- Visualizing the demand forecast for the next week, showing maximum, minimum, and peak demand hours.
- Including uncertainty bands and allowing room to easily plug in additional metrics or data sources with features like tabs for better organization.

System Health:

- Monitoring temperatures for cells and cabinets, including average, max, and min values over time, compared with external temperatures.
- Flagging unusual temperature deviations and analyzing cell voltage spreads to identify modules that most reduce station capacity.

Station "J" Insights:

- Tracking the number of sessions and cumulative session times to gauge popularity.
- Reporting power delivery stats and flagging deviations in max or average power.
- Incorporating temperature analysis similar to the system health section.

Environmental Context:

- Adding historical weather data (like temperature and humidity) for each site and comparing it to internal humidity sensor readings in cabinets.

The goal is to deliver a dashboard that's not only scalable but also intuitive, helping users identify trends and anomalies more easily.

e. Evaluation Setup

The evaluation of the tool focuses on availability and latency between the communicating systems. The assessment considers three aspects of charging station functionality: communication latency between the Charging Station Management System (CSMS) and the charging station, authorization latency, and the availability of the dashboard tool.

To measure communication latency, the CSMS was configured with an SQL database that records charging station data obtained through the communication protocol. By comparing the timestamps of the charging station's authorization request and the recorded payment time, the requirements for UC3.FR4, FR5, and FR17 were met. Similarly, the EV charger's availability status was compared, meeting the requirements for UC3.FR8 and FR9.

The latency of price settings was measured through a manual test using a single charging station. Price changes were applied, and the timestamp reflecting the change in the CSMS was recorded, ensuring compliance with UC3.FR6, FR10, and FR11.

eCamion's charging station provides sensor data indicating cell and cabinet health. This data is sent to the CSMS through configured messages and recorded in the SQL database. By comparing the timestamps of records from both the charging station and the CSMS, transmission latency was measured, satisfying UC.FR17.

To evaluate the availability of the visualization dashboard, which is hosted on AWS Kubernetes, AWS downtime metrics were referenced to obtain data for UC.FR1.

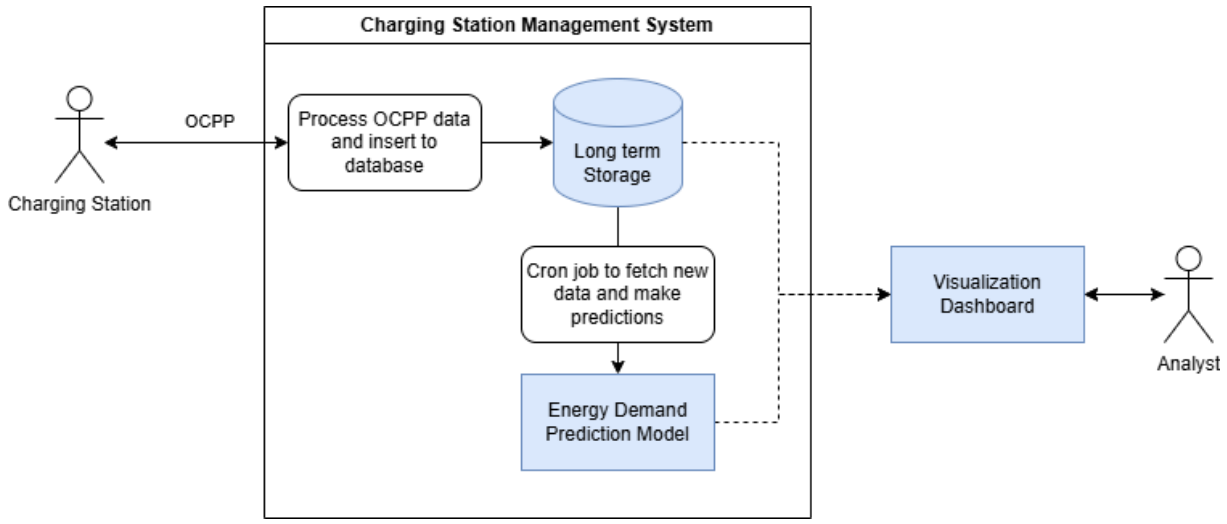


Figure 39: Charging Station Management System

f. Evaluation results

Requirement	Tool	Solution partner	KPI Definition	KPI Base Values	KPI Target Values	KPI achieved Value
UC3.FR4 UC3.FR5 UC3.FR17	Tool A	eCamion	The EV charging authorization (non PnC) latency should not exceed 30 sec after the user has completed payment at the terminal The EV charging authorization must have at least a success rate of 99% to ensure the quality of service	2 min	30 sec	15.0 sec
UC3.FR8 UC3.FR9	Tool A	eCamion	The EV charger availability status must reporting latency no more than 20s The reservation must have a latency for no more than 15s	2 min	15s	13.5 sec
UC3.FR6 UC3.FR10 UC3.FR11	Tool A	eCamion	The monetary transaction details must be reflective to the pricing setting with a max age of 2 min	5 min	2 min	2.418 sec
UC3.FR17	Tool A	eCamion	Considering the real-time charger status reporting for eCAMION's JuleNet operation, the latency shall not exceed 1 min	5 min	1 min	15.0 sec
UC3.FR1	Tool A	eCamion	It must be made for a high available application (max annual down time </= 5 min)	60 min	5 min	0 sec (no downtime recorded)

UC3.FR4, UC3.FR5, UC3.FR17: The authorization latency of the EV charging station is measured from the time the client completes payment to the start of energy delivery. With an initial KPI target of 30 seconds, our system recorded a median latency of 15 seconds.

UC3.FR8, UC3.FR9: The KPI accounts for latencies in both charger reservation and charger availability. Since a successful reservation implies availability, the KPI was measured for charger availability. With a target of 2 minutes, our system recorded a median latency of 13.5 seconds.

UC3.FR6, UC3.FR10, UC3.FR11: The KPI measures the latency between a price change made in the Charging Station Management System and its application at the EV charging station. Measurements were conducted manually, with 10 trials performed on a charging station, and the average latency was recorded.

UC3.FR17: The KPI ensures that all charging station statuses are recorded by the Charging Station Management System. With a target latency of 1 minute, we observed a median latency of 15.0 seconds.

UC.FR1: The dashboard tool should have no more than 5 minutes of downtime annually. Our evaluation of the AWS Kubernetes environment, where the dashboard application is hosted, found no recorded downtime.

g. Recommendation for industry adoption

The dashboard provides potential in health monitoring and AI prediction to improve charging system performance. In health monitoring, using the dashboard, the analyst can flag the charging system based on issues such as:

- Temperature deviation which could indicate fan failure, voltage spreads causing reduced capacity
- Low system voltage due to insufficient power
- Water damage caused by high humidity or low dew points

These diagnostics enable faster response and repair, minimizing system downtime.
Using the AI prediction trained on charging station usage data, the prediction adds value by identifying peak hours and expected demand, allowing for external battery to be charged during the off-peak hours.
The current limitation of the dashboard is that health monitoring would identify issues only after they occur. However, further work can be done to incorporate AI to predict the faults before the failure, thus increasing the availability of the system.

The SmartDelta methodology has been a valuable framework for our EV charging station infrastructure development, particularly in an industry that demands rapid innovation. Given the fast-paced evolution of EV technology and the necessity for frequent enhancements to remain competitive, SmartDelta provides a structured approach for evaluating each new variation of our product.

By leveraging SmartDelta's methodology, we ensure that each incremental change or new feature is assessed for its impact on system performance, reliability. This approach enables efficient iteration and validation of functionalities without compromising system stability.

6. Use-Case 4 from NetRD

a. Use-Case Description

CPaaS Overview: Platform as a Service (PaaS), which is one of the service models of cloud computing, is a cloud-based application development platform, so that service providers can create, develop, and deploy their applications instead of knowing the resource utilizations by their applications. In the PaaS model, network, server, storage, and other services required for the customer's application development processes are provided and maintained by cloud provider[3].

CPaaS, is based on PaaS, is a cloud-based development platform that allows developers to embed real-time communication services such as video, chat, and voice to their services. In this model, application developers do not need to build their own backend infrastructure for communication stack. It offers a development framework to build real-time communication features by employing APIs (Application Programming Interface) and integrated development environments.

[3] C.M. Mohammed, S. Zeebaree, "Sufficient Comparison Among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review", International Journal of Science and Business, 5, issue 2, p. 17-30, 2021.

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[4].

Problem and Challenges: Debugging and problem addressing process varies according to the components in the CPaaS platform, which contains many microservices and consists of many different components. Table 3 provides some sample issues in CPaaS and the components where these errors were observed. In addition, the time spent by the operations engineers for addressing and fixing errors is given in *percent*. Although these values vary according to the experience of the engineers in charge, they have been obtained from the *Jira* data where the CPaaS project is managed.

Table 3: Example Fault Components and Addressing/Recovery Durations

Fault Reason	Fault Component and Effort to Fix	
	Component	***Effort***
Disk Usage	Docker, GlusterFS, Elasticsearch	10%
Memory Usage	System, containers, microservices etc.	10%
Network	Keepalived, Weavenet	10%
Routing	Kong, Nginx	20%
Container	Docker, Harbor, Kubernetes	11%
Databases	Postgresql, Redis, Mariadb, Minio etc.	9%
Services	Microservices	22%
Other	Ansible, Rabbitmq, Hazelcast etc.	8%

According to the Table 3, 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.

Debugging Approach for User Scenarios: Figure 40 presents a flowchart of a DevOps engineer's approach to troubleshoot observed errors in user scenarios. After Kubernetes system health check and failed API identification, the next step is to examine the user's REST request and check routing rules on the gateway. Then, starting from the logs of the first microservice to which the request was sent, a log analysis is performed in a chained manner, considering the interactions with other microservices. The biggest handicap in this approach is the time it takes to detect the interaction between microservices. The fact that the values given for *Routing* and *Services* in Table 3 have the highest time spent proves this handicap.

[4] K. Aktaş, H.H. Kilinc, N. Arica, "Microservice Interaction Prediction in Communication Platform as a Service," 30th Signal Processing and Communications Applications Conference (SIU), Safranbolu, Turkey, 2022, pp. 1-4.

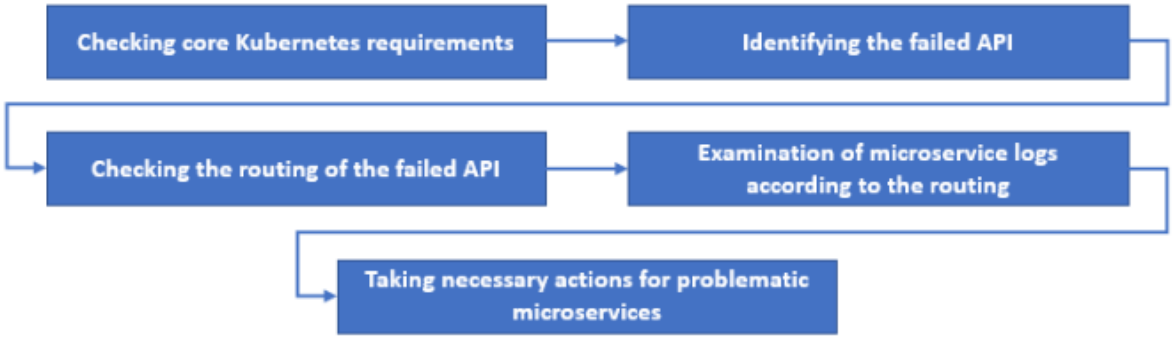


Figure 40: DevOps engineer's debugging approach for user scenarios

b. Link to SmartDelta Methodology

Within DIA4M there is "Delta" tracking in both time (in the sense of version) and space (in the sense of variant). It allows to compare deltas for two different versions of the same microservice and for two different microservices.

- Log comparison for microservice versions and variants
- Comparing resources such as memory, cpu for microservice versions
- Latency and trend comparison for microservice versions

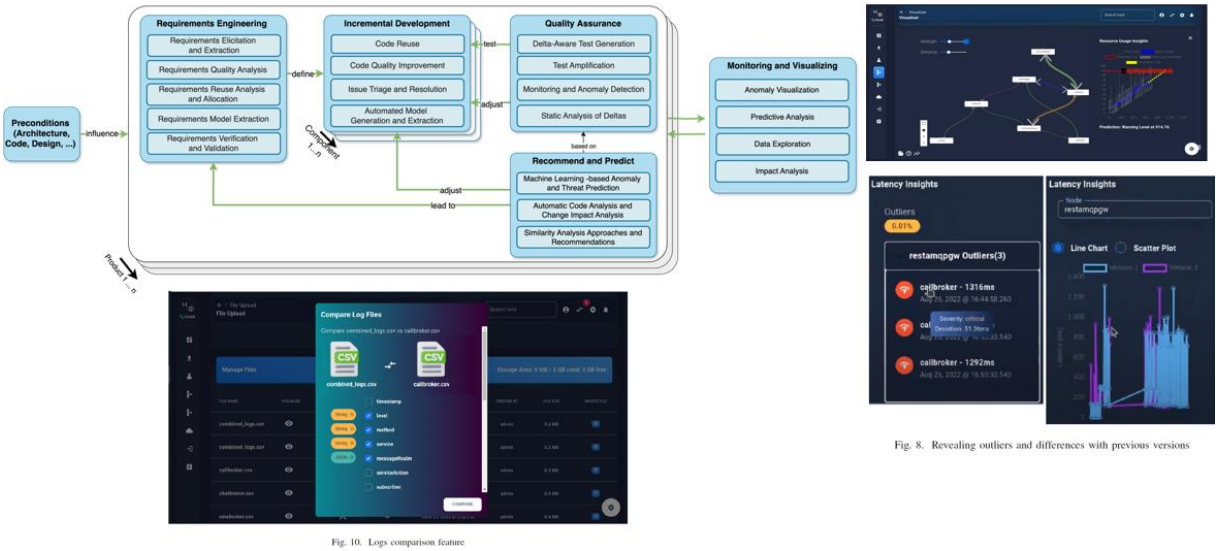


Fig. 8. Revealing outliers and differences with previous versions

Fig. 10. Logs comparison feature

Figure 41: Position of the DIA4M tool in the SmartDelta Methodology.

c. Tools Descriptions

DIA4M (Discovery of Interactions and Anomalies for Microservices) is a tool which have been developed from scratch.

DIA4M is a web-based tool designed to enhance the efficiency of DevOps engineers managing cloud-based distributed platforms. The tool focuses on mapping microservice interactions and quickly identifying anomalies and faults, leveraging advanced analytics to automate processes and address issues in CI/CD processes.

DIA4M aims to provide a comprehensive toolkit that leverages advanced statistical methodologies to detect critical issues, anomalies, and hard-to-find errors throughout DevOps processes. With Elastic Cloud Infrastructure that leverages APM agents having auto-instrumentation capabilities for logging, tracing, and error detection, DIA4M is designed to handle high event rates in both real-time

distributed systems and systems with many users[5]. The primary goal is to automate monitoring data collection for Continuous Verification, reducing error rates and improving response times without introducing additional layers of complexity to already intricate systems. One of its most crucial goals is to minimize troubleshooting time for DevOps engineers by employing state-of-the-art visualizations and adhering to best practices in UI/UX design. Moreover, DIA4M aims to offer analytic services at no cost, thereby supporting the open-source community and amplifying its overall impact.

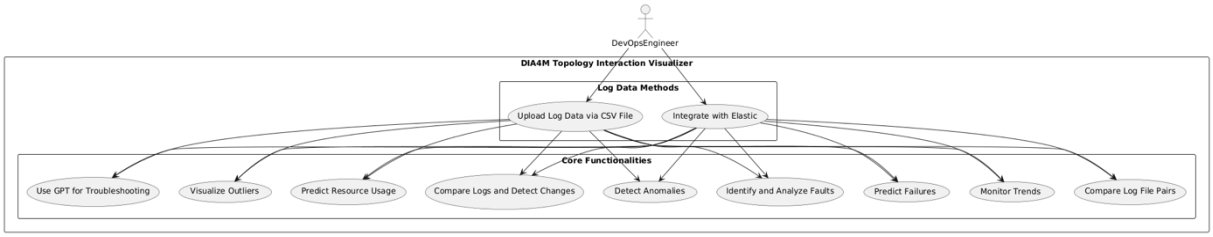


Figure 42: Conceptual topology of DIA4M.

The tool evaluates quality attributes by analyzing microservice log data to extract patterns and understand interactions, enabling the identification of anomalies and faults efficiently.

Figure 42 represents the usage scenarios for the DIA4M Topology Interaction Visualizer. The primary actor is the DevOps Engineer, who can interact with the system through two main methods: uploading log data via CSV files or integrating with Elastic Cloud. Each method enables the engineer to perform various core functionalities, such as visualizing outliers, predicting resource usage, comparing logs, detecting anomalies, identifying faults, predicting failures, monitoring trends, comparing log file pairs, and using a GPT-based troubleshooting guide.

d. Visualization

DIA4M has many modules and features. Visualization is done according to the requirements. The four key features and visualizations that distinguish the tool from others are as follows.

Service Mapping Feature: The service mapping feature includes handling file uploads, reading the CSV file, extracting columns, filtering data, generating output with nodes and edges to be visualized and used as input in other analytics related modules. As shown in Figure 43, DIA4M employs a drill-down approach in its service mapping, using node-based visualizations to adhere to user experience principles by avoiding information overload in a single view. It offers the ability to adjust the strength between nodes using a slider interaction.

Compared to other well-known service maps, a unique feature of DIA4M's mapping visualization is its flexibility. As the number of services and their dependencies increase, the mapping still maintains its clarity for DevOps engineers by offering the ability to change the distance between nodes using a slider interaction.

[5] Elastic. Elastic apm documentation. Accessed: 10-Dec-2024. [Online].
Available: https://www.elastic.co/docs/current/integrations/apmhow-to-use-this-integration

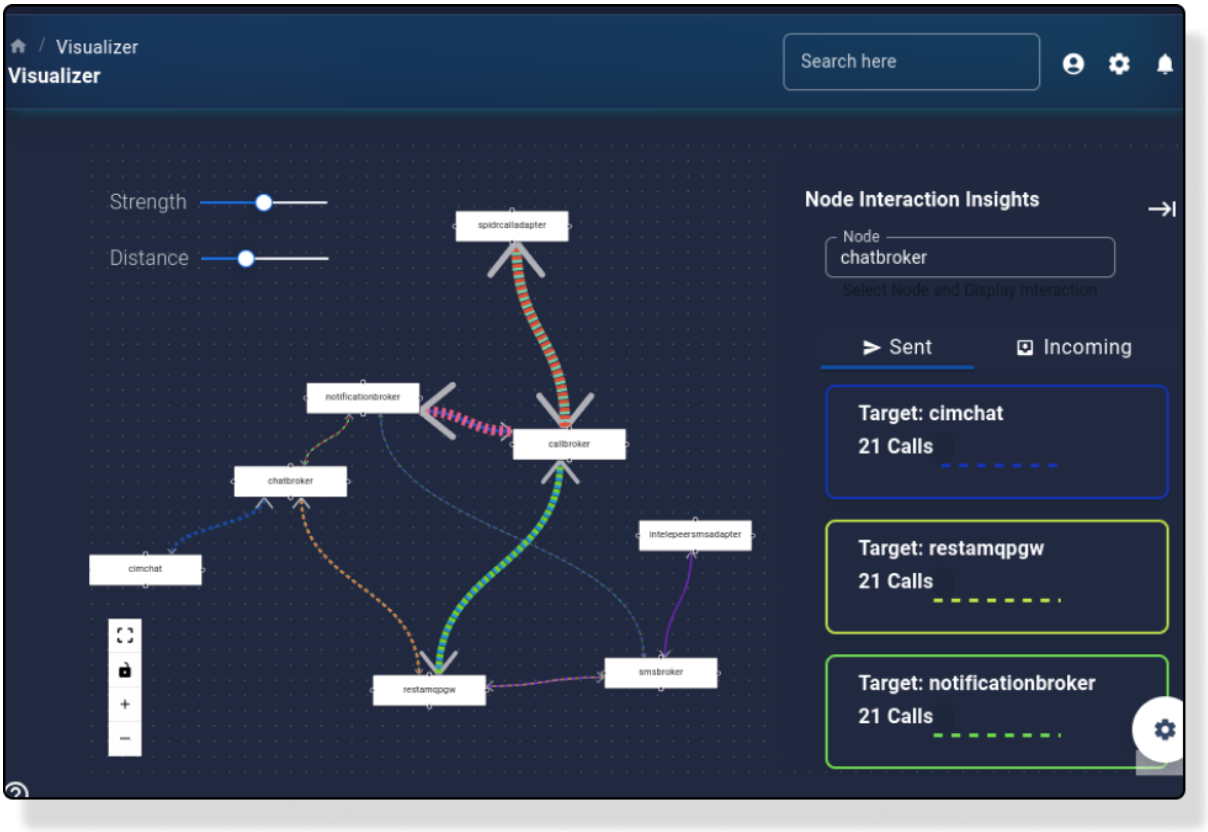


Figure 43: Service mapping by log file

Anomaly Detection Feature: Anomaly detection is an important aspect of system monitoring, especially to identify irregularities in log data that may indicate possible system failures or security breaches. This process uses a mixture of statistical methods and domain-specific heuristics to improve detection accuracy. Anomaly scores derived from logs guide DevOps engineers in identifying areas that require attention, as shown in Figure 44.

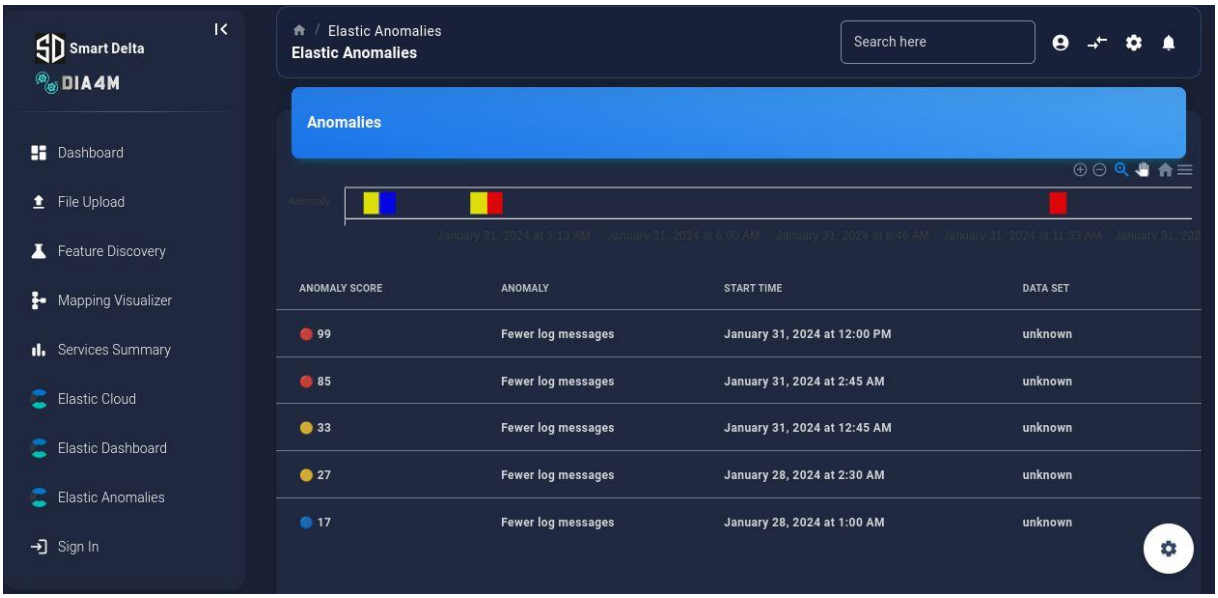


Figure 44: Anomaly detection with scoring

Service Health Feature: The Services Health Summary UI in DIA4M redefines the way DevOps engineers analyse microservice performance through a visually engaging table with embedded line

graphs as shown in Figure 45. This interface provides a comprehensive view of critical metrics for each service.

Figure 45: Each service`s health status in a single view

Logs Comparison Feature: The Log Comparison feature facilitates the analysis of existing logs in relation to those in previous versions and enables users to discern changes and patterns in log data. Using the "Horizontal Comparison" method, the tool clusters messages based on their instances in each log file and then compares these clusters, highlighting differences and similarities. This approach allows for a detailed examination of log data and supports selective analysis of specific log areas of interest, increasing the ability to effectively understand and monitor log changes, as shown in Figure 46.

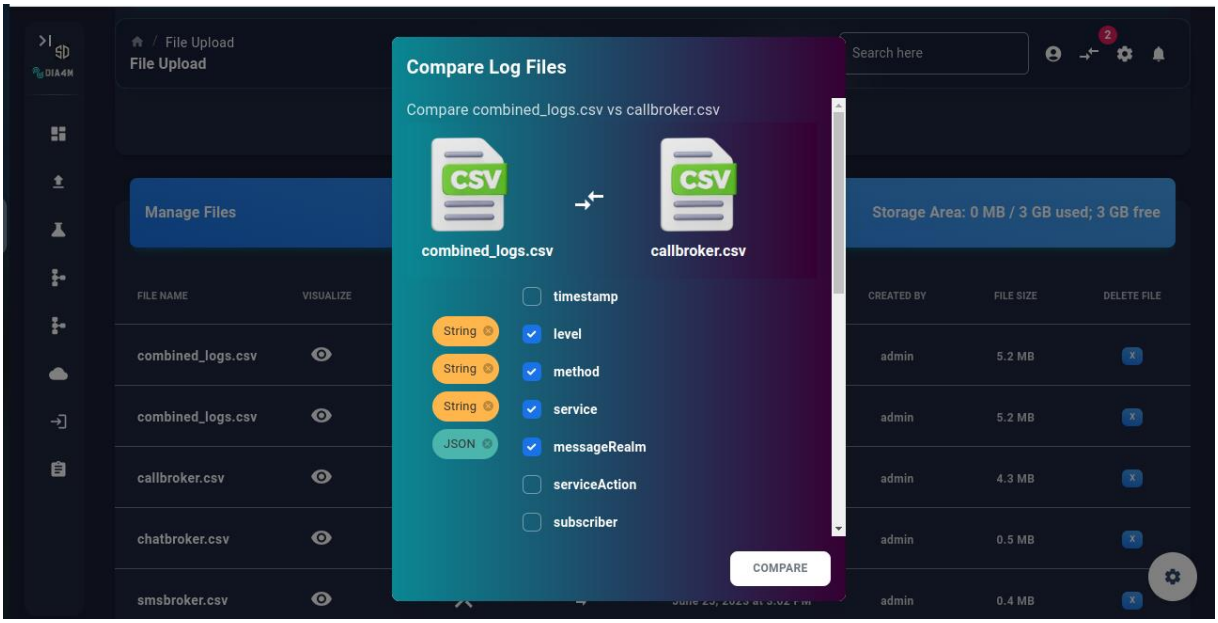


Figure 46: Logs comparison feature

Visualisation requirements

There are 9 functional requirements for this use case. The 10th requirement is related to the visualization of the first 9 FRs.

Requirement	KPI Definition	KPI Base Values	KPI Target Values	Comments
UC4.FR10	We aim to live the best customer experience	%20 satisfaction	%100 satisfaction	Solution should be able visualize the below: Topology of microservice interactions (relates UC4.FR1) Predicted interactions for previous and next hops (relates UC4.FR2) Faults and anomalies (if exist) in the predicted microservices (relates UC4.FR3) Faults and anomalies (if exist) in the extracted topology (relates UC4.FR4) Deployed microservice's state in the cluster (relates UC4.FR5) Comparison of last 2 image sizes of the same microservice (relates UC4.FR6) Running microservice versions in the cluster and their releases (relates UC4.FR7 and UC4.FR8) Resource utilization of each microservices (individually), nodes in the cluster and system components (UC4.FR9)

Figure 47: Visualization requirements

e. Evaluation Setup

DIA4M users can easily select their microservice monitoring type with these three selections:

- Directly from Kubernetes Cluster
- Via a Cloud Provider (Amazon Web Services, Google Cloud, or Microsoft Azure)
- Via Localhost

Additionally, four different monitoring systems are available for better visualization and analysis for Cloud Provider and Localhost selections, which are Prometheus, SigNoz, Zabbix, and Datadog.

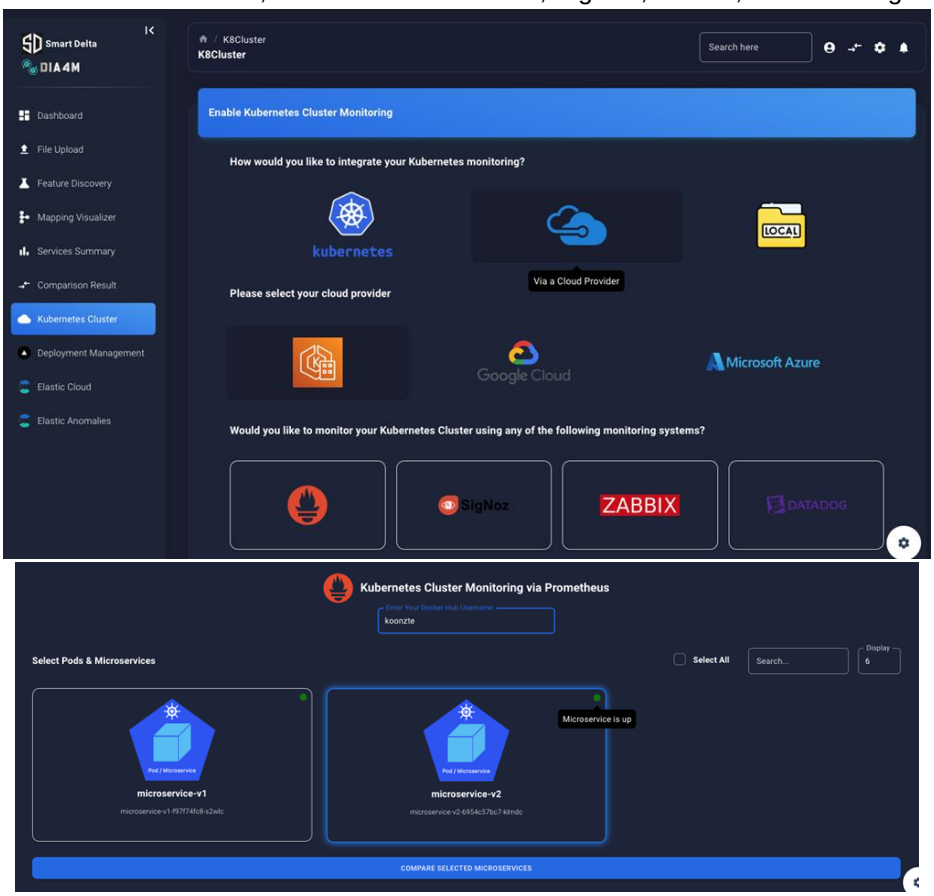


Figure 48: Evaluation setup environments

In addition, to prove whether the microservices are running after new version deployments, we used Vercel Deployment Management as a third way.

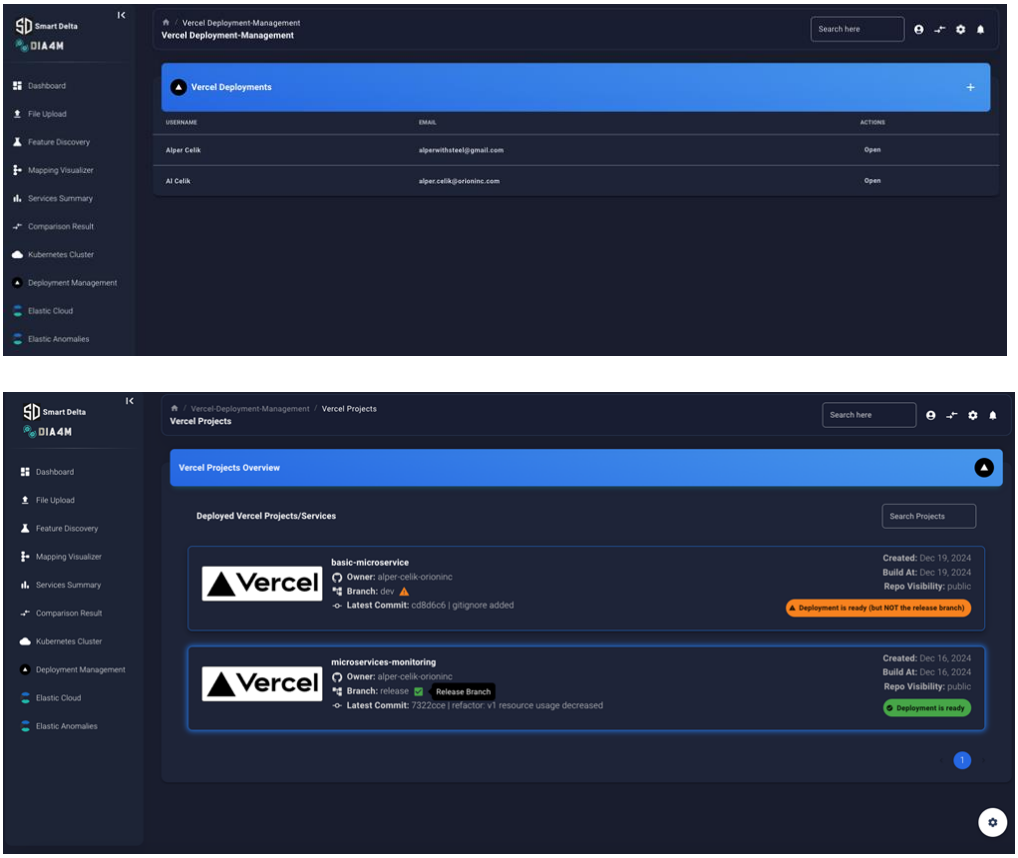


Figure 49: Vercel setup environments

f. Evaluation results and Recommendation for industry adoption

The evaluations and recommendations made by the DevOps team are as follows. Only the points where additional development is requested, and recommendations are reported here. **General evaluations and KPI targets are reported to have been met.**

Product Implementation Sample#1: CPU & Memory Metrics for Microservices

CPU usage, memory usage and image sizes of microservices can be observed in a single, dynamic designed, modern looking dashboard. Any microservice can be selected in CPU Usage and Memory Usage tables for comparison. Also, their time ranges can be adjusted from “last 30 minutes” to “last 1 year”. Further, zoom in and zoom out features exists for checking detailed analysis for DIA4M users. All selected microservices’ cpu and memory usages can be seen as timestamp log files format in the tables below. In addition, all selected microservices’ docker image sizes, which shows their diskspace, are observable.

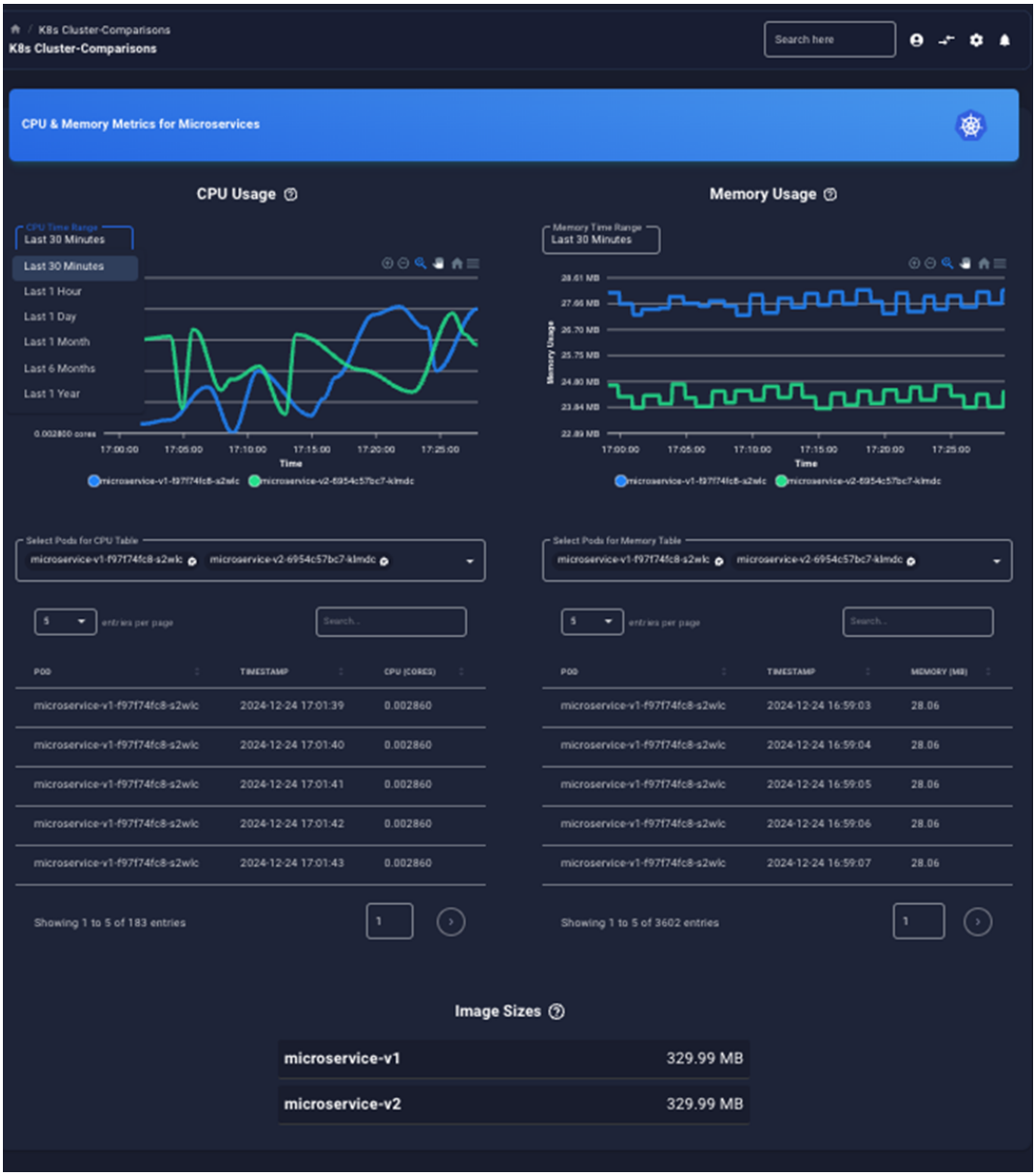


Figure 50: CPU usage, memory usage and image sizes of microservices

Recommendation#1: Heap memory usage and memory breakdowns can be added. Typically, heap memory usage refers to the memory (RAM) usage of virtual machines in an environment using hypervisor or virtual machine technologies. The hypervisor manages the memory usage of the system and balances the amount of memory allocated to each virtual machine with the total memory capacity of the physical machine.

Product Implementation Sample#2: Vercel Deployment Management for Release Branch Tracking

The Vercel Deployment Management module in the DIA4M Tool provides an efficient and secure way to manage and monitor microservice deployments. By leveraging Vercel's free and user-friendly platform, this feature simplifies the deployment process, particularly for tracking release branches. This integration allows users to observe the deployment status of their microservices and ensures that non-release branches are flagged with warnings to maintain proper version control.

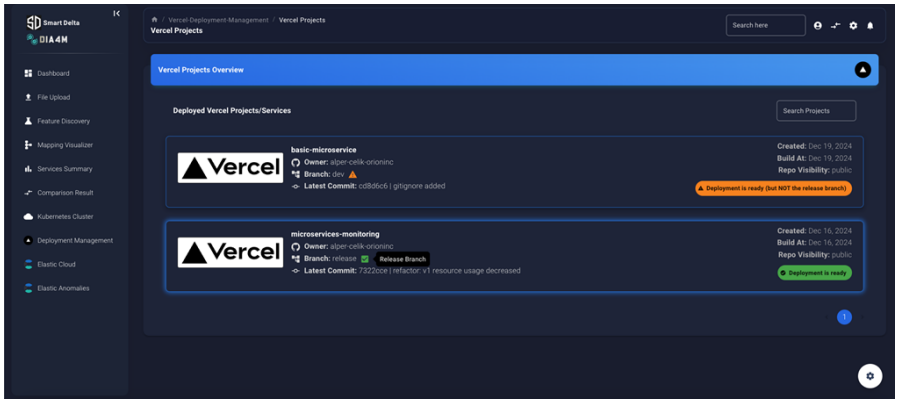


Figure 51: Vercel Deployment Management Module

Recommendation #2: Information about where the branches are deployed can be added. This information can be retrieved from the pipeline config. Targeted IP address can be shown, and GIT link can be added to Latest Commit.

Product Implementation Sample#3: Anomaly Detection Module

The Service Anomaly Detection module aims to enhance monitoring capabilities by implementing a user-friendly and dynamic anomaly detection system. The system provides insights into service metrics and anomaly scores, allowing users to identify and respond promptly to unexpected deviations in performance.

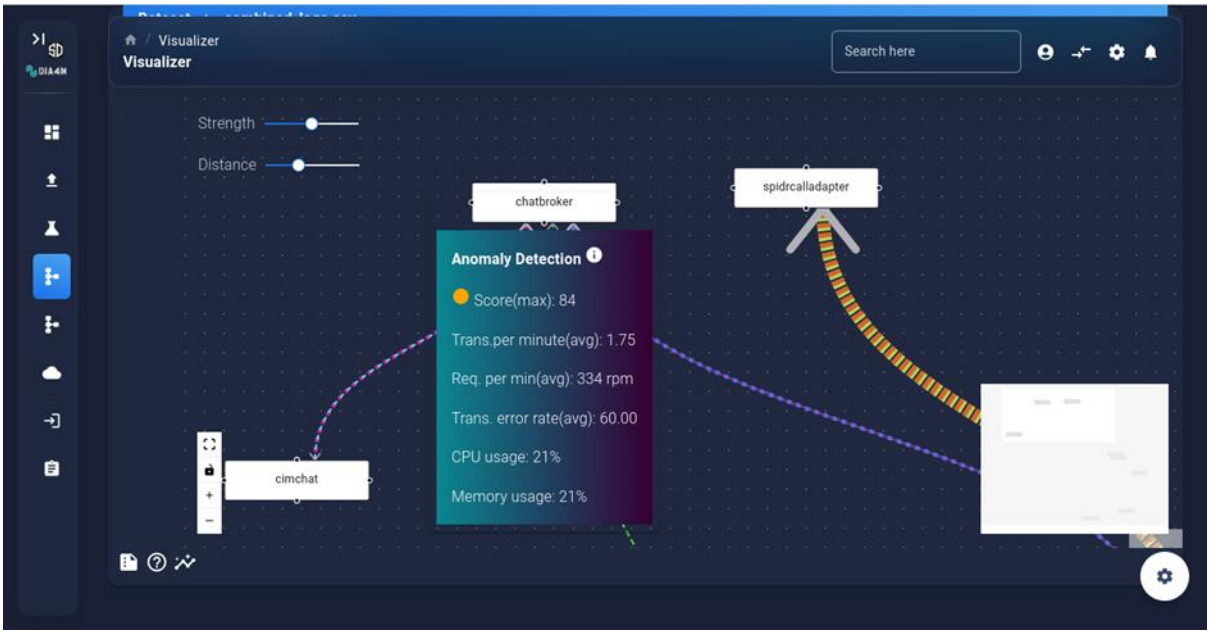


Figure 52: Anomaly Detection

Recommendation #3: There should be a notification mechanism such as email and sms for anomalies. These notifications should be configurable according to the anomaly score. In addition, a link should be added to access the logs related to the anomaly.

General Recommendation: The developed solution is a successful solution that analyzes and monitors microservices in detail. The solution can be developed to give and receive recommendations for small actions. For example, it should be able to give suggestions such as “restart the system if you see this alarm”.

7. Use-Case 5 from Kuveyt Türk

a. Use-Case Description

Banking Systems Overview: Banking systems are platforms that store all customer, account, transaction, and system configuration information for a bank. Unlike other systems, transactions are extremely fast, and system and service continuity is nearly 100%. Additionally, data consistency and high levels of system and data security are essential. Another distinction is that banking services can be accessed through distribution channels such as ATMs, the internet, and mobile devices. With the widespread use of these channels, the importance of continuity and security has increased.

Banking System Worked On: The BOA Banking Platform enables the entire banking system, including channels, to operate on a single platform. This integrated system not only speeds up processes but also reduces IT operation and investment costs, offering a significant advantage, especially in new product development.

BOA is a comprehensive banking platform that allows all necessary banking functions to operate on a single platform. Currently, it is actively used by six banks and two financial institutions. It incorporates the participation finance knowledge and corporate capabilities that Kuveyt Türk has accumulated over more than 30 years. Additionally, it has been developed by a highly experienced technology team. It is the most up-to-date and stable banking software package actively working in the market. It is continuously updated to meet customer needs, global trends, security requirements, and rapid technological advancements. Approximately 650 qualified engineers are actively working on the product, ensuring it remains a robust and current banking platform.

The platform is currently active in Kuveyt Türk, Emlak Katılım, Vakıf Katılım, Destek Bank, Golden Global banks, Turkcell Finansman, and TOM Digital companies in Turkey, as well as KT Bank AG in Germany. Some modules are also operational in Kuwait Finance House in Kuwait. It is a banking software product approved by the Turkish Banking Supervision and Regulation Authority (BDDK) and the German banking regulator BaFin.

The platform includes modules for core banking, loans, treasury, foreign trade, international banking, payment systems, CRM, campaign management, financial control, accounting, digital banking (mobile (iOS, Android), internet), direct banking, branchless banking, ATM, XTM, call center, telephone banking, corporate integrations, collections management, human resources, administrative services, purchasing management, mobile sales automation, budget management, artificial intelligence models, and all other modules a bank might need. It continues to serve more than 10 business channels with over 10,000 screens, more than 1,500 business intelligence reports, and over 1,200 business processes.

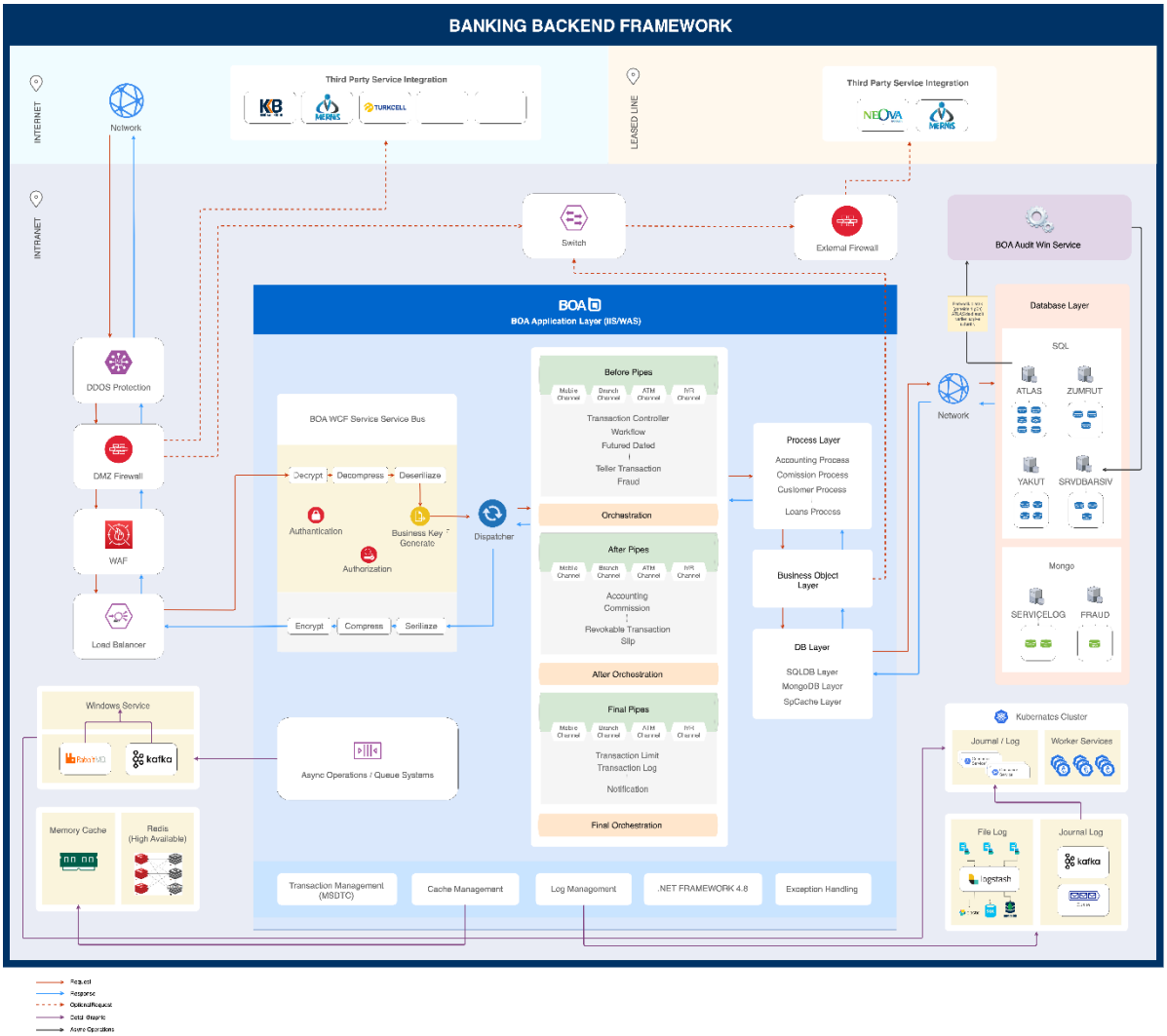


Figure 53: Banking Backend Framework

In addition to all these features, BOA can be considered one of the pioneering low-code platforms. Its infrastructure modules are entirely definition-based, making it very easy to develop new modules or update existing ones. The application architecture is straightforward and easy to understand. It has been developed by incorporating the best practices from PMI, Agile, COBIT, ITIL, Owasp, VISA, MasterCard, ISO, and other international standards and global experiences. It does not require purchasing additional licenses to operate. The application update (DevOps) cycle is simple and is managed with the software that comes with it. BOA includes an application called BCP (BOA Change Protocol) for data model and application updates. Data dictionary and data model management, as well as software development lifecycle management (SDLC), are handled through its internal modules. Errors and performance issues that arise during operation are automatically detected, and a bug report is generated for the relevant developer through the responsibility management system to resolve them. Source codes are continuously scanned using CAST and SonarQube applications, and critical findings are addressed to ensure ongoing improvements in code quality. Its modular structure allows for easy installation.

One of the primary advantages of BOA is that it simplifies complex banking transactions into a plug-and-play format in the digital environment. Its simple structure allows for quick setup from scratch, and the operating cost is very low. It is also an easy system to learn.

Additionally, BOA has a comprehensive infrastructure that ensures business modules operate stably, securely, and with high performance. Business process management, digital document

management, central authorization, logging, scheduled task management, definition-based application infrastructures (such as accounting, commission, receipt, etc.), information management, corporate mobile application platform, in-office document management (BOA Cloud), central transaction management (limit, authorization), and fraud management infrastructure modules all work in a definition-based and performance-oriented manner.

One of the most significant advantages of the BOA platform is its security. Leveraging the best international experiences, security analyses and tests are conducted separately for each module and channel. The security analysis and testing process is naturally integrated into the entire development process. Advanced NT, SSL, and multi-factor authentication methods are active options in BOA. Oauth2 features include data encryption, single-point authentication, and regional IP restriction.

Another advantage is the emphasis on performance at all levels of BOA. It is designed to be easily scalable, and areas that could cause bugs and performance issues are made accessible for developers. Thanks to these superior features, in its early years of development, BOA achieved the highest number of transactions per second, with 14,200 transactions per second, in tests conducted at Microsoft Redmond laboratories, setting a world record.

The BOA™ brand represents the application software infrastructure that forms the core of banking applications developed by Kuveyt Türk. The Core Banking Platform (BOA) features a 3-tier architecture (database servers, application servers, and branch servers). The client side includes WPF, React, Android, iOS, and ASP.NET, while the middle tier uses C# (WCF), and the data tier uses MSSQL.

Motivation: Monitoring code quality and security in the banking system is crucial, especially given the recent cyber-attacks targeting banks with security vulnerabilities. Ensuring the continuity of the banking system is therefore essential.

Typically, banks that offer individual services aim for an average continuity rate of 99.99%. To achieve this, they allow for a maximum of 52.56 minutes of service interruption per year. This 99.99% continuity rate, often referred to as "four nines," equates to 52.56 minutes of downtime annually. A slightly lower continuity rate of 99.9%, known as "three nines," translates to 8.76 hours of downtime per year, which is considered quite lengthy for the banking sector. In 2021, the total downtime across all systems amounted to 480 minutes, with 180 minutes attributed to A Class interruptions (affecting all channels) and 300 minutes to B Class interruptions (affecting only one channel).

Efforts have been initiated to reduce both the duration and frequency of these interruptions. It has been determined that the most effective way to address this issue is by enhancing code quality, security, and performance.

Goals: Our BOA Banking infrastructure framework (Business Oriented Architecture) will be used in the use case. BOA is used as a banking framework in many banks in Turkey and the Middle East. BOA has a Monolithic Architecture, avoiding unnecessary stratification and over architecture as much as possible, while providing visual experiences that make life easier for users with a service-oriented, performance and traceable infrastructure.

In such large multi-channel systems, any unnecessary added layer to the architecture later returns as maintenance cost and flexibility issues. On the other hand, overly primitive designs negatively

affect rapid automation, horizontal scalability, parameterization, and standardization in software development processes.

It is essential that the system supports horizontal scalability with a physical 3-tier architecture. Here, it was possible to design and host core business processes in logical units in the middle layer, and to take the burden of transaction management and orchestration from the data layer. The architecture supports the "Service Bus" topology to allow software development automation. In this way, it is possible to design "strongly typed" transfer objects or contracts for declarative programming of requests flowing on the Service Bus. Business processes can be designed quickly with the "software generating" engines of the system

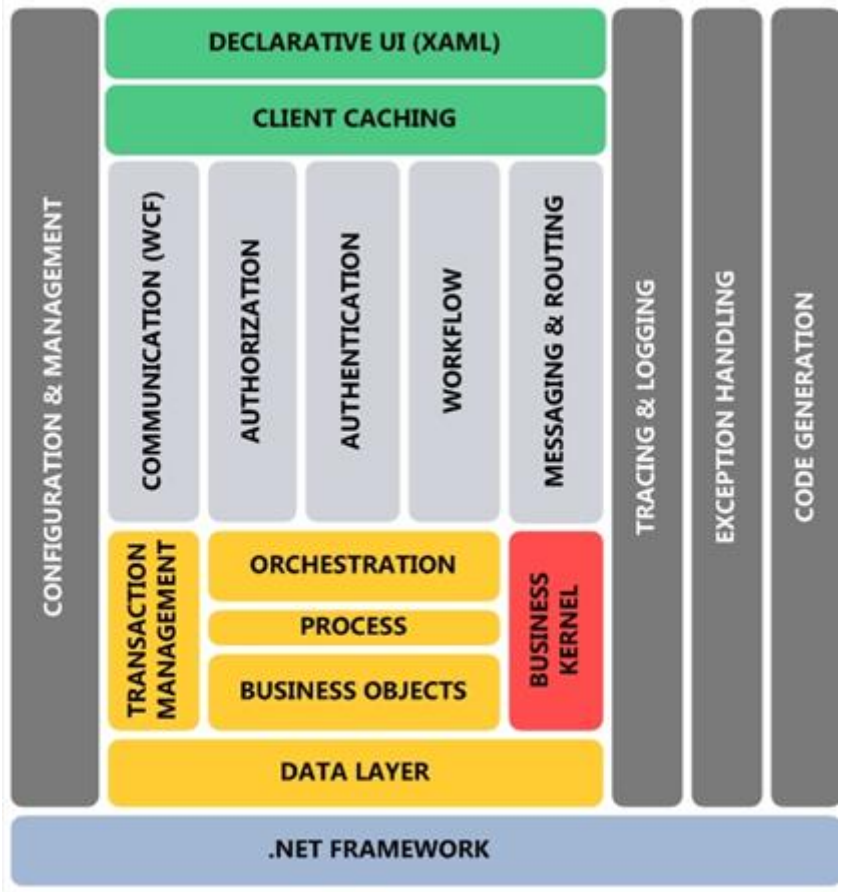


Figure 54: Business Processes

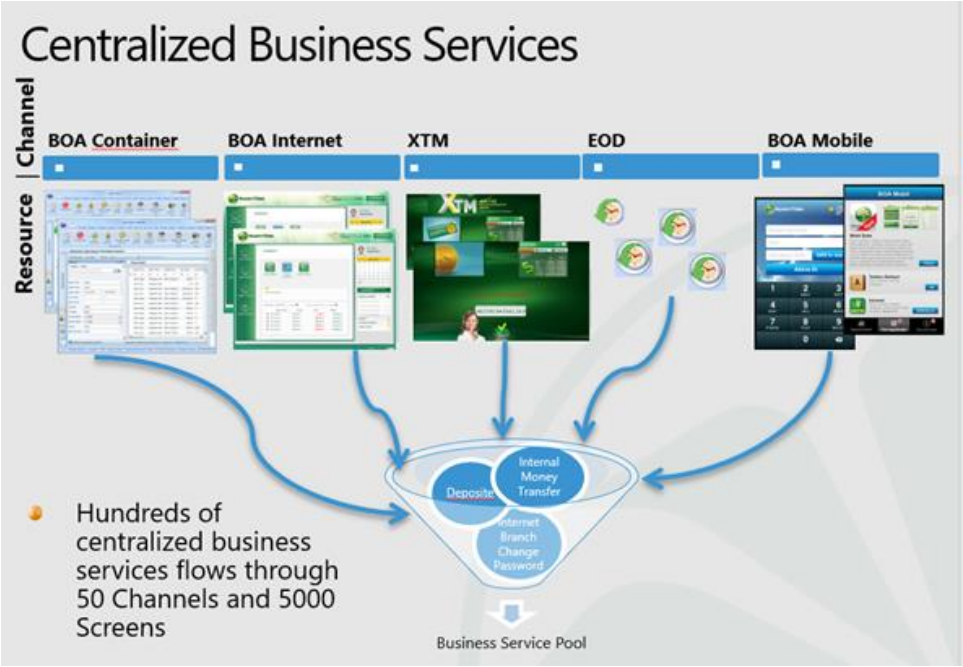


Figure 55: Business Services

Transaction management is automatically managed by the engines of the system, not by the software development teams. Otherwise, nested, and distributed transactions, which increase transaction times, make it impossible to manage the system. In the data layer, there is a minimum business, if possible close to zero, in this layer atomic simple objects are used. The front-end architecture was standardized as much as possible and designed in a declarative model to support switching between different front-end technologies.

Kuveyt Türk Main Banking Program (BOA) consists of approximately 14.8 million lines of code in total, of which 85% is .NET and 10% is SQL based. The BOA Framework receives approximately 100 GB of data and 45 TB of system log data per month. ELK structure is used to collect system log data. In test environments, Jenkins is used for compilation and Azure is used for building the source code, DevOps is used for deployment. In the live environment, Azure DevOps is used for both compilation and deployment. DevOps techniques are used in code deployments and SonarQube application is used to analyze code quality. Also Fortify is used for Code Security. Increasing the quality of code between versions is one of our important business requirements.

Problems and Challenges: Deployments are carried out monthly, incorporating numerous changes each time. The goal is to shorten the intervals between these versions, necessitating continuous integration and continuous improvement.

Each deployment involves approximately 1,000 DLL changes across about 50 channels and 100 different applications. Thousands of files undergo code changes between the monthly deployments, and hundreds of requests and calls are logged in the demand call system. Monitoring code quality, security, and performance changes between these versions poses a challenge. Sometimes, efforts to enhance code quality and security can negatively impact the system's overall performance, while also increasing its size and complexity.

The SonarQube application, used for code quality, provides outputs for code analysis, code review, code quality, and code coverage. However, it does not assist with static code analysis and delta analysis, nor does it offer personnel-based reports by analyzing commit histories in projects.

There is no correlation between code quality and the performance of the work done, and it does not propose design improvements for future structures based on this association.

System performance monitoring and analysis of changes between versions are conducted using traditional methods. Performance metrics for cross-version requests need to be analyzed.

User Stories: The focus is on analyzing code quality and performance within the BOA (Business Analytic Architecture) core banking platform, which involves a substantial amount of code, logging, journal entries, and performance metrics. It is essential to manage the deltas that arise between deployments. There are two stories below.

Story A: Code Quality Measurement Approach:
A Software Development Engineer is involved in various tasks and makes code changes across multiple projects. While they can use code quality tools to monitor the analysis of these projects, they cannot track the differences in code quality metrics between two versions. Similarly, they are unable to follow the results of static code analysis or review design proposals for future projects. Based on personal commit history, it is also not possible to observe code quality metrics on an individual or project basis.

Story B: Cross-version Performance Detection:
When a performance issue or system interruption occurs after a version transition, it is the responsibility of the infrastructure engineer to identify the problematic code blocks. However, tracking all the code changes made after deployment is a challenging task for the engineer.

Firstly, pinpointing the source of the performance issue is essential. This requires generating reports using traditional methods, which involve metrics such as the average request time and the number of errors. This process is time-consuming, and traditional methods do not support delta analysis.

Although pre- and post-version comparisons are conducted monthly during each deployment process, it is not possible to compare the deltas of version changes.

b. Link to SmartDelta Methodology

Extracting, analysing and visualization of the request performance data & code quality metrics of the main banking system is done.

Placement in the SmartDelta-Methodology:

- Condition Assessment
- Prediction
- Visualization

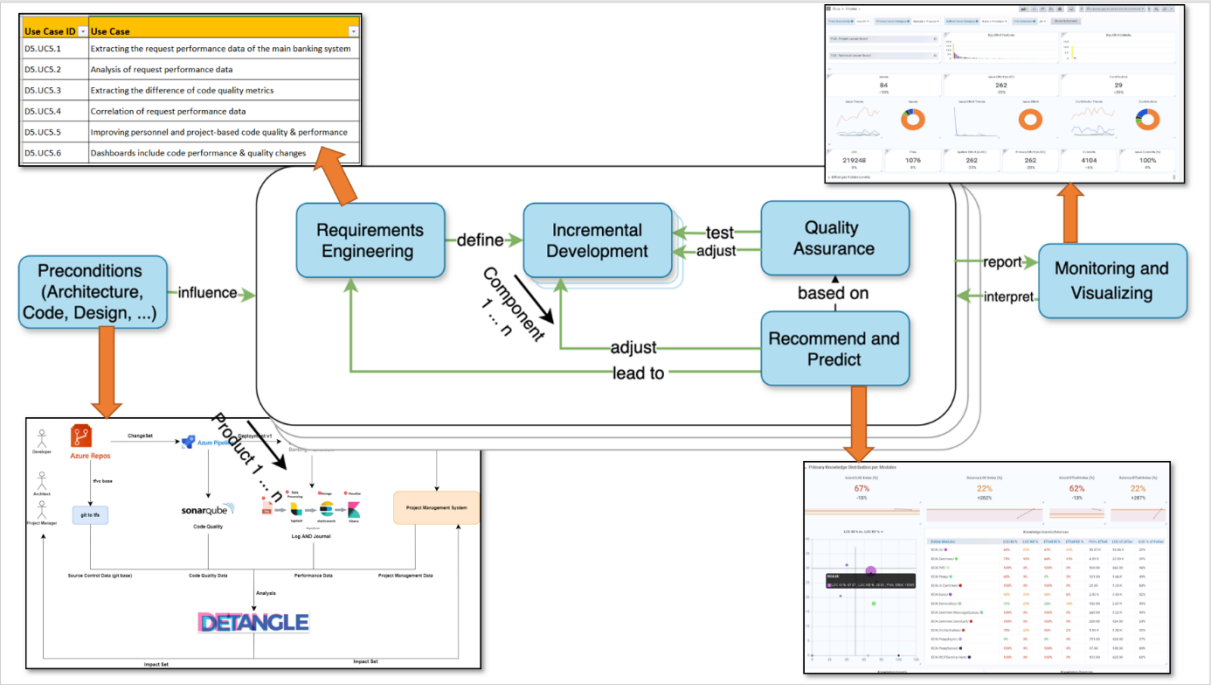


Figure 56: Kuveyt Türk Use Case - Mapping to Methodology

c. Tool Description

DETANGLE®, a software suite to analyze software and locate and measure Technical Debt and knowledge distribution issues in R&D projects, with the following capabilities:

- Identify the Technical Debt that really endangers the future of the software.
- Know the effort required to get the Technical Debt under control
- Locate all code and architecture quality issues to efficiently counteract it
- Examine the organization as a cause for quality problems
- Maintain the ability to innovate by limiting maintenance effort

DETANGLE is an ideal fit for our specific scenario as it seamlessly supports our objective of establishing a correlation between production performance metrics and software quality metrics. Our plan involves utilizing DETANGLE's software quality metrics while contributing our performance metrics for seamless integration within the DETANGLE dashboards.

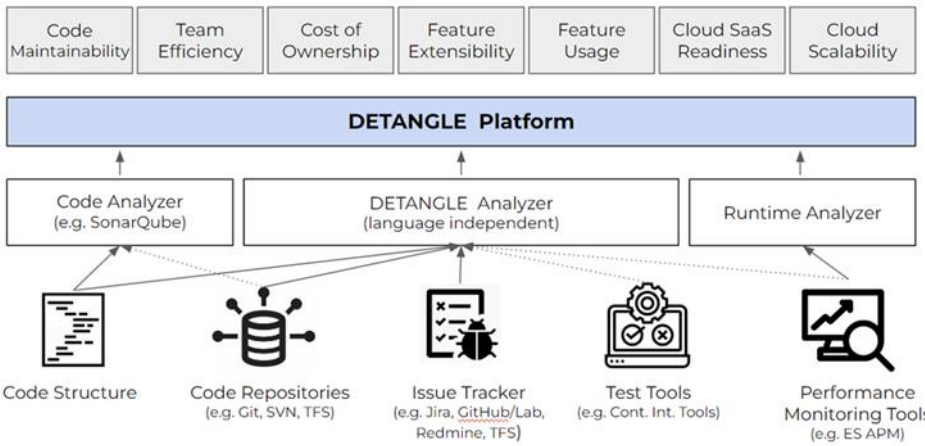


Figure 57: Kuveyt Türk Use Case and DETANGLE

The reference architecture is as below.

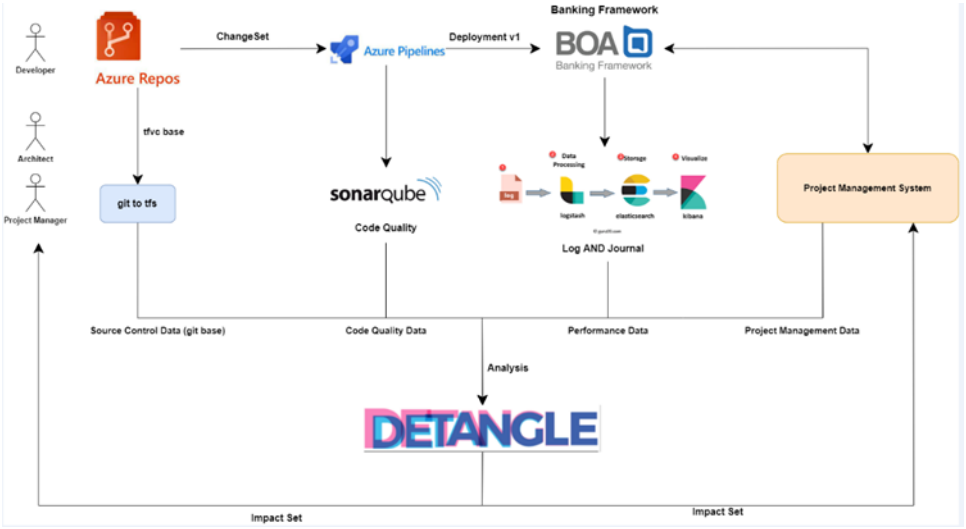


Figure 58: Reference architecture with DETANGLE

d. Visualization

Kuveyt Türk has monthly deployment cycle and software deployments can impact performance metrics. By analyzing these metrics monthly, we can effectively correlate changes over time.

Detailed breakdown of the features and bugs were mentioned as below. We have provided insights into the total effort invested and how that effort has been distributed across different issue types during each monthly period.

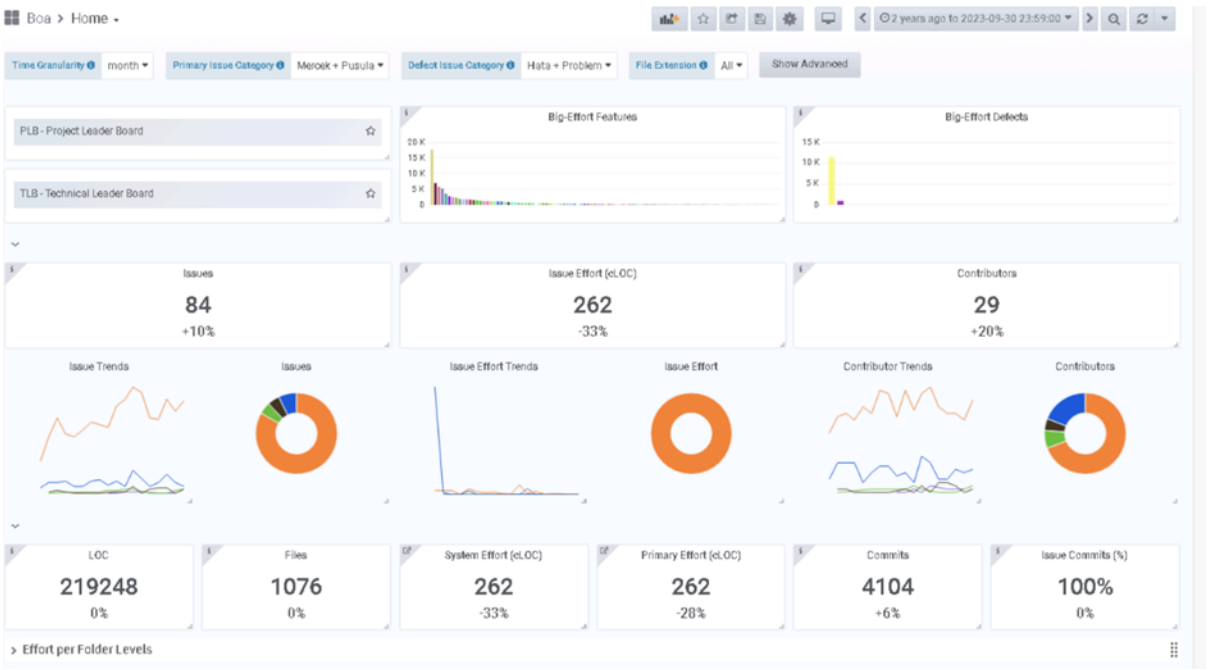


Figure 59: Visualization

Static code quality metrics integrated from Sonarqube. These metrics provide valuable insights into the quality of our codebase

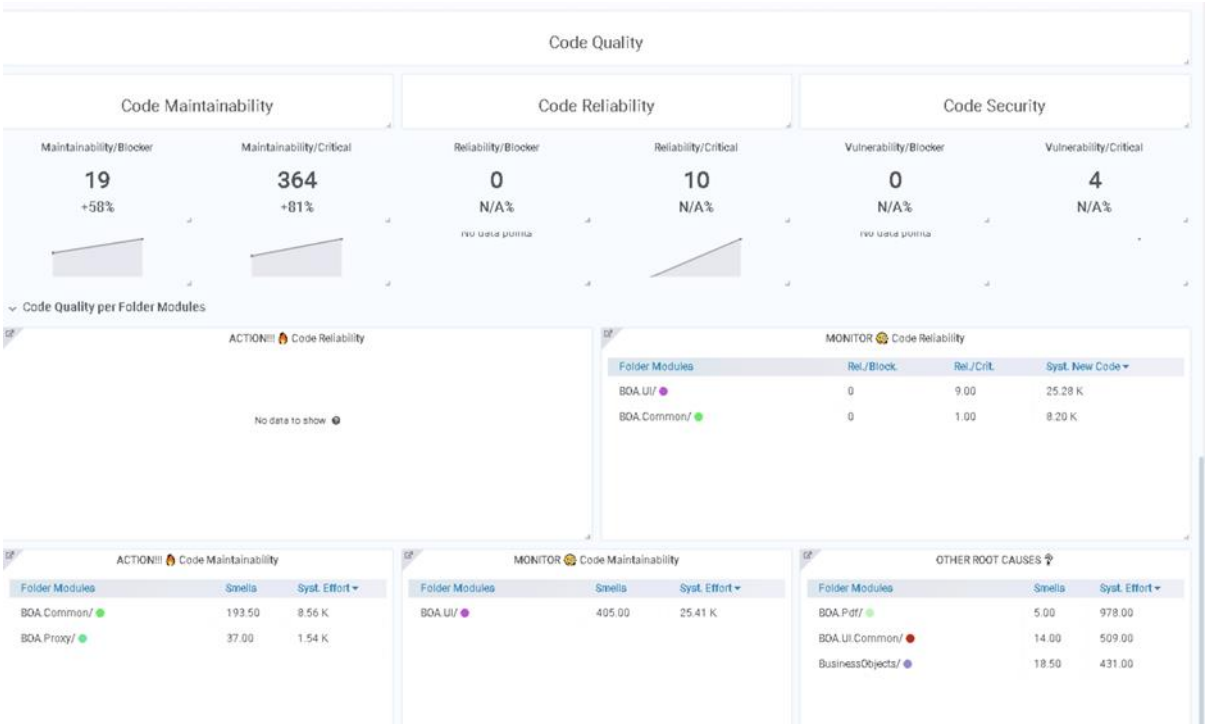


Figure 60: Visualization and metrics from Sonarqube

In the timeseries below, we keep a close eye on specific projects and their PDI (primary debt index) values, allowing us to delve deeper into the details when we seek to enhance our project's extensibility.

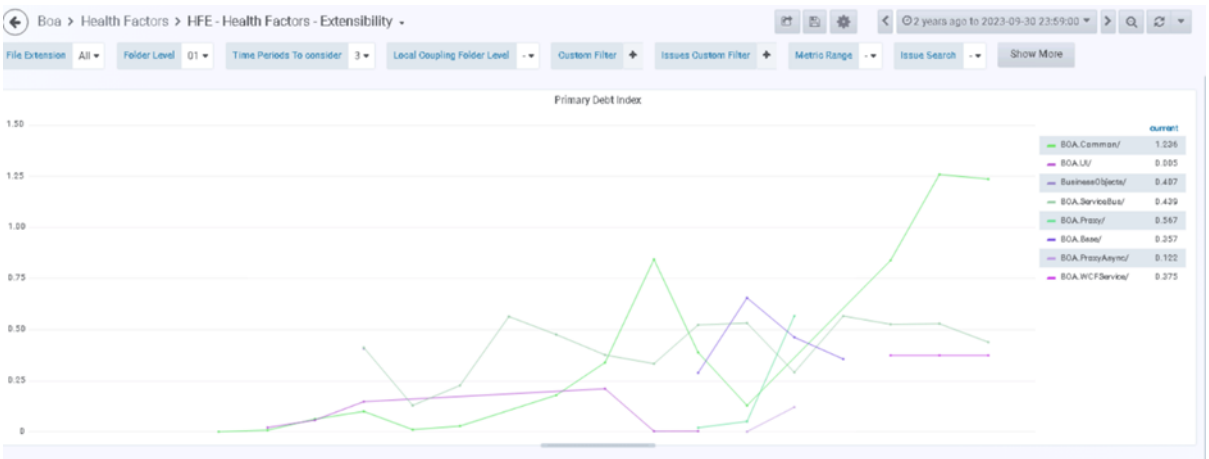


Figure 61: PDI Visualization

Regarding architectural maintainability of our project, these metrics provide insights into how manageable it is to maintain our project. In simpler terms, it tells us how straightforward it is to identify the specific code segment responsible for a bug and how likely we are to fix it without having to make extensive changes to multiple code modules.

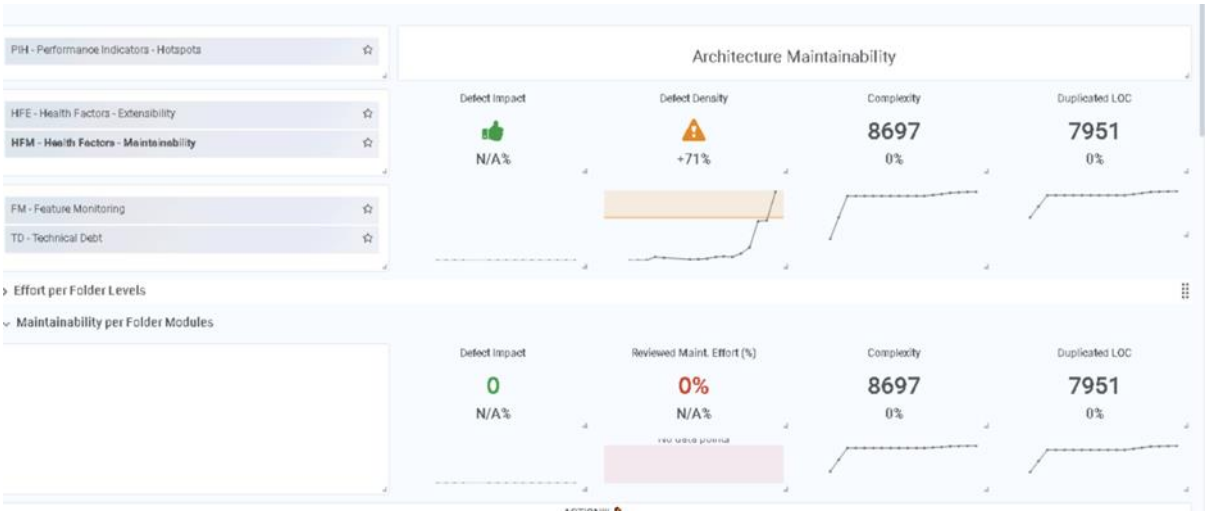


Figure 62: Architectural maintainability visualization

By utilizing various technical debt metrics, we gain the ability to track the amount of effort and time required to address issues within our codebase. This proactive approach allows us to systematically reduce the technical debt that accumulates over time in our project.

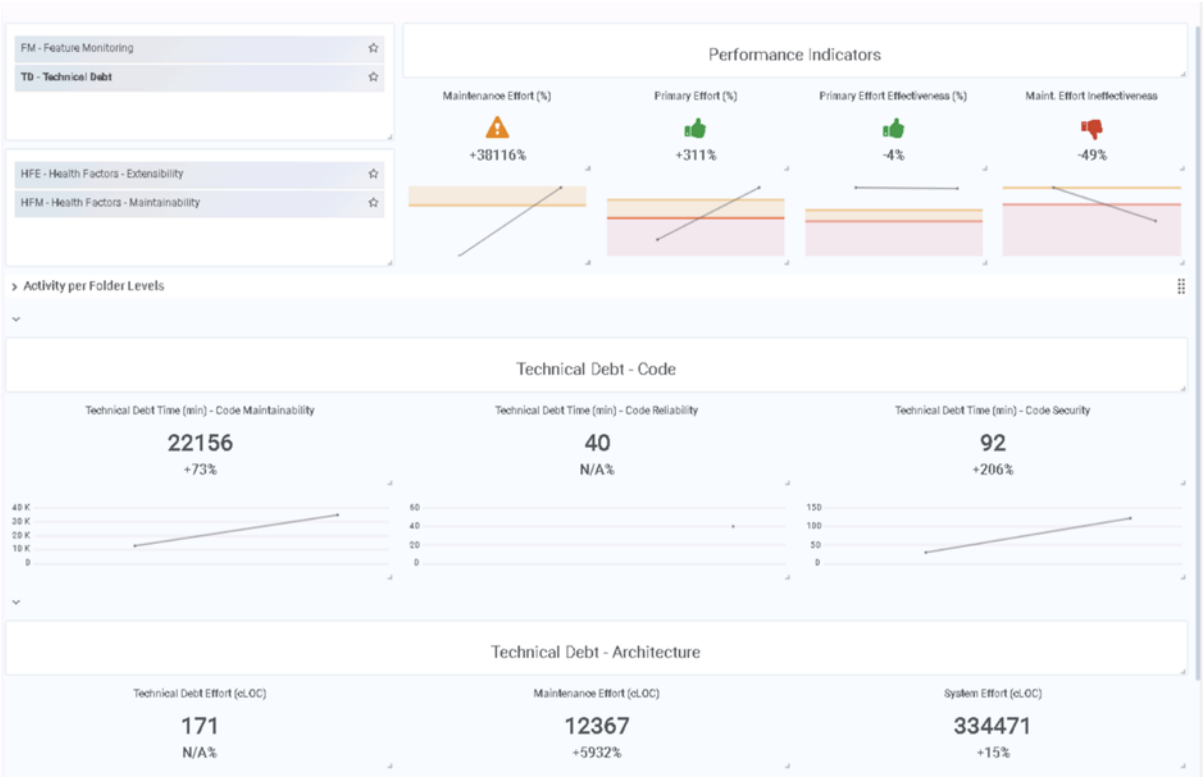


Figure 63: Technical debt metrics visualization

For virtually every metric, we can observe trends in the quality metrics as they evolve over time. This allows us to zoom in on specific time ranges and conduct a more detailed examination of our project. As an illustration, we'll look at some performance indicator metrics and their trends.

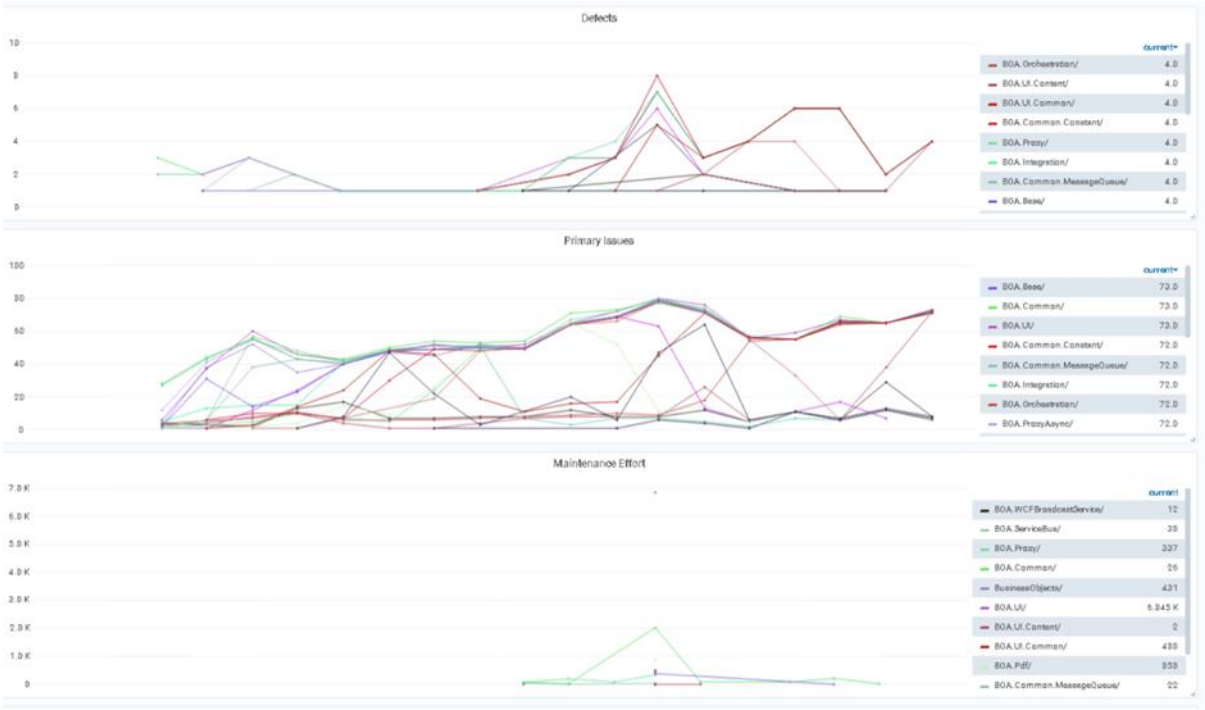


Figure 64: Visualization of quality metrics over time

Moving on to another aspect of our developer analysis, we have a dedicated page featuring a network diagram. This diagram visually illustrates the knowledge-sharing dynamics among our developers with respect to code modules. In this visualization, circles represent contributors, squares represent code modules. You can see “Furkan” as highlighted.

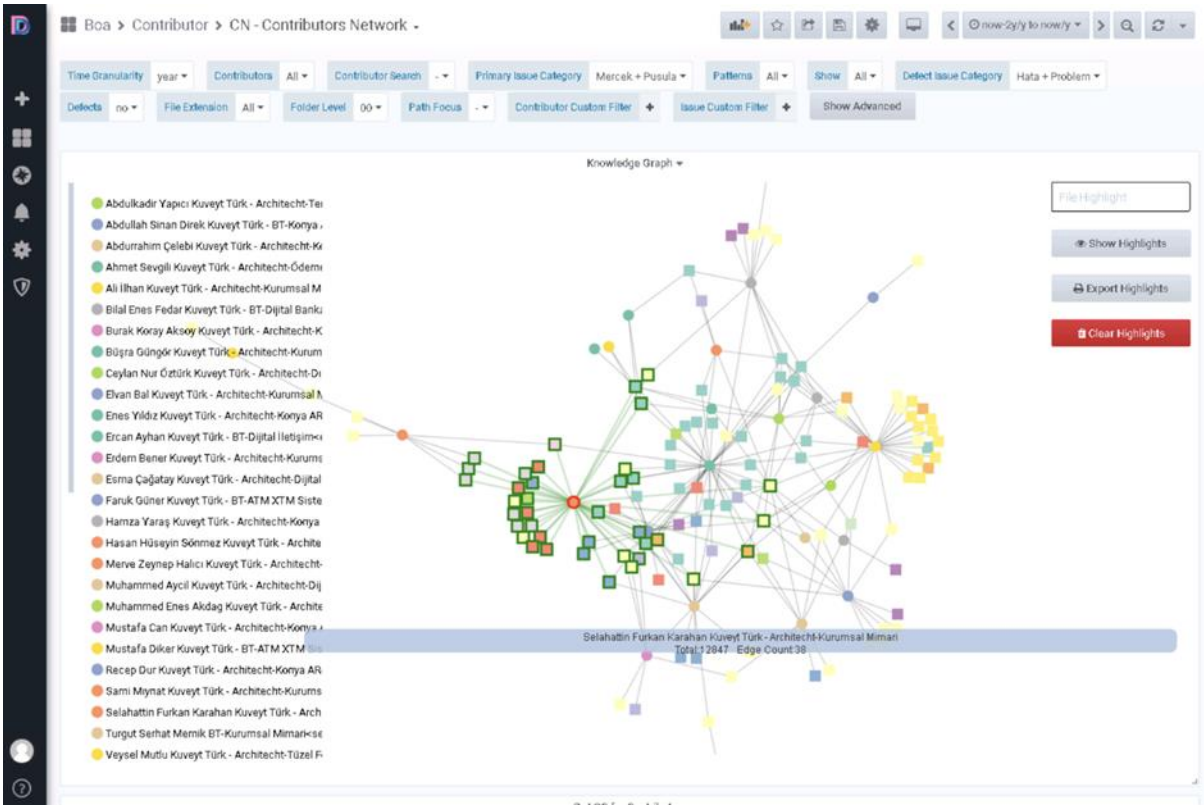


Figure 65: Visualization of developers’ knowledge sharing

According to the knowledge sharing dynamics we identified, we can also show knowledge sharing metrics for folder modules. As you can see in the bubble chart and the table, some projects (folders) have very high percentage of “LOC Knowledge Island”, which means stated percentage of the folder module is developed by only one contributor.

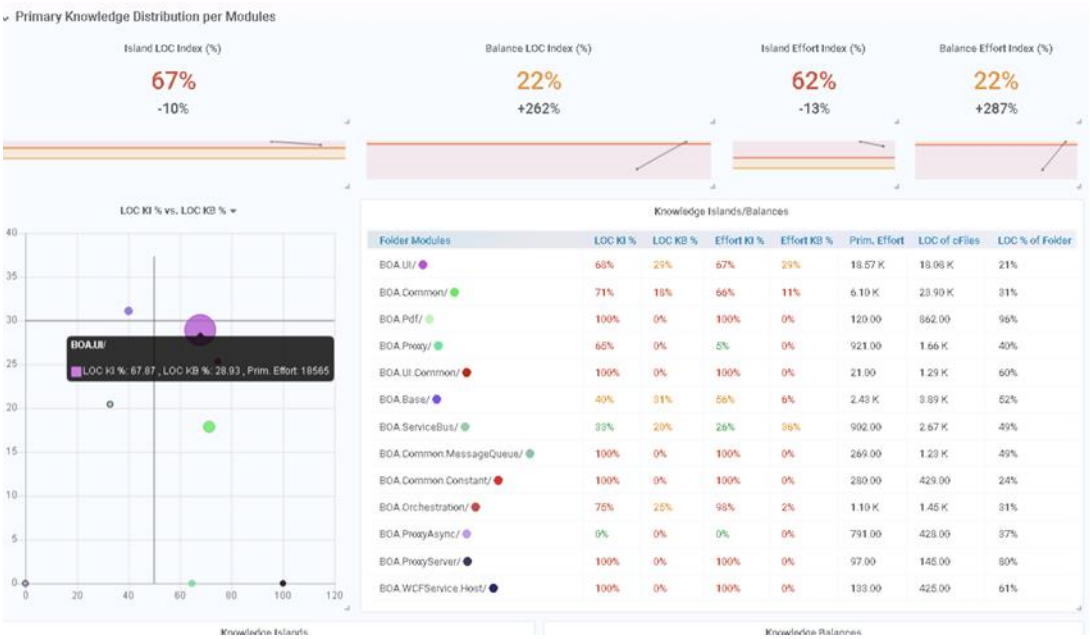


Figure 66: example of a LOC Knowledge Island

We can also visualize the network graph with features and files. In the following network diagram, rectangles represent files and circles represent features. We filter the issues and the files that are not coupled to each other and show only the problematic ones.

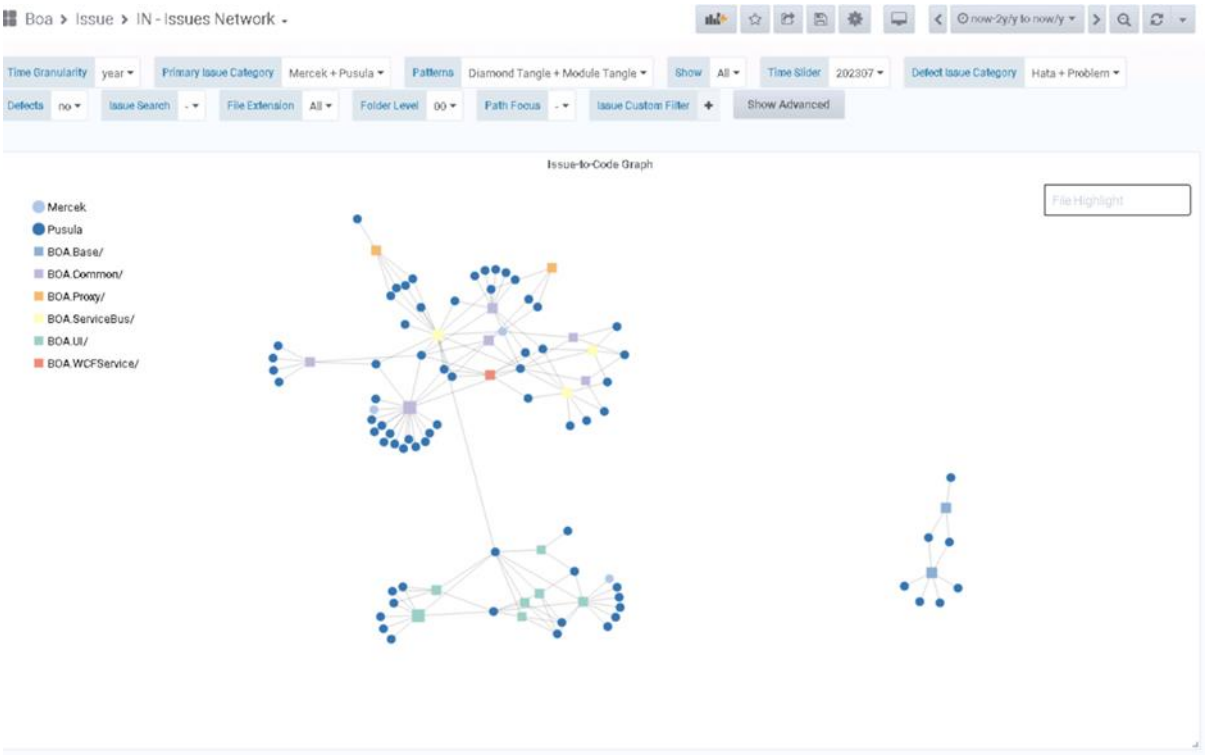


Figure 67: Visualization of features and files network

Visualisation requirements

There are 5 functional requirements for this use case. The 6th & 7th requirements are related to the visualization of the first 5 FRs

Table 5: KPIs overview table

Requirement	KPI Definition	KPI Base Values	KPI Target Values	Description
UC5.FR6	We need to visualize analysed data	0%	100%	Data table, gauge, horizontal & vertical line and bar, pie chart, metrics, time series can be visualized on dashboard
UC5.FR7	We need to visualize analysed data	0%	100%	Dashboards include code performance & quality changes for each deployment, code status metrics, project quality metrics, responsibility metrics, time series for each deployment. Also dashboard can be generated by end-user

e. Evaluation Setup

In Story A, Code Quality Measurement Approach, static code analysis was performed via SonarQube. By using SonarQube APIs, integration between Detangle and SonarQube was done. Within the scope of this story, UC5.FR3 and UC5.FR4 requirements are met.

Within the scope of Story B, Cross-version Performance Detection, Journal and log data were started to be stored using queuing systems on Elastic Search (ELK) to track and analyze the system performance (request average duration) after monthly deployments. Integration is made with Detangle in Csv format. The requirement UC5.FR1 & UC5.FR2 are expected to be met.

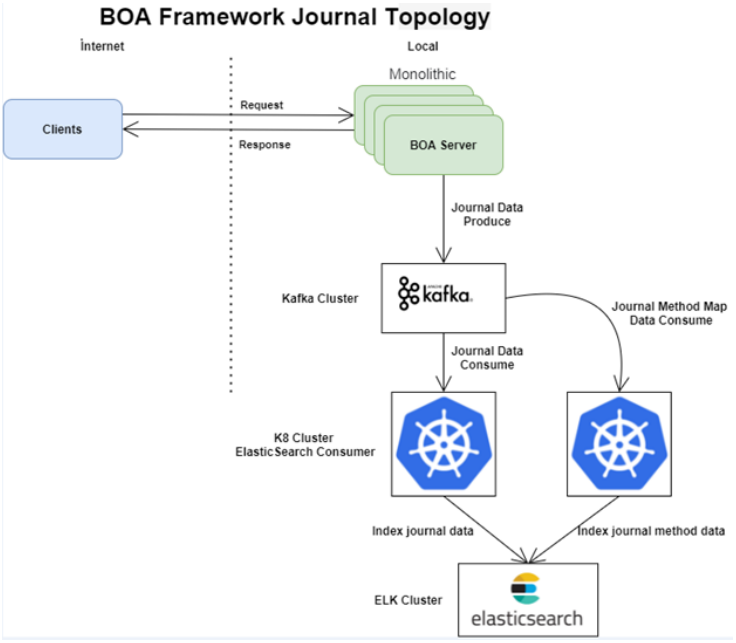


Figure 68: Framework journal topology

f. Evaluation results

Table 6: KPIs overview table

Req.	Tools	Solution partner	KPI Definition	Base Value	Target Value	Achieved Value
UC5.FR1	DETANGLE	ERSTE	We need to extract performance and log data	20%	80%	80%
UC5.FR2	DETANGLE	ERSTE	We aim to analyze the performance and log data between two deployments	0%	100%	100%
UC5.FR3	DETANGLE	ERSTE	We need to correlate the performance and code quality data	0%	100%	100%
UC5.FR4	DETANGLE	ERSTE	We need to extract code quality data	30%	100%	100%
UC5.FR5	DETANGLE	ERSTE	We need to create personnel and project based code quality & performance by interacting check-ins in the code repository with in-house project management tools	0%	100%	100%

g. Recommendation for industry adoption

We assume high potential for improving efficiency, performance and quality of our main banking software:

- Easy metric tracking via Detangle
- Architectural quality, project or person-based outputs, technical debt calculation, code quality calculation, performance tracking
- Correlating and visualizing inputs and metrics will contribute to the development processes.

Challenges:

- Performance data consists of a huge data set and so, the data will be transmitted as a group in order to prevent possible performance problems.
- In every project step, all shared data must be approved by IT Security Department.

8. Use-Case 6 from Software AG

a. Use-Case Description

Motivation

Enterprises are highly dependent on the availability and reliability of their software in today's world. Many functions become impossible or severely slow down when the software cannot be used or trusted. Software AG recognizes its important place in the industry as a supplier of world-class enterprise software and strives to provide its customers with high levels of security, reliability, and quality in the software it provides.

This led to the development of specific policies, controls, and procedures within the company that rely on design, development, and testing artifacts to ensure given characteristics of the products. This decade-long focus of the company naturally resulted in high quality, reliable and secure software for enterprises. However, the artifacts and accompanying controls are often static and may not well support the intent of the company to provide continuous improvement.

Software AG delivers a multitude of products both for on-premises and cloud use. These products are delivered by thousands of people producing millions of artifacts. Every event in the lifecycle of a product is recorded and can be referenced. This daily increasing information could be used to automatically deliver important insights into the current trends in software quality and security supporting Software AG's intention to continuously improve the quality of the software and set the best possible quality and security standards for the industry.

Challenges

Software AG faces several challenges in the production of enterprise software, especially related to the massive amount of code and other artifacts accumulated over time. We expect these challenges are universal for any company that develops a non-trivial amount of software.

1. Repetition of design, code, and tests: There is a large amount of possible reuse of code fragments, components and tests that is not possible to tackle in a manual way due to the sheer size of the source base. For instance, there can be issues similar to something already being developed and developers could save the effort if they would be aware of this.

2. Code quality: Maintaining high code quality is a key aspect for delivering secure, reliable and performant software. A common problem in this context is that the code produced could be sub-optimal and there could be "a better way" but the developers are not aware of this. One potential way to solve this problem is through training but an automatic system pointing out changes leading to sub-optimal performance, e.g., based on "Lessons Learnt", could potentially improve the code even further.

User Stories

Story A: Product development
The developer writes and checks in code on a regular basis. The code adds to the code base of the product and the suite. It is important for the developer and the management to know that the code is suitable and properly written to avoid future re-work.
The developer expects that the various tests will confirm that the code is performing as expected but this confirmation is first available after some time after the developer submits the code. So, the further away from the developer the testing moves, the more likely it is that a required re-work would not be done immediately but will have to be scheduled as a new piece of work, lowering the overall efficiency.
The management is naturally interested in following up on the situation because more re-work and re-work further down the line means more resources tied up and not going towards the development of new functionalities. Currently, the management must rely on KPIs such as the number of tests passed or failed and customer reports to judge the code quality. This feedback is somewhat retarded and poorly reflects the situation.

Story B: Code re-use and re-factoring
Design and code contain numerous blocks that could be viewed semi-independently from the rest of the product. These components are often similar between different products and different parts of one product. The code follows function, so we know that for similar functions we will observe similar code.
The developers often try to re-use the design and code but lack tool support that would allow to map the desired design patterns to existing components and code fragments.
Conversely, when a fragment is found that does not properly contribute to the quality of the software as we expect, there is a pronounced difficulty in finding similar code fragments or components that would likely benefit from refactoring as well.

Functional requirements

- UC6.FR1: The solution must be able to perform the analysis of the development artefacts recorded both in semi-formal and natural language shape in an automated manner to gather the necessary information for further advice on decision taking by management, development, and testing.
- UC6.FR2: The solution must be able to provide a running estimate of the overall product code quality and trend for the management.
- UC6.FR3: The solution must be able to provide an actionable indication of the source code fragment’s quality submitted to it for inspection.
- UC6.FR4: The solution must be able to select existing code fragments that require refactoring to improve the code quality or security based on existing fixes’ analysis.
- UC6.FR5: The solution must be able to provide suggestions for design and code reuse based on the natural language description of feature requirements.

- UC6.FR6: The solution must be able to propose categorizations (labels) for the issues based on the issue title and description, e.g., determine whether an issue is security relevant or not.
- UC6.FR7: The solution must be able to provide a ranking of tests within the whole test base of the product by their relevancy to the feature description and/or code changes.
- UC6.FR8: The solution must be able to suggest tests for reuse based on feature description and/or code changes.
- UC6.FR9: The solution will provide integration capabilities by supporting scripting and API calls while returning outputs in formats that allow for automated parsing.
-

Non-functional requirements

- UC6.NFR1: The solution must be able to operate on reasonably complex code bases and numbers of artefacts (tens of thousands of lines of code).
- UC6.NFR2: The solution must be automatic beyond the installation and configuration stage, i.e., not require a constant human input or supervision.
- UC6.NFR3: The solution must have a low false positive rate even at the expense of a high false negative rate (bias for low noise).
- UC6.NFR4: The solution must be able to operate with reasonable speed with a reasonable amount of resources, like a single contemporary Intel based server computer

b. Link to SmartDelta Methodology

The SmartDelta methodology provides a structured approach for leveraging deltas between artifacts generated in software development processes to address challenges in software development such as software quality management and artifact reuse. The following section describes in more detail how the SmartDelta methodology, its stages and corresponding tools (Figure 69) were applied to our use case to

1. reduce the repetition of design and code,
2. leverage Lessons Learnt to improve quality and
3. automatically classify security-related issues for faster treatment.

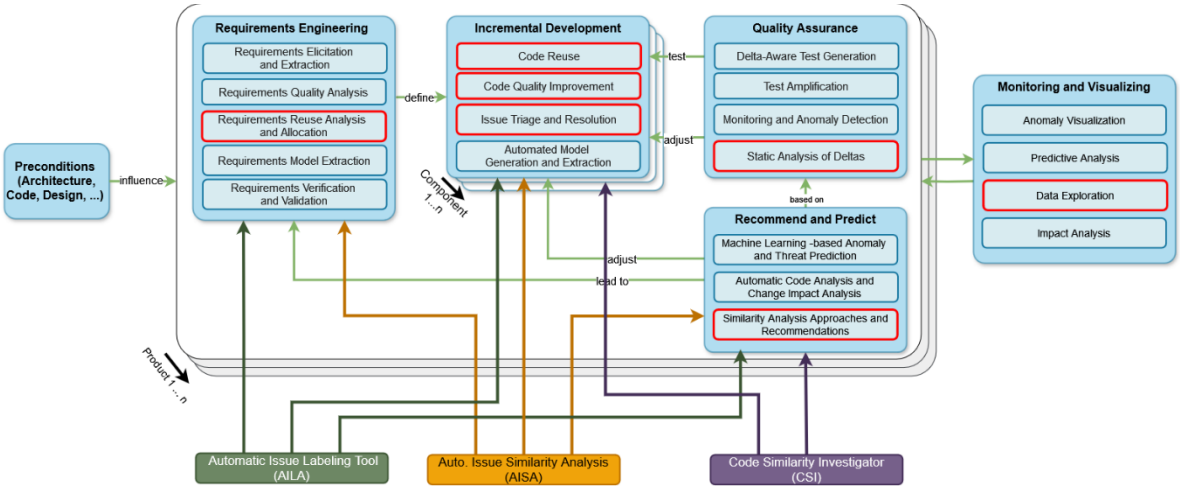


Figure 69: SmartDelta Methodology stages (red boxes) and tools used in the Software AG use case.

Requirements Engineering

In this stage, we process new incoming issues, such as new feature requests and bug reports, to identify security-related issues that need to be addressed as fast as possible, and to detect

similarities to already completed issues which may point us to reusable software artifacts to reduce repetition of design and code.

We address the former task by using the Automatic Issue Labeling Tool (AILA) developed by IFAK. It automatically identifies security-related issues which allows us to prioritize and assign these issues to corresponding experts for faster treatment. This helps us to fix security-related issues in less time and frees time of our security experts, which in turn can be spend on other important tasks.

For the detection of reusable artifacts, we use IFAK's Automatic Issue Similarity Analysis Tool (AISA). It allows us to automatically compare incoming issues with all existing ones stored in our database. This can point us to existing design and code artifacts that could be reused or adapted to solve the incoming issue. It also enables us to detect duplicate issues more effectively. Through AISA, we can effectively search our massive and ever-growing base of software development artifacts, which would not be possible before because of its sheer size. This helps us to effectively reduce the amount of repetition in design and code, which not only can save time but also helps to improve the overall code quality.

Quality Assurance

In our use case, we are interested to make use of "Lessons Learnt" to improve the quality of our code base. A common problem is that the code produced by the developers could be sub-optimal and there could be "a better way" but the developers are not aware of this. We tackle this challenge by using the Code Similarity Investigator (CSI) tool developed by TWT, which computes the similarity between two classes or methods. If we use the tool to compare new code with code that has been fixed in the past and obtain a high similarity score, it might be the case that the new code is affected by the same or similar issue. This helps to avoid similar errors across different products, projects, and development teams. Additionally, the identified fixes can provide hints for the implementation of better code.

Incremental Development

In this stage, we analyze and use the results obtained in the Requirement Engineering and Quality Assurance stages to select reusable artifacts and provide developers with guidelines and hints to improve their code.

For each pair of new and similar existing issues identified, it is now determined, whether the existing code and tests associated with the existing issue can be reused to implement or solve the new issue. This step is ideally carried out by the developers who implemented the code and tests to be reused. They decide whether to use the existing artefacts as-is, modify them, or discard them in favor of a new implementation.

The process for the detected code similarities to fixed code in the past are processed in a similar way. The developer receives the detected similar code snippets along with the corresponding fixes to check if his/her code is affected by the same issue. If so, the developer can improve the code based on the provided fixes.

Monitoring and Visualizing

In the last stage, we use a dashboard along with standard visualizations to provide information and insights about the previous stages and related KPIs to the management. This includes, e.g., a

line chart visualizing the quality trend of selected products over time or a bar chart illustrating the number of already used and potentially reusable software artefacts.

c. Tools descriptions

The following section describes the tools used and evaluated in Software AG's use case. For further details about these tools, please refer to document *D4.5 - SmartDelta Quality Optimization and Recommendation Methodology*.

AILA – Automated Issue Labeling Analysis (IFAK)

IFAK developed a tool for software requirements and issue classification in close collaboration with Software AG. The recommendations on non-functional properties such as security relevance will support development and test teams to reduce time spent on security issues identification, prioritize their requirements and monitor the quality of their software between different versions. Modern issue tracking systems shall support software development teams to keep track about possible bugs, feature requests and critical quality issues such as potential security gaps in their products. Still, a huge effort of advanced security experts is required to review textual issue descriptions manually and judge on potential risks and decide on the security relevance of the issues.

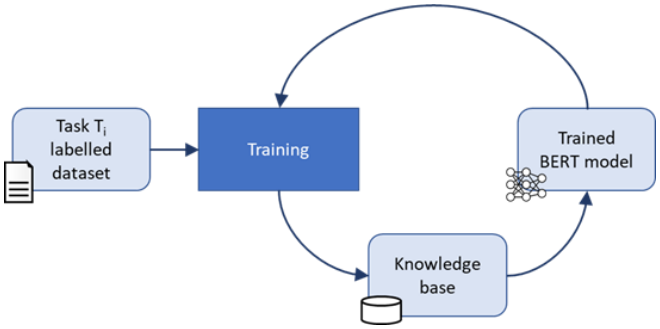


Figure 70: Issue classification flow

For solving these issue classification problems, IFAK has conducted thorough studies and experiments using modern Machine Learning methods and language models such as BERT. One interesting research aspect is the continual learning approach for training a ML model in iterative steps throughout several versions of the code development. Usually, there is the so-called "catastrophic forgetting" which makes it very difficult to train classification tasks incrementally. IFAK analyzed and evaluated several state-of-the-art approaches that aim to improve this behavior. They identified that Continual Lifelong Learning approaches, e.g., with Experience Replay, perform much better than standard Transfer Learning and Multi-Task learning techniques.

Figure 71: Software product quality categories

Requirements can be classified into different types depending on the purpose for which they are needed. For example, requirements can be categorized based on their functional category to facilitate partitioning and reuse of requirements or based on their quality category to identify non-functional requirements. IFAK follows the ISO/IEC 25010 software quality attributes: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, portability. They use a pre-trained BERT base model as the base model. This model will be further trained for the request domain to use the domain-specific information.

AISA - Automated Issue Similarity Analysis (IFAK)

IFAK also developed a tool for issue similarity analysis in close collaboration with Software AG. The recommendations of similar issues and bug duplicates will save time for solving identical problems and is the basis for supporting the development and test team in the reuse of code and tests.
There are many approaches in the literature for bug duplicate detection. Some basic approaches use statistical information based on words such as Bag of Words, BM25 and TF-IDF. More recent approaches use Machine Learning/Deep Learning models based on syntactic information such as CNN, Siamese NN and word embeddings. The state-of-the-art are advanced Deep Learning models based on semantic information, such as Sentence-BERT and other Large Language Models. IFAK has studied many of those approaches and applied them to publicly available issue datasets with labelled bug duplicate information.
Most recently, there have been emerging developments for storing the vector embeddings in common databases, e.g., ChromaDB. This makes it very efficient to obtain different issue descriptions and related embeddings and calculate the similarity scores between them. As the tool is intended to work in an industrial context and should be able to process thousands of issues, IFAK has integrated this solution. As a first step, the tool processes a large corpus of existing issues and creates the related embeddings. Secondly, the new set of issues are used as input and the similarity scores to all other issues will be calculated efficiently. In most papers, the top k, e.g., k = 5, results of the similarity analysis are recommended. However, IFAK uses a different metric in that they recommend only those issue pairs that have a similarity score above some defined threshold, say 0.85. This way, the user will review less false positives (FP) and obtain more true positives (TP).

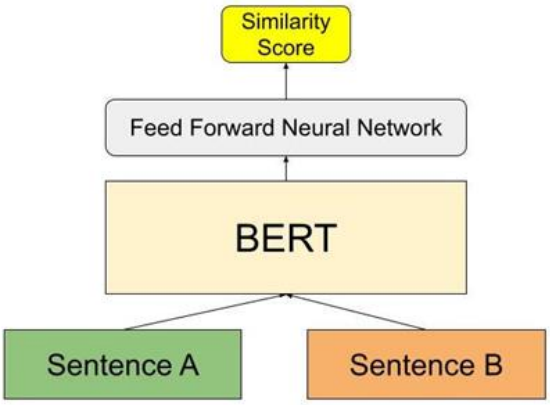


Figure 72: Similarity score calculation

CSI - Code Similarity Investigator (TWT)

Code Similarity Investigator (CSI) is a tool designed by TWT to help software developers navigate the increasing complexity of modern codebases. As software systems grow more intricate, developers face significant challenges in maintaining code, reusing existing functionalities across projects, and

complying with regulations. CSI addresses these difficulties by providing automated support for code reuse suggestions, identifying suitable opportunities for API replacements, guiding refactoring efforts, and helping to prioritize test cases. In doing so, it alleviates the burden of repetitive tasks and streamlines development workflows.

Unlike traditional code similarity tools, which primarily detect syntactic clones, CSI leverages graph-based code representations to capture semantic nuances. Previous methods, such as CCFinder and JPlag, have effectively detected exact or near-exact duplicates but often struggle with language limitations and fail to fully understand deeper code behaviors. Recent advances have introduced semantic analysis through Code Property Graphs (CPGs), which unify structural (AST), control-flow (CFG), and data-flow (PDG) information. These graph representations offer more meaningful code similarity analysis and vulnerability detection, with techniques like GNN-based embeddings further enhancing the capability to identify functional similarities.

At the core of CSI's methodology is the use of CPGs to model code sections. By extracting appropriate subgraphs, the tool concentrates on relevant parts of the code, striking a balance between contextual richness and computational efficiency. CSI then applies Graph Edit Distance (GED) calculations to assess how closely two code graphs resemble each other. Since computing GED is NP-hard, the approach relies on approximate algorithms to maintain practical runtime performance. This allows CSI to handle large, real-world projects without excessive computational resources.

CSI marks a step forward in automating code similarity analysis. By integrating semantic-rich code graphs with efficient similarity calculations, it enables developers to more easily maintain large codebases, accelerate development cycles, and ensure more consistent code quality. Future work on CSI will focus on expanding testing across diverse code repositories, refining approximation algorithms for better scalability, and exploring machine learning enhancements.

d. Visualization

Dashboard solution

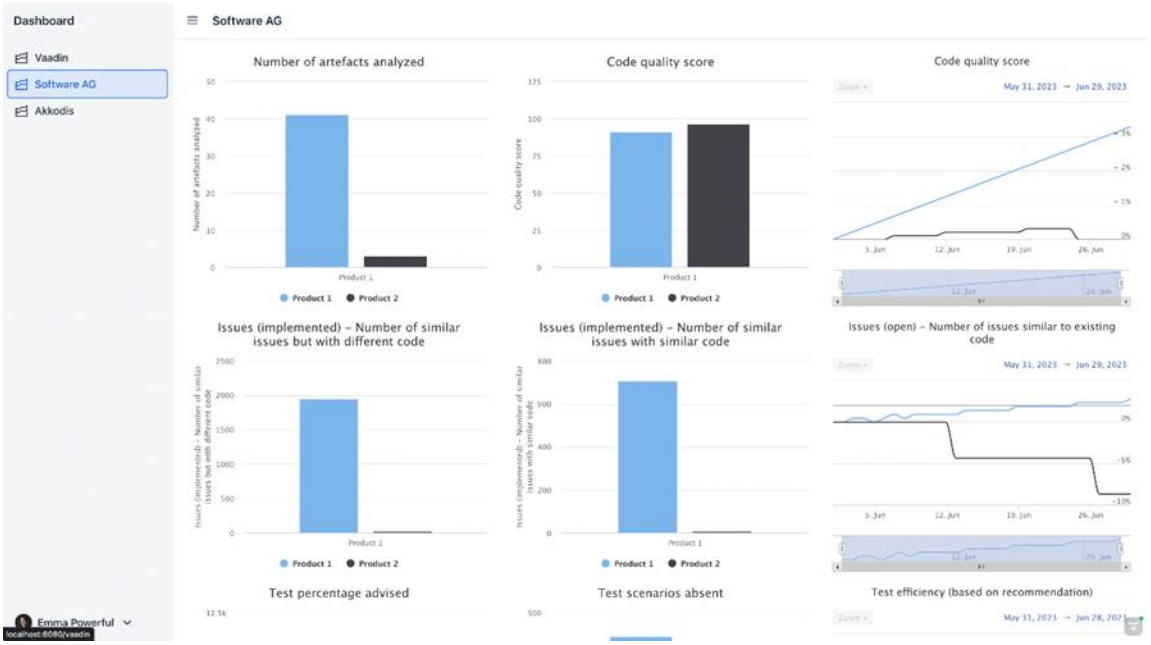


Figure 73: SmartDelta Dashboard for Software AG's use case

Visualization requirements

Apart from the dashboard and corresponding visualization for the management, we did not specify any additional forms of visualization. The reason for this is that we are planning to use and integrate the SmartDelta solutions developed for our use case through APIs only and visualize their outputs using existing tools of our infrastructure such as Jira if necessary. The list of requirements for the management dashboard is shown in table below.

Table 7: KPIs overview table

Req. ID	Title	Description
D5.UC6.1	Management Dashboard - Overview	The management dashboard provides an overview of different KPIs for a selection of products. Detailed KPI views can be opened on demand.
D5.UC6.2	Code Quality Visualizations	This requirement describes visualizations related to the “Code Quality” metric. It comprises the following views: 1. Number of artefacts analysed per product 2. Code quality score 3. Code quality score trend
D5.UC6.3	Issue and Code Similarity Visualizations	This requirement describes visualizations related to issue and code similarity analysis. It comprises the following views: 1. Issues (implemented) 2. Issues (implemented) 3. Issues (open)
D5.UC6.4	Test Recommendations Visualizations	This requirement describes visualizations related to test recommendations. It comprises the following views: 1. Test percentage advised 2. Test scenarios absent 3. Test efficiency
D5.UC6.5	Code Commits Analysis Visualizations	This requirement describes visualizations related to the analysis of code commits. It comprises the following views: 1. Number of degrading code commits This view shows the number of code fragments detected in the recent 2. Frequency of degrading code commit
D5.UC6.6	Bad Code Visualizations	This requirement describes visualizations related to the “bad code” metric. It comprises the following views: 1. Size of bad code 2. Size of bad code (trend)

e. Evaluation Setup

The tools developed in the context of Software AG’s use case have been evaluated by Software AG in separate evaluation setups as described below. Some of these tools have also been used and evaluated by Vaadin in the context of their use case (UC6, Section 12). For the corresponding evaluation setups and results, please refer to Sections 1132.E and **12.F** respectively.

AILA – Automated Issue Labelling Analysis (IFAK)

Background
The classification of issues for further processing is a cumbersome and error-prone manual task that often requires expert knowledge, especially when dealing with issues that are relevant for security. For this reason, Software AG is interested in automating the issue classification task as much as possible (UC6.FR6) and thus set the evaluation focus of IFAK's Issue Classifier on its capabilities to correctly identify security-relevant issues. These include, e.g., bug reports and support requests describing security-relevant issues such as recently discovered vulnerabilities or proposed security enhancements.

Evaluation Methodology
The tool developed by IFAK has been evaluated in two scenarios. First, by IFAK using public data and second, by Software AG using company-internal data. Since Software AG's goal according to requirements is to effectively reduce manual effort by achieving a high degree of automation, the overall objective of model fine-tuning is to reduce the number of false positives, i.e., those issues that are not security-relevant but are recognized by the model as security-relevant since those will be directly manually reviewed.

AISA – Automated Issue Similarity Analysis (IFAK)

Background
Identifying similar or related issues, e.g., to newly created issues is an important prerequisite for making recommendations for re-using software artefacts such as code and tests. The tool developed by IFAK, in combination with other solutions, will thus form the basis for fulfilling Software AG's requirements UC6.FR5, UC6.FR7 and UC7.FR8.

Evaluation Methodology
This tool has been evaluated by Software AG using company-internal data. To be sure that the results the tool produces meet the requirements and expectations, the issues marked as similar have been reviewed manually. This is also necessary to estimate the accuracy of the tool, as there is no duplication marker or connection between similar issues available in the dataset.

CSI – Code Similarity Investigator (TWT)

Background
When developing new software with similar features to existing products such as product variants, it is likely that this new software requires some components and functionality, that have already been developed before. If these parts are constructed in a similar way, it is also likely that if one is found to expose a security vulnerability, the other one will be affected the same vulnerability as well. The tool developed by TWT, in combination with other solutions, will thus form the basis for fulfilling Software AG's requirement UC6.FR4 by providing suggestions of similar code fragments to sections, that have already been identified to include vulnerabilities or other issues.

Evaluation Methodology
To evaluate the CSI tool, it has been fed with company-internal data by Software AG. Since the tools goal is to find similar code fragments or components, the evaluation purpose lies within reducing the number of false positives and in promoting those fragments to the user that show the highest similarity. As there are no similarity indicators available in the dataset, the evaluation took an explorative approach, trying to identify patterns in the tool's results as well as manual classification of subsets.

f. Evaluation results

AILA – Automated Issue Labeling Analysis (IFAK)

Experimental evaluation by IFAK

IFAK trained and evaluated the Issue Classifier on 10k publicly available issues published on the GitHub repository of Microsoft's Visual Studio Code IDE. The purpose of this initial evaluation was to get some initial results regarding the accuracy of the classifier tool before proceeding with the data of Software AG. It is known from the literature that fine-tuning is helpful due to the ability of the BERT model to transfer learning, i.e., to transfer the knowledge on which it was trained to a similar task or similar data. This initial evaluation was conducted on a qualitative basis and showed promising results, as several security related issues were correctly classified. A more detailed quantitative evaluation was skipped to focus on testing the model in the context of Software AG's use case and its data.

Use case-oriented evaluation by Software AG

We evaluated IFAK's Issue Classifier and its capabilities to generate ML models for predicting whether an issue is security-relevant or not in multiple steps. Thereby we trained and fine-tuned different models based on IFAK's model pretrained on 10k GitHub issues and on the original BERT model and using different SAG-internal ground truth datasets.
The issues of these datasets have been annotated with a reliable "security flag" indicating that the issues are either security-relevant or not. The SAG-internal ground truth data comprises the following datasets:

1. *SAG_small* – This dataset comprises 936 issues manually labelled as security-relevant or not by a security expert
2. *SAG_small_v2* – This dataset comprises all issues of the *SAG_small* dataset but contains 12 additional issues that have been added later during the evaluation phase
3. *SAG_large* – This dataset contains 9508 issues in total. 3986 (~42%) of them are reliably labelled with CWE labels representing the security-relevant part of the dataset, while 5522 (~58%) issues are related to localization or performance problems that can reliably considered to be not security-relevant
4. *SAG_full* – The full dataset represents the union of the *SAG_small* and *SAG_large* datasets containing 10439 issues with confirmed security labels in total
5. *SAG_full_v2* – This dataset is equivalent to *SAG_full* but uses the extended SAG_small_v2 dataset. It contains 10451 issues with confirmed security labels in total
6. *SAG_full_v2_noCSO* – This dataset contains all issues of the *SAG_full_v2* dataset but those with a CSO label. These issues have been omitted since they are security relevant but are not related to RnD security issues we are interested in. They often contain keywords that could be interpreted as an indicator for security-relevant issues such as the terms "security" or "vulnerability"

The following sections describe the evaluation of the generated and fine-tuned models using these datasets in detail. Note that our focus according to our requirements is to achieve the highest level of automation that is possible. Therefore, we are interested in minimizing the number of false positives, i.e., non-security issues that are reported as being security relevant.

Evaluation and finetuning of IFAK's 10k GitHub model

To get an initial impression of the accuracy of the base model trained by IFAK on 10k publicly available GitHub issues, we used it to predict the "security flag" of ~1500 issues with known Common Weakness Enumeration (CWE) labels. On this dataset, the model achieved a promising accuracy of ~80%. Therefore, we conducted additional experiments to improve the accuracy further.

1. In the first experiment, we fine-tuned the model using the *SAG_small* dataset using five epochs. This new model achieved a slightly increased accuracy of ~83% but still erroneously reported 79 issues as being security relevant (8.4% of the issues). A potential explanation was that the GitHub issues used for training the initial model are too different from the SAG issues, e.g., regarding their style of writing, keywords, etc.

2. In a second experiment, we thus first finetuned IFAK's 10k GitHub model with our entire issue dataset containing ~1 million issues but with uncertain security labels to indirectly tweak the model towards the issue style used at Software AG, followed by a second fine-tuning using the *SAG_small* dataset. While the results of the first finetuning step looked quite promising with an accuracy of 98.4% (the real accuracy was probably much lower because of the uncertain labels in the dataset), the results for the finetuning step on the ground truth data showed very bad results in comparison with an overall accuracy of only ~52.5%. Moreover, the resulting model was not able to correctly classify any of the real security-related issues in the dataset.

Table 8: Results for experiments 1 and 2

Input Model	Input Data	Epochs	TN	FP	FN	TP	Accuracy
IFAK BERT GitHub 10k	SAG_small	5	412	79	76	369	83.4%
IFAK BERT GitHub 10k	995615 issues with uncertain labels	5	N/A*	N/A*	N/A*	N/A*	98.4%**
IFAK BERT GitHub 10k + 995615 issues	SAG_small	5	491	0	445	0	52.4%

* The confusion matrix was not created because of a software bug after completing the training
** The real accuracy is very likely much lower since the input data is not reliably labelled

Evaluation and finetuning of the BERT model

Since the highest accuracy achieved in the experiments with the 10k issues GitHub model was only ~83% on the *SAG_small* dataset, we decided to perform additional experiments using the BERT model as basis in combination with extended ground truth data (*SAG_large* and *SAG_full* datasets) to assess if this could deliver better results.

As a first step, we conducted the following experiments:

3. We finetuned the BERT model with the *SAG_large* dataset (9508 issues) using 5 epochs. This resulted in a very high accuracy of ~98.7%, with only 28 false positives (FP) and 94 false negatives (FN), respectively. The resulting model was then further tuned in a second step on the *SAG_small* dataset with five and 15 epochs. Interestingly, this additional tuning reduced the accuracy of the model to ~89,2% and ~90,7%, respectively. This indicates that the classification of the issues in the *SAG_small* dataset (i.e., issues manually labelled by a security expert) is much more difficult for the model, especially if it has been previously trained on a dataset only containing CWE-labelled issues and issues related to localization and performance problems.

4. In the second experiment, we finetuned the BERT model using 15 epochs on the *SAG_full* dataset. This dataset contains the same issues as in the first experiment but this time, they have been processed in a single step. This model achieved an accuracy of ~97.3% with 158 FP and 122 FN.

Table 9: Results for experiments 3 and 4

Input Model	Input Data	Epochs	TN	FP	FN	TP	Accuracy
BERT	SAG_large	5	5494	28	94	3892	98.7%
BERT + SAG_large	SAG_small	5	N/A*	N/A*	N/A*	N/A*	89.2%
BERT + SAG_large	SAG_small	15	461	30	57	388	90.7%
BERT	SAG_full	15	5855	158	122	4304	97.3%

* The confusion matrix was not created because of a software bug after completing the training

In the second step, we continued the finetuning experiments using the apparently hard to classify *SAG_small* dataset and the *SAG_full* dataset, but this time including 12 additional manually labelled issues that were not classified correctly in previous experiments. Moreover, we used a different number of epochs to assess how this will impact the overall accuracy of the models.

5. This time, we tuned the BERT model on the *SAG_small_v2* using 5, 10 and 15 epochs. The highest accuracy of ~94.5% was obtained during the training with 15 epochs with just 9 FP and 43 FN.
6. For the finetuning experiments on the *SAG_full_v2* dataset, we selected 20, 25 and 30 epochs. The highest accuracy on this dataset of ~98,1% was achieved with 30 epochs resulting in 75 FP and 121 FN.

Table 10: Results for experiments 5 and 6

Input Model	Input Data	Epochs	TN	FP	FN	TP	Accuracy
BERT	SAG_small_v2	5	485	16	94	353	88.4%
BERT	SAG_small_v2	10	481	20	44	403	93.2%
BERT	SAG_small_v2	15	492	9	43	404	94.5%
BERT	SAG_full_v2	20	5783	240	61	4367	97.1%
BERT	SAG_full_v2	25	5826	197	76	4352	97.4%
BERT	SAG_full_v2	30	5948	75	121	4307	98.1%

The model generated in experiment 6 was the best performing model so far. But even though the number of “only” 75 FP appears to be overall low, it is still too high to use the model in a fully automated manner, as per our requirements.

To further improve the results, IFAK thus suggested to pretrain the BERT model on the entire SAG issue dataset (1 million issues) to tweak the model towards the “style” of our issues before finetuning it for a specific purpose such as classification of security relevance. This approach is similar to experiment 2 with the 10k GitHub issues model but this time, the training does take the potentially incorrect security flags into account and only uses the title and description of the issues.

Evaluation and finetuning of the SAG Base Model

After receiving the updated tool with pretraining support from IFAK, we created a new base model called SAG Base Model by pretraining the BERT model on our entire issue dataset of ~1 million issues in total.

7. For this last experiment, we finetuned the SAG Base Model using the *SAG_full_v2_noCSO* dataset. This dataset is a subset of the *SAG_full_v2* dataset, which does not include issues that are security-relevant but are not related to RnD, the area we are interested in, and thus could easily be misinterpreted because of keywords such as “security”. First, we tuned the model using 30 epochs and observed the accuracy trend. The final accuracy of this model was ~95% with only 34 FP. Since the highest accuracy of ~98% was reported for epoch 27, we

initiated the tuning step again using 27 epochs. This time, the model only reported 21 FPs and the overall accuracy increased slightly to ~95.2%.

Table 11: Results for experiment 7

Input Model	Input Data	Epochs	TN	FP	FN	TP	Accuracy
SAG Base Model	SAG_full_v2_noCSO	30	5910	34	479	3933	95%
SAG Base Model	SAG_full_v2_noCSO	27	5923	21	479	3933	95.2%

AISA – Automated Issue Similarity Analysis (IFAK)

Experimental evaluation by IFAK

For a first experimental evaluation, IFAK identified publicly available issue datasets with duplicate information from the paper "Duplicate Bug Report Detection: How Far Are We?" (Zhang et al., 2023). They provided six high-quality datasets from 3 different issue-tracking platforms (GitHub, Jira, BugZilla). IFAK has chosen the Eclipse and VSCode datasets with 27.583 issues (1.447 duplicates) and 62.092 issues (4.386 duplicates), respectively.

They evaluated the statistical methods TF-IDF and BM25, the word embedding technique Word2Vec and several Sentence-BERT large language models (paraphrase-distilroberta-base-v1, sentence-t5-base, all-mpnet-base-v2). As a first measure, IFAK calculated the top k most similar issues according to the cosine similarity scores. As can be seen in the table below, about 31% of all duplicates can be recommended as the first best result (using TF-IDF on Eclipse data). This means, if IFAK provides the first recommendation for all 27k issues, they identify about 500 duplicates. However, it would be an enormous effort for a software engineer to check the large number of recommendations in view of the relatively low success rate. Furthermore, providing the top 5 recommendations would mean reviewing 5x 27k issues with only a slightly increasing success rate of about 700 identified duplicates.

Table 12: Identified issue duplicates in the top k recommendations

Top k	# Duplicates	% Duplicates
1	**511**	**31.3%**
2	**594**	**36.4%**
3	**645**	**39.6%**
5	**706**	**43.3%**
10	**774**	**47.5%**
50	**958**	**58.8%**

IFAK therefore decided not to use the standard measure used in the literature, but to take the similarity scores into account. As can be seen in the table below (using S-BERT on Eclipse data), if IFAK defines a certain similarity threshold and count the duplicate issues among all predicted issues, the success rate (precision) is much higher. For example, for a score threshold of 0.85 the number of issues to be reviewed is about 1.000 with more than 400 duplicates identified. In this case, the number of identified duplicates compared to the total number of all labelled duplicates (recall) is about 17%. Since the recall does not increase significantly at lower thresholds, they considered the value 0.85 to be a good balance between human effort and identified duplicates for a further round of evaluation at Software AG. It turned out that S-BERT (all-mpnet-base-v2) achieved the best results compared to all other methods and models. Careful preprocessing of the issues has also considerably increased the accuracy.

Table 13: Identified issue duplicates using a similarity threshold

Score	TP	FP	Precision	Recall
1.00	237	21	91.9%	9.9%
0.95	296	113	72.4%	12.3%
0.90	352	276	56.1%	14.6%
0.85	424	568	42.7%	17.6%
0.80	586	1248	32.0%	24.4%
0.75	784	2885	21.4%	32.6%
0.70	992	6197	13.8%	41.2%

Use case-oriented evaluation by Software AG

In a first step, pre-processing and filtering rules were determined in a small subset, which then allowed to exclude obviously irrelevant issues (e.g. automatically generated reports, empty issues, irrelevant contents) from the results to reduce the number of issues that must be manually classified as well as increase the share of relevant issues.

In the next step, a time-limited subset was extracted from the dataset for evaluation. This subset contained about 4,236 issues before applying the filtering rules from the first step. For the manual classification, an effort of over 600 minutes was put in to classify ~2,334 issues, although the time spent is likely to be higher, as the tracking of time and issues was added later during the evaluation process.

During the evaluation, a ratio of about 1:3.5 issues to suggestions was observed. This means that for every issue, that has been part of the evaluated subset, there were 3.5 suggested similar issues that have been inspected and classified.
The evaluation was conducted by two persons, who tracked time and issues/suggestions evaluated. Person A classified 107 issues and suggestions in 14.5 minutes, resulting in a speed of ~7.38 issues per minute. Person B classified 1,400 issues in 559 minutes, resulting in a speed of ~2.5 issues per minute. While person A was slightly more familiar with the dataset than person B, it is likely that the average person's speed falls somewhere around the average of the two observed speeds. This implicates an average speed of ~5 processed issues per minute, when the classification is done for short intervals. The dataset used consists of an average of ~280 issues per day, which would result in ~56 minutes of work, a person would spend to classify these using the tool as a daily task. The suggested similar issues from the overall dataset for issues from the subset were classified as ~21% similar and relevant, ~71% similar but irrelevant and up to 7% as not similar.

The numbers suggest that the tools objective comparison and similar suggestions capabilities have a low error rate of ~7%, but for our use case only 21% of the suggestions were relevant.

CSI - Code Similarity Investigator (TWT)

Experimental evaluation by TWT

TWT conducted a comparative evaluation of the CSI algorithm using 38 total comparisons: 23 file comparisons and 15 method comparisons. All method comparisons came from Vaadin, while Vaadin also provided 9 out of the 23 file comparisons, with the remaining 14 file comparisons contributed by Software AG. The CSI algorithm assigns each pair of items (files or methods) one of four labels—*no, low, medium, high*—indicating perceived similarity.

TWT converted these labels to numerical rankings from 0 to 3 (no=0, low=1, medium=2, high=3). Each pair's accuracy is determined by taking the absolute difference between the programmer's label (ranking) and the CSI label. A difference of 0 means complete agreement, 1 indicates mild disagreement, 2 represents disagreement, and 3 implies strong disagreement.

This process was repeated before and after performing hyperparameter tuning. By comparing how many comparisons fall under each different category (0, 1, 2, or 3), TWT assesses whether the tuning helps align the CSI scores more closely with the programmers' judgment.

For *file comparisons*, there is a notable improvement after tuning: perfect matches (difference=0) rose from 5 to 10 out of 23, while no comparisons had a difference of 2 or 3 post-tuning (compared to 3 cases of difference=2 before). This clearly shows that tuning made the CSI algorithm much more consistent with the programmers' labels at the file level.

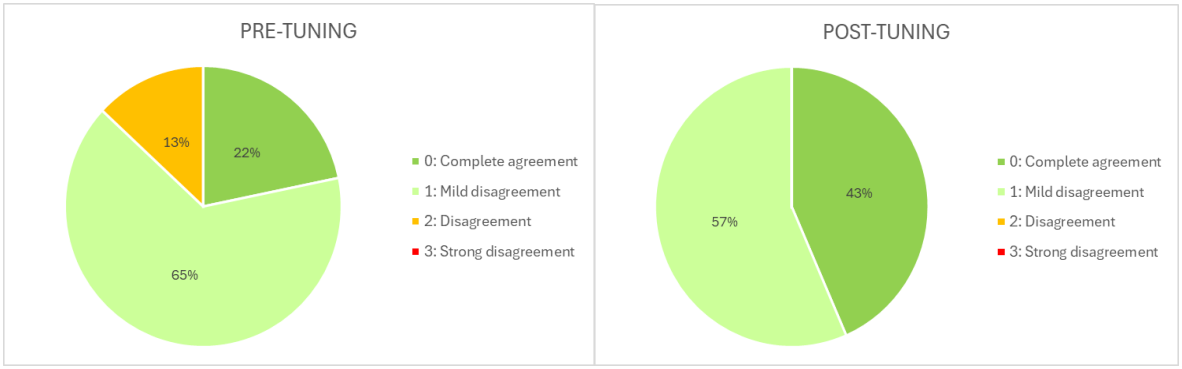


Figure 74: Distribution for the file comparison

For *method comparisons*, the algorithm's performance slightly dipped: perfect matches decreased from 10 to 9, and mild disagreements increased from 5 to 6. Although the decrease is minor, it suggests that while tuning significantly helped file-level similarity detection, it might require further adjustments to maintain the same level of performance for method-level comparisons.

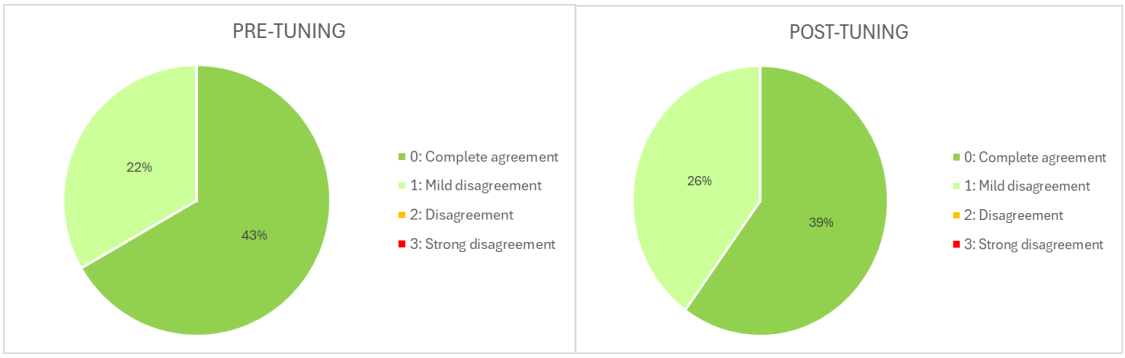


Figure 75: Distribution for the method comparison

Use case-oriented evaluation by Software AG

As outlined in the previous section, TWT performed some first experiments using input provided by Vaadin and Software AG. For these experiments, we provided two different datasets. The first dataset comprised code submissions written by university students. These were made of small programs implemented according to formal exercises and allowed the comparison of different solutions from up to three students per exercise. The advantage of this dataset was that the included programs had the same functionality according to the exercise specifications but represent different variations as they were developed by different individuals independently from each other. In a second step, TWT was provided with another small dataset containing samples

for different similarity levels (none, low, medium, high). For this, an excerpt from the first dataset was combined with some new code samples from company-internal projects. The degree of similarity of the code samples included in this dataset have been classified manually by a developer.

After these initial experiments performed on individual, curated files by TWT, we evaluated the CSI tool on a large dataset containing code from multiple company-internal projects. Here, we used full software development projects as input, including source files but also supplemental files that may not be relevant to the tool. The tool's task, besides comparing the files and calculating the similarities, was to recognize project structures and ignore irrelevant files. The included software development projects are implemented in different programming languages like Java and JavaScript/TypeScript. While the size of the analyzed code base dramatically increased in comparison to the first experiments by TWT, the projects were still rather small compared to enterprise software projects in productive use, as they were mainly created for other research projects or for experimental purposes.

To evaluate the tool's performance, all files were run through it in an attempt to calculate the similarity scores for all combinations of files. The results were saved in a CSV file, which was then inspected regarding the distribution of results to find unusual behavior or accumulations. Neither were found, but the thresholds between the different classifications, high, medium, low or no similarity have been adjusted. Also the weighting of the class names in the calculation of the scores was reduced.

Summary and achieved KPIs

The functional requirements defined for Software AG's use case along with their KPI base, target and achieved values are shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**. Apart from one skipped requirement and one partially fulfilled KPI, all KPIs achieved their target value.

Table 14: Functional requirements of the Software AG use case

Req.	Tools	Solution partner	KPI Definition	Base Value	Target Value	Achieved Value
UC6.FR1	all		Automated processing and analysis of artefacts recorded both in semi-formal and natural language shape to gather the necessary information for further advice on decision taking by management, development, and testing	0%	100%	100%
UC6.FR2	SoHist	UIBK	Running estimate of the overall product code quality and trend for the management	0%	100%	100%
UC6.FR3	CSI	TWT	Actionable indication of the source code fragment's quality	0%	100%	80%
UC6.FR4	CSI	TWT	Automatic selection of "problematic" code fragments based on existing fixes' analysis	0%	100%	80%
UC6.FR5	AISA	IFAK	Automatic suggestions for design and code reuse based on the natural language description of feature requirements	0%	100%	100%
UC6.FR6	AILA	IFAK	Automatic suggestions for issue categorization (e.g., security labels) based on the issue's title and description	0%	100%	100%
UC6.FR7	N/A	N/A	Prioritization of tests within the test base of a product by their relevancy to the feature description and/or code changes	0%	100%	N/A
UC6.FR8	AISA, CSI	IFAK, TWT	Suggest tests for reuse based on feature description and/or code changes	0%	100%	100%

Further details about the achieved KPI values are listed below:

- UC6.FR1: Once setup and configured, all tools work fully automatic without requiring further manual input. The results reported by the tools still need to be manually investigated and processed.
- UC6.FR2: SoHist in combination with SonarQube allows us to measure code quality at specific timepoints and visualizes the overall code quality trend over time.
- UC6.FR3/UC6.FR4: Comparing source code fragments with existing code fragments known to be affected by a bug or other issue allows us to identify if the input fragments might also be affected by the same issues and to measure their quality. TWT's tool CSI enables this comparison but it is limited to code comparisons on class or method level.

For our desired application scenarios, this limitation does not have a huge impact. Since our original requirements target arbitrary unrelated code fragments, however, we consider them to be partially achieved (80%).

- UC6.FR5: The AISA tool developed by IFAK enables us to assess the similarity of issues written in natural language. The found similar issues in turn can point us to design and code artefacts that can potentially be reused to address the corresponding other issue.
- UC6.FR6: The AILA tool developed by IFAK showed very good results in classifying issues as security-related or not.
- UC6.FR7: This low priority requirement has been finally skipped in favor of focusing on improving the results for the other requirements.
- UC6.FR8: As outlined above, the AISA and CSI tools allow us to compare issues written in natural language as well as code to identify similar artefacts. The reported similar artefacts and their links to tests can point us to tests that could be reused.

g. Recommendation for industry adoption

The application of the SmartDelta Methodology and the associated tools helped us to successfully overcome the challenges described in our use case. The SmartDelta Methodology not only provided us with structured guidelines to tackle the challenges, it also demonstrated new ways of addressing them such as suggesting approaches for the automatic comparison of software artefacts effectively reducing manual effort. This is key, especially when dealing with massive amounts of software artefacts that cannot be processed manually due to their sheer size like in our case.

The tools developed in SmartDelta showed very promising results. AILA enables to automatically and accurately assign security labels to issues. This leads to faster assignment of issues to the relevant experts and in turn faster treatment, which helps to improve the overall security and quality of the software. Moreover, the tool is not only limited to the prediction of security labels. It could also be used for other classification tasks such as team assignment in general.

The issue and code comparison tools AISA and CSI along with the recommendations from the SmartDelta Methodology provide the basis for various follow-up tasks and benefits. This includes, e.g., reuse of artefacts such as design, code, and tests to reduce repetition saving time and costs as well as code quality assessment and error prevention for improved software quality.

These features enable companies to develop software of higher quality in less time by leveraging existing knowledge and data as well as reducing manual effort. This results in improved product quality, reduced costs, faster time to market and ultimately in increased customer satisfaction, which is relevant for any company developing software, regardless of the company's size.

9. Use-Case 7 from c.c.com

a. Use-Case Description

Located in Styria, Austria, c.c.com Moser GmbH (hereafter referred to as c.c.com) is specialised in developing systems for measuring traffic volumes along roads. Therefore, we use (1) Bluetooth- and Wi-Fi access points to collect information about the number of vehicles passing a specific point of interest. Subsequently, (2) data is given to the BLIDS IoT Elastic Cloud, (3) which performs mathematical-statistical models to (4) retrieve traffic density statements via the BLIDS Web platform.

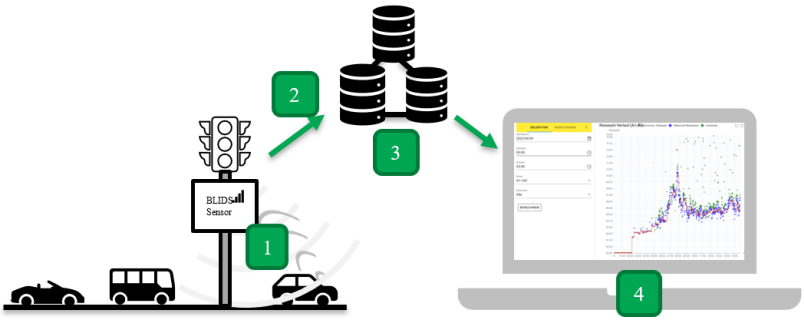


Figure 764: c.c.com from Measurements to Plattform

Consequently, a large codebase is needed, covering functionalities for individual sensors as well as for managing the entire pipeline, including data retrieval, computations, and visualization across different platforms for customers.

Within the SmartDelta project, our business objective had been to improve different individual quality aspects of their software solution and their monitoring concept, specifically using historical data. Mainly, we focused during the project progression on the following elements and gained the insights into

- **Anomaly Detection Parameters for Abnormal Behaviour Detection:**
  In the last years, we had a few times some anomaly – behaviour that was not expected. This resulted, for instance, in sensor data not being processed properly for a short period. In a worst-case scenario, the system could fail for a long time e.g. if the sensors are out of battery. However, in most cases, we were able to detect and resolve issues quickly. Nonetheless, we continuously strive to improve our approaches. Therefore, we focused on exploring new methods for runtime anomaly detection and identifying key monitoring parameters essential for setting effective thresholds and rules.

- **Improved Time-Series Visualizations to Represent Quality Assurance Metrics:**
  We operate multiple BLIDS sensor stations that continuously measure traffic flow and log sensor condition data. Additionally, our server cluster monitors and provides health quality metrics. In this way, we aimed to analyse and visualize long historical and real-time data in WP5.

- **Software-Related Retroperspective Quality Insights:** On the software level, one interest was to systematically collect software quality metrics over time, analyze the evolution of a project's quality, and identify potential areas for improvement. To achieve this, we required a tool capable of conducting historical code analysis while supporting multiple programming languages. Specifically, we were interested in how code quality is correlated to sensor conditions. We did not find a suitable tool that could meet these requirements, making it difficult to track and compare code quality across various stages of development if tools like SonarQube had not been employed since the beginning.

Due to changes and provided solutions, we addressed 2 of 4 user stories: *Story A: Finding metrics to detect early critical scenarios on their BLIDS IoT Elastic Cloud,* and *Story D: Needing a visualization dashboard to represent quality assurance metrics*.

b. Link to SmartDelta Methodology

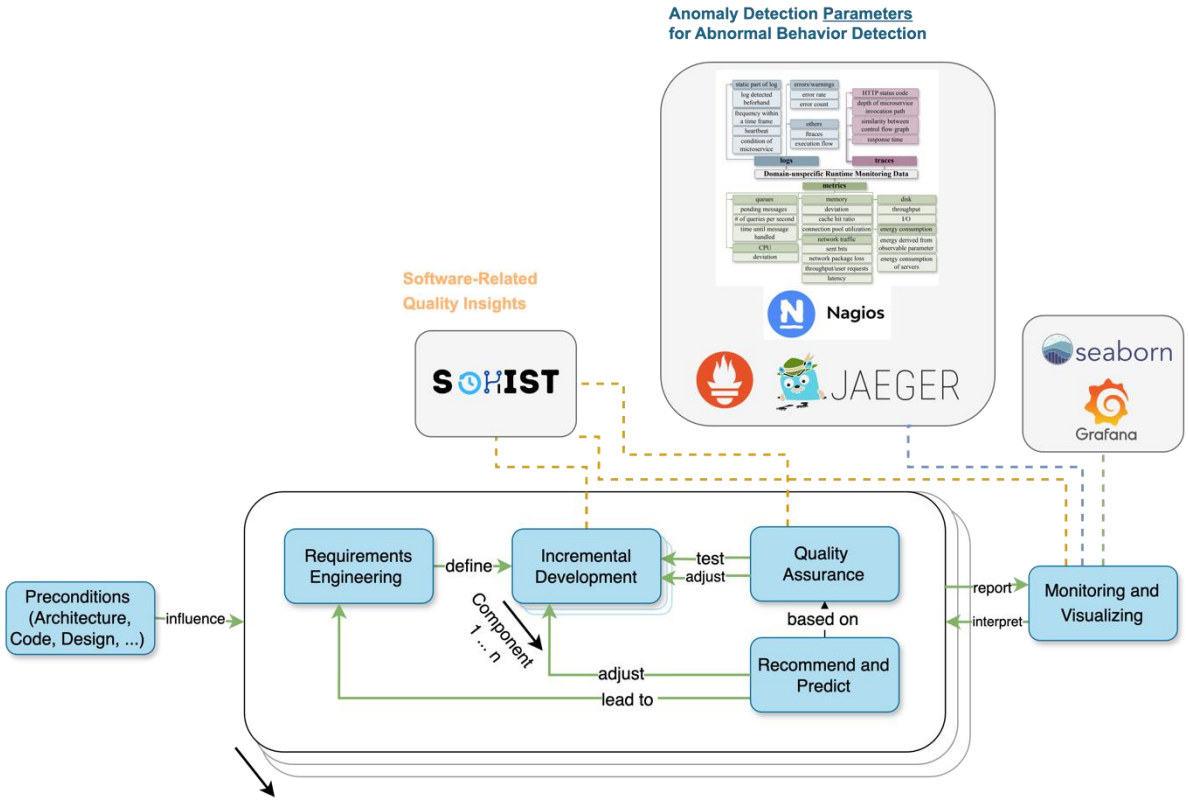


Figure 775: Mapping of c.c.com tool stack, improved by SmartDelta outcomes

During the project, the new SmartDelta methodology was developed as a demonstrator to map tools and provide insights, helping to identify specific *Delta aspects* in a software development lifecycle. In our use case, we can map the different goals to the different stages, as we had a variety of outcomes.

Specifically, we track our code updates as commits on GitLab. Each change builds incrementally on the previous version, creating a history of our codebase. By integrating *SoHist,* we can effectively monitor quality changes over time. In this case, we address the *Incremental Development*, *Quality Assurance*, and *Monitoring and Visualization* stages.

Additionally, we employ Nagios, Jaeger, and Prometheus with Grafana to monitor the system and identify possible abnormal behaviour. Nagios handles anomaly detection with rule-based alerting, Jaeger facilitates distributed tracing to track request flows across microservices, and Prometheus collects metrics and enables alerting for proactive system health monitoring. In all these monitoring solutions, we need the correct parameters to collect and monitor. Therefore, the *Monitoring and Visualisation* stage is mainly involved.

Finally, we focused on the visualization aspect. Our efforts centered on (1) the visualization of code metrics over time (SoHist) (2) the visual analysis of long-term sensor data provided through logs, and (3) the representation of short-term information for analysis.

c. Tools descriptions

We have been extensively involved in the development of **1. SoHist v2**, building it from the ground up by actively participating in coding and testing while incorporating feedback from *SoftwareAG*. Additionally, we have contributed to **2. deriving industry-relevant insights** from approaches used for runtime monitoring data in anomaly detection. Furthermore, **3. we have developed a tool to update and test BLIDs sensors with new versions**. These contributions are discussed in more detail below:

1. **SoHist**

To ensure this high code quality, the open-source software quality framework *SonarQube* (Community Edition) can be helpful. Running SonarQube static analysis capabilities, triggered by a commit on GitHub or GitLab, gives you an "up to date" profile about your current code artifact and its correlated measurements of technical debts.

Nevertheless, having a current analysis is good, but seeing the projects' total history is even better. In such a way, you can build up your project´s code quality evolution graph and use it for other analysis approaches (e.g., finding correlations between code metrics and performance parameters like CPU usage or energy). SonarQube's intended approach currently leads to limitations and challenges (see *D3.4 - Delta-oriented Quality Assurance Methodology*) in evaluating a whole Git repository, which we tried to resolve with SoHist.

SoHist v1 is a tool that extends SonarQube's capabilities to provide historical code analysis, tracking metrics like Technical Debt (TD) over time. It is containerized and deployable via Docker, integrating SonarQube, a database, SonarScanner, a Git interface, and a web UI. Users connect SoHist to Git, set analysis parameters, and trigger code analysis. SoHist ensures consistent metrics across versions and offers two visualizations - Code Evolution and Weighted Code Evolution Significance - to track and assess code quality changes.

More can be found in Deliverable

- *D3.4 - Delta-oriented Quality Assurance Methodology*.
- *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*

In ***SoHist v2*** **(2024)**, a new quality assessment feature was introduced. This feature leverages a dataset of over 2,006 SonarQube projects historically evaluated and meet specific quality criteria, including more than 100 commits, at least 4 contributors, and a minimum of 10 GitHub stars. With this data, users can benchmark their project's quality metrics against a broad range of comparable projects, considering various programming languages and selected quality criteria. A chart visualizes the distribution of these metrics across other projects, providing insights into key metrics such as Test Coverage, Code Smell Density, and more.

More can be found in

- Deliverable *D4.5 - SmartDelta Quality Optimization and Recommendation Methodology*
- *Dornauer, B., Felderer, M., Saadatmand, M., Abbas, M., Bonnotte, N., Dreschinski, A., Enoiu, E. P., Tüzün, E., Uçar, B. M., Devran, Ö., & Gröpler, R. (2024). SmartDelta Methodology: Automated Quality Assurance and Optimization for Incremental System Engineering.* In Proceedings of the 22nd International Conference on Information Technology: New Generations (ITNG 2025).

2. **Approaches for Runtime** Monitoring Data for Anomaly Detection (e.g., Nagios, …)

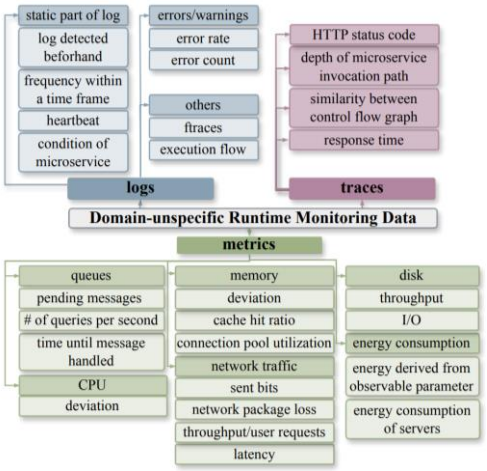


Figure 78: How Industry Tackles Anomalies during Runtime: Approaches and Key Monitoring Parameters

Deviations from expected behavior during runtime, referred to as anomalies, have become increasingly prevalent due to the growing complexity of systems, particularly in microservices architectures. Analyzing runtime monitoring data - such as logs, traces for microservices, and metrics - poses challenges due to the vast volume of data collected. At c.c.com, Nagios is utilized to detect anomalous behavior through predefined rules and thresholds.

Through the SmartDelta initiative, c.c.com has gained new experience and insights into industry practices for anomaly detection. Therefore, we have collaborated with the University of Innsbruck as an academic partner and other industrial partners, in total 12, such as Vaadin, Akkodis or Hoxhunt. The partners shared valuable insights from their experiences, prompting us to explore the prevailing approaches in anomaly detection—whether they are predominantly AI-driven or still largely rule-based. One main objective of this collaboration was to develop a comprehensive list of parameters that can significantly enhance anomaly detection processes.

More can be found in

- *M. Steidl, B. Dornauer, M. Felderer, R. Ramler, M. -C. Racasan and M. Gattringer, "How Industry Tackles Anomalies during Runtime: Approaches and Key Monitoring Parameters,"* 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Paris, France, 2024, pp. 364-372, doi: 10.1109/SEAA64295.2024.00062.*

3. Rust CLI Tool (individual tool by c.c.com)

Before a temporary BLIDS sensor is sent to a customer with the latest firmware version, we must test and evaluate its functionality. Prior to SmartDelta, this process was conducted manually in small steps by a tester, requiring significant human effort. One of SmartDelta's objectives was to automate this process to reduce manual workload.

To achieve this, we selected Rust as our primary language and developed an automated CLI tool. After configuring a sensor with the necessary parameters (e.g., name, SSH credentials, SSH

server address, Bluetooth count, attenuation, etc.), the client tools can be executed to perform all required tests, including network connectivity via HTTPS, attenuation checks, SD card verification, Bluetooth functionality testing and so on.

Afterwards, the sensors can be directly incorporated into the BLIDS pipeline.

```
jweinzerl@weinzerl-desk:~$ /workspace/blids-sensors-check/blids-sensor-checks -h
2024-10-02T11:13:51.697725+02:00  INFO blids_sensor_checks: started blids-sensor-checks 0.2.0
Usage: blids-sensor-checks [OPTIONS] -c <CONFIG>

Options:
  -c <CONFIG>
  -t, --task <TASK>      The name of the task to be executed. Reuse the argument to define multiple tasks. Leave it empty and all tasks starting
with check will be executed. The tasks will be executed in the order they were specified [possible values: check-http, check-https, check-ssh, ch
eck-server-connection, check-ppp, check-attenuation, check-dump-config, check-sd-card, check-watchdog, check-backup, check-bluetooth, check-ble,
check-logs-ntp, check-logs-voltage, check-logs-temperature, check-logs-bluetooth, check-logs-ble, check-logs-gps, check-logs-config, check-logs-d
ebug, check-logs-online, remove-backup, reboot]
  -s, --sensor <SENSOR>  The name of the sensor from the specified config to run tasks against. Reuse the argument to define multiple sensors. Le
ave it empty and all will be executed
  -h, --help             Print help
jweinzerl@weinzerl-desk:~$
```

Figure 79: Bllids-sensor-checks client tool usage.

d. Visualization

We have collected various data related to:

- Server utilization,
- Individual sensors, and
- The quality of our software artifacts over time.

The observed deltas represent changes in specific properties over time. To analyze these variations effectively, we explored multiple visualization approaches and identified the most suitable one for our requirements. Additionally, we have tested Vaadin - see Dashboard Solution for more details.

(1) We **chose Seaborn in Python** to initial visualize collected logs exceeding 100GB. This enabled us to represent individual sensor properties - such as temperature, CPU capacity, and memory allocation. A key advantage was the ability to preprocess the data given as logs using pandas before generating visualizations with Seaborn. Given the size of our dataset, this approach proved to be effective and well-suited to our needs. Thus, we have also integrated the visualisation into Grafana.

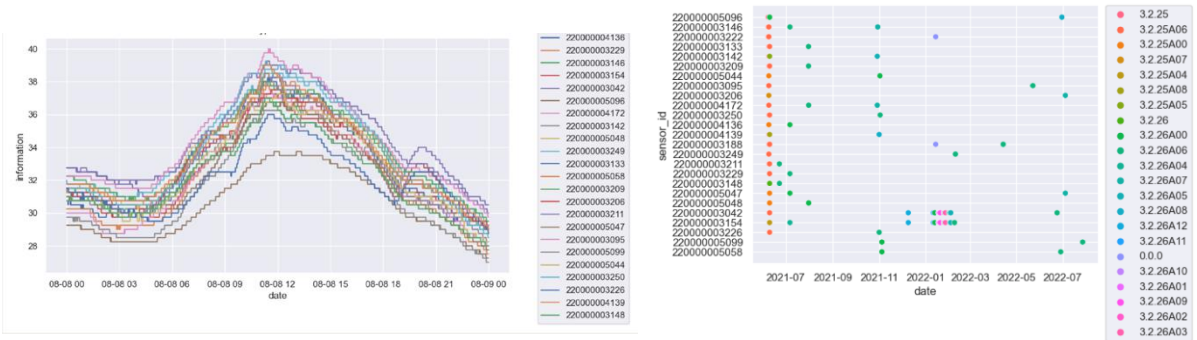


Figure 80: Visualising sensor properties with Python Libraries (left: temperature per sensor, right: sensor version changes over a year)

(2) We have further **improved the Grafana's capabilities** to enhance visualization features for the sensors and BLIDS IoT Elastic Cloud, providing a clearer overview of ongoing changes. Based on our experience, we will continue prioritizing Grafana, as it facilitates real-time monitoring and dashboard-based visualization without requiring additional programming.

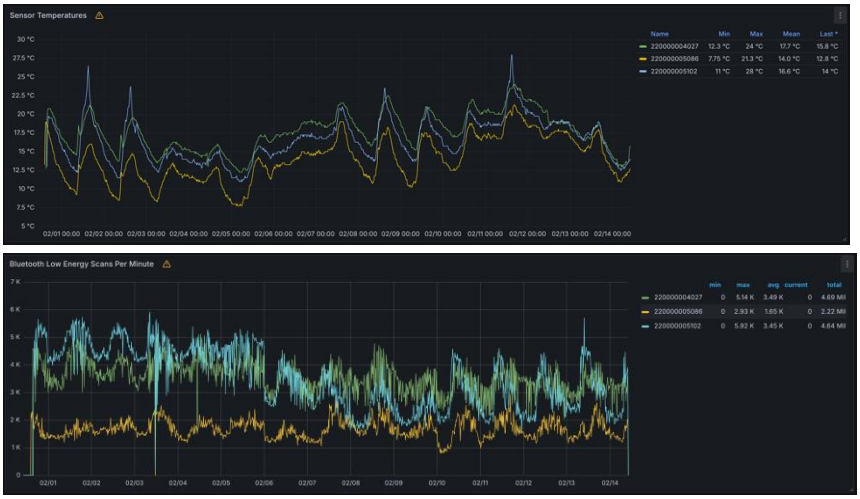


Figure 81: Enhanced visualization of sensors and BLIDS

So far, we have used and extended Grafana to create time-series visualizations for both our individual sensors and the entire server cluster. A live-demo is given here. As part of Work Package 5, we also experimented with Vaadin Charts to evaluate new technologies. The necessary infrastructure and required licenses were provided by our partner Vaadin. We created charts that we also implemented in Grafana.

Compared to Grafana, we see the main advantages of Vaadin Charts in its flexibility and customizability. However, the code-specific setup is significantly more complex, making integration or replacement within our existing infrastructure impractical.

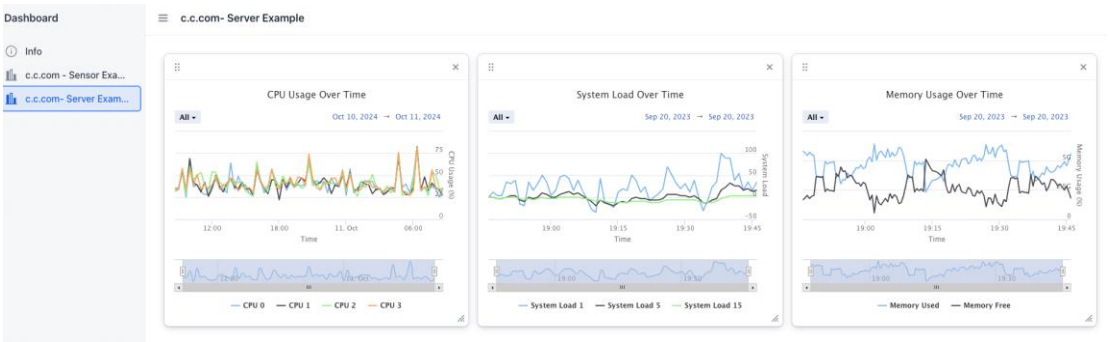


Figure 82: Comparison with Vaadin solution

(3) Apart from the *SoHist* provided views to see how individual commits on the main branch affected the code quality (see Work Package 3 und 4 for details).

e. Evaluation Setup

Our evaluation process was mainly iterative (SCRUM-like), with regular feedback and ongoing improvements. We worked very closely with our partner, the University of Innsbruck (subcontracted), ensuring they met our goals and helping them in the research activities. Through discussions, we identified areas for optimization, refining key components to improve overall quality, often in overlapping roles as both use case and solution provider, for instance, for SoHist.

To align with WP1's formal requirements and establish a clear evaluation framework, we conducted an evaluation. This included defining functional requirements and setting KPI values, which we gathered from technical experts' experience. This initial input served as the baseline for evaluating progress.

At the end of 2024, we conducted a reassessment to determine how the requirements and KPIs had evolved throughout the SmartDelta project. By comparing the initial values (Base Value) to the results (Achieved Value), we were able to gain insights into how well we had met our objectives and identify areas for further improvement.

f. Evaluation results

In this hybrid setting, involving academic partners (specifically UIBK) and use case providers, we supported the work of other partners and learned from each other. Besides that, we pushed our work to improve in what we do, which is most often code related.

Before this project, we relied solely on Linter tools integrated into our IDEs. However, these tools did not provide insights into the entire codebase, notably lacking a retrospective view. With ***SoHist***, we could now connect to our GitLab instance via a user interface, select specific properties such as a computer or branch, and perform code analysis using SonarQube in the background, if necessary. As a result, we can gain a history of key code quality aspects in the different phases of our development. See for details here.

We were also interested in improving our **anomaly detection approaches**. Thus, we pushed the research on "How Industry Tackles Anomalies during Runtime: Approaches and Key Monitoring Parameters." This allowed us to identify the advantages and disadvantages of AI-based vs. traditional rule-based approaches. Based on insights from interviews and the literature, we concluded that we would proceed with the rule-based approach, as the industry has not yet consistently implemented AI solutions. One of the key outcomes of this activity was finding a categorized list of parameters (see SEAA paper), along with detailed information, for runtime monitoring data. By gathering this information, we now have a lot of new insights for future work.

Within SmartDelta, we have also **enhanced our visualization approaches for time-related changes**. This included a detailed historical data analysis using Python, which was the foundation for improving our Grafana visualizations. Additionally, we explored Vaadin Charts as an alternative and conducted thorough testing. See *Visualisation and Requirements* for more insights and charts.

The newly developed **CLI tool for testing our temporary BLIDS sensor** helped us improve efficiency by automating previously manual tasks. This saved time and reduced errors, allowing our team to focus on more important activities and significantly increasing overall productivity.

Table 15: KPIs Overview

	Tool	Partner	KPI Definition	KPI Base Values	KPI Target Values	KPI Achieved
FR1	Jupyter Notebook	UIBK	We aim to extract from sensor logs at least the following parameters: power consumption, the Received Signal Strength Indicator (RSSI), modem quality and temperature into an analyzable data structure for at least one year.	unstructured log files	analysable data structure of at least one year	analysable data structure of at 1 ½ year
FR2	Jupyter Notebook	UIBK	We aim to analyze in detail the historical data for at least one year for each sensor configuration. We aim to have in total a data coverage of 90% of all log files.	0%	90%	100%
FR3	Rust CLI Tool (individual tool by)	-	Implementing this functional requirement will support software engineers in comparing/updating software versions and configuration settings (Sensor Testing). This will save 50% workload spend every week on those tasks.	30 h/week (sum)	15 h/week (sum)	Yes, now 3 h/week (10%)
FR4 / FR 5	BLIDS-Portal Extension + Grafana Improvements	-	We expect to get a total overview of (FR4) each sensor / (FR 5) sensor cluster utilization. We want to reduce the current effort to gather the necessary information by 75%.	10 h/week (sum)	2.5 h/week (sum)	9 h / week
FR6	Metrics Overview	UIBK (in coop. with Akkodis, Vaadin, Hoxhunt)	We aim to find possible metrics that could be an indicator for possible anomalies and possible correlations between those, using Prometheus and Jäger. Thereby, we expect to get experience to reduce the average downtime.	No list of metrics.	List of metrics.	List of metrics.
FR7	Grafana	UIBK	We aim to have a visual dashboard that helps to get a quick synopsis of the sensors' past and current (live) metrics. In	12 h/ week	2 h/week	8h/week.

	Tool	Partner	KPI Definition	KPI Base Values	KPI Target Values	KPI Achieved
			multiple circumstances, the dashboard could reduce the human effort spent every week by 85 % for those tasks.			
FR8	SoHist	UIBK	We would like to be able to analyze the whole GIT repository for each BLIDS IoT sensor version to be able to find possible changes for different code quality aspects.	0 % analysis	100 % of commits	100 % of commits
FR9	Grafana	UIBK	The visual dashboard should also contain information about the past and current BLIDS IoT elastic cloud measurements. We want to reduce the current effort to overview the information provided by 75%.	4 h / week	1 h / week	3h / week
FR10	n/a	n/a	The planning effort for new BLIDS projects is high. One problem is the approximation of the increased utilization by each project. Within SmartDelta, we assume to reduce the current time effort spent by 50 % on those activities	20 h / project	10 h / project	n/a

g. Recommendation for industry adoption

Within the project, we have focused on various aspects to enhance our solutions and systems. However, we have only leveraged a portion of the potential and knowledge offered by our numerous partners. In our view, the ***SmartDelta Methodology*** now provides a comprehensive overview that will be valuable in the future for identifying relevant tools and solutions in the development lifecycle.

10. Use-Case 8 from Glasshouse

a. Use-Case Description

Glasshouse Systems is enhancing Security Operations Center (SOC) capabilities by integrating machine learning (ML) models within IBM QRadar to enable real-time anomaly detection and

offense prioritization. This initiative aims to improve incident management efficiency and accuracy by automating key processes, reducing manual effort, and optimizing threat response times.
With a rapidly evolving cybersecurity landscape, SOC analysts must quickly identify and prioritize threats while handling high volumes of events per second (EPS). The integration of ML-based anomaly detection within QRadar enables SOC teams to effectively manage cybersecurity events, reducing noise and ensuring critical incidents receive immediate attention.

Challenges and Motivation

The Glasshouse use case addresses four key challenges structured into user stories:

Story A: Managing High EPS with Precision

SOC environments often handle an overwhelming volume of security events, with logs and alerts continuously generated from various network and system activities. The challenge lies in ensuring that SOC analysts can efficiently identify, filter, and prioritize threats while maintaining accuracy and avoiding alert fatigue. The traditional approach relies on rule-based filtering, where predefined security rules trigger alerts based on recognized patterns. However, as cyber threats evolve, rule-based systems often fail to detect complex attack patterns and generate a significant number of false positives, leading to unnecessary investigations and wasted resources.
With the implementation of ML-enhanced anomaly detection in QRadar, the system can intelligently differentiate between normal and suspicious activities by learning from historical data. Instead of relying solely on predefined rules, the machine learning models dynamically adapt to changing attack behaviors, improving detection accuracy. This reduces the burden on analysts by filtering out irrelevant alerts and allowing them to focus on truly significant threats. The primary objective of this use case is to evaluate how effectively ML can increase precision while handling high EPS, ensuring that security teams remain efficient even as network activity scales.

Story B: Faster Identification of Anomalies

One of the critical requirements of a modern SOC is the ability to detect anomalies in real time and respond before they escalate into security breaches. Traditional security monitoring tools rely on predefined indicators of compromise (IoCs) and manual investigation processes, which can lead to delayed threat detection. In fast-paced attack scenarios, such as zero-day exploits or advanced persistent threats (APTs), even small delays can result in significant data breaches and operational disruptions.
The current approach in Glasshouse's SOC involves manually reviewing logs and correlating security events across multiple platforms. This process is time-consuming and prone to human error, as analysts must sift through large datasets to distinguish normal variations from real threats. The integration of machine learning models within QRadar enhances this capability by continuously analyzing incoming logs, identifying deviations from expected patterns, and flagging anomalies in real time. The ML models learn from past security incidents, refining their detection capabilities over time.
This use case aims to measure the effectiveness of ML-enhanced QRadar in reducing the time between an anomaly's occurrence and its detection. By leveraging automated anomaly detection, the system can alert analysts to potential security breaches faster, minimizing response times and improving overall SOC efficiency. The evaluation will focus on how ML-based detection compares to traditional rule-based methods in terms of speed and accuracy.

Story C: Accelerated Response to Critical Incidents

Identifying security threats is only one part of the challenge; the real test for a SOC is how quickly and effectively it can respond to incidents once they are detected. In traditional SOC workflows, security alerts must go through multiple stages, including investigation, triage, and response coordination, before action is taken. This manual process often results in delays, especially for high-priority incidents, allowing attackers to inflict damage before containment measures are applied.
The existing response framework at Glasshouse relies on manual offense prioritization, where analysts assess alerts based on predefined severity levels. However, this approach is not always effective, as it can misprioritize incidents, causing critical threats to be overlooked while lower-priority alerts consume valuable time.
By integrating ML-driven offense prioritization into QRadar, the system can automatically rank security incidents based on their threat level, ensuring that the most critical threats receive immediate attention. Machine learning algorithms assess multiple risk factors, such as the affected system's importance, historical attack patterns, and real-time anomaly scores, to determine the urgency of each incident.
This use case focuses on measuring the impact of ML-enhanced prioritization on incident response times. Specifically, it will evaluate how quickly analysts can triage, investigate, and mitigate threats when offense prioritization is automated, compared to traditional manual methods. The goal is to accelerate Mean Time to Detect (MTTD) and Mean Time to Respond (MTTR), improving overall security posture.

b. Link to SmartDelta Methodology

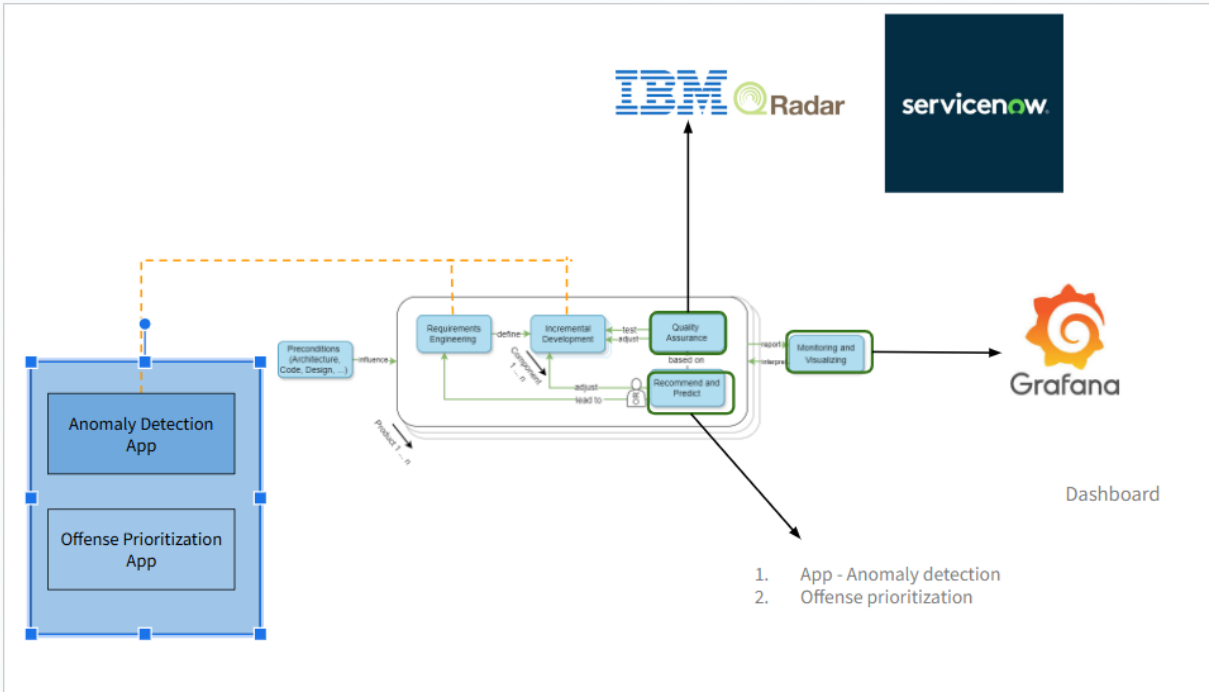


Figure 82: Mapping of Glass House tools, improved by SmartDelta outcomes

The SmartDelta methodology provides a robust framework for implementing solutions in the Glasshouse use case by emphasizing automated quality assurance and optimization. This approach enables systematic management of software quality across multiple versions and variants, utilizing smart analytics to examine development artifacts and operational data, helping identify areas of quality degradation and improvement. For Glasshouse, SmartDelta's methodology supports the ongoing refinement of our EPS Anomaly Detection and Offense Prioritization Tools, enhancing SOC performance through machine learning and real-time data analysis.

Our use case effectively leverages the SmartDelta methodology's product, time, and resource dimensions to improve SOC operations within Glasshouse.

1. **Product Dimension:** The EPS Anomaly Detection Tool and Offense Prioritization Tool have been designed with flexibility, catering to specific SOC requirements and aligning each tool's version to meet operational needs. By focusing on modular tool enhancements, we efficiently target unique security requirements in each development stage.
2. **Time Dimension:** Tracking changes over time allows our team to identify emerging trends and optimize SOC responses through continuous improvement. This dimension is especially beneficial in scenarios like anomaly detection and offense prioritization, where identifying trends in MTTD and MTTR over time supports ongoing adaptation of threat detection measures.
3. **Resource Dimension:** Automation, artifact reuse, and predictive analysis minimize the resource load in security operations, ensuring that analysts can focus on critical threats without sacrificing efficiency. This approach reduces manual processing time, while the meta-methodology guides the adaptation of resources to meet evolving SOC demands, maintaining an optimal balance of precision and resource allocation.

Resource-Related Delta Considering SOC Performance

We have implemented machine learning inside QRadar SIEM as an app to help SOC analysts detect and resolve offenses faster. We have developed two QRadar apps: the EPS Anomaly Detection Tool and the Offense Prioritization Tool. These tools align with the SmartDelta methodology by providing automated quality assessments and smart analytics from development artifacts and system executions.

Our development process involved rigorous and strict stages, including:

- **Prototyping**: Initial prototype creation to test core functionalities.
- **Alpha Version**: Developing an alpha version for initial feedback.
- **Client Feedback**: Incorporating feedback from clients to refine the tool.
- **Lab Environment Testing**: Extensive testing in a controlled lab environment.
- **Production Deployment**: Final deployment in a production environment.

We maintain a GitHub repository to push updates, continuously improving the codebase and implementing software quality best practices.

Software-Related Delta Considering SOC Performance

To ensure efficient memory usage, we evaluated various ML models and selected those that minimize storage requirements. Initially, some models were cost-prohibitive in terms of storage. We are now utilizing AQL queries and the ServiceNow API to access data. The Grafana dashboard is integrated with ServiceNow APIs to provide real-time monitoring and historical data analysis.

Metrics for Evaluating SOC Performance:

- **Ingested Events Per Second (EPS)**: Evaluate the SOC's ability to process an increased volume of log records.
- **Mean Time to Detect (MTTD)**: Measure the time from log ingestion to security incident alerting.

- **Mean Time to Respond (MTTR)**: Measure the time from incident detection to response initiation.

By integrating SmartDelta's meta-methodology into Glasshouse's tools and KPIs, we align each development phase to improve SOC resource efficiency and incident response time, creating a sustainable and adaptable incident management ecosystem.

c. Tools descriptions

Our tools play a vital role in enhancing each stage of the incident management process, ensuring efficiency and reliability. The EPS Anomaly Detection Tool enables SOC analysts to swiftly diagnose incidents by analyzing event-per-second (EPS) data, while the Offense Prioritization Tool helps expedite the escalation of high-priority incidents to relevant teams. ServiceNow integration supports timely stakeholder communication, and continuous monitoring assists teams in accurately identifying incident root causes. Real-time alerts facilitate quick resolutions, with KPIs like MTTD and MTTR guiding effective closure, ensuring quality service and optimized SOC operations.

EPS Anomaly Detection Tool

- Purpose: Monitors incoming events per second (EPS) to identify anomalies in SOC environments.
- Implementation: Python-based ML model to detect true and false positives with interactive charts highlighting potential anomalies.
- Impact: Enables rapid anomaly detection, ensuring the SOC can handle increased EPS volume without compromising response precision.

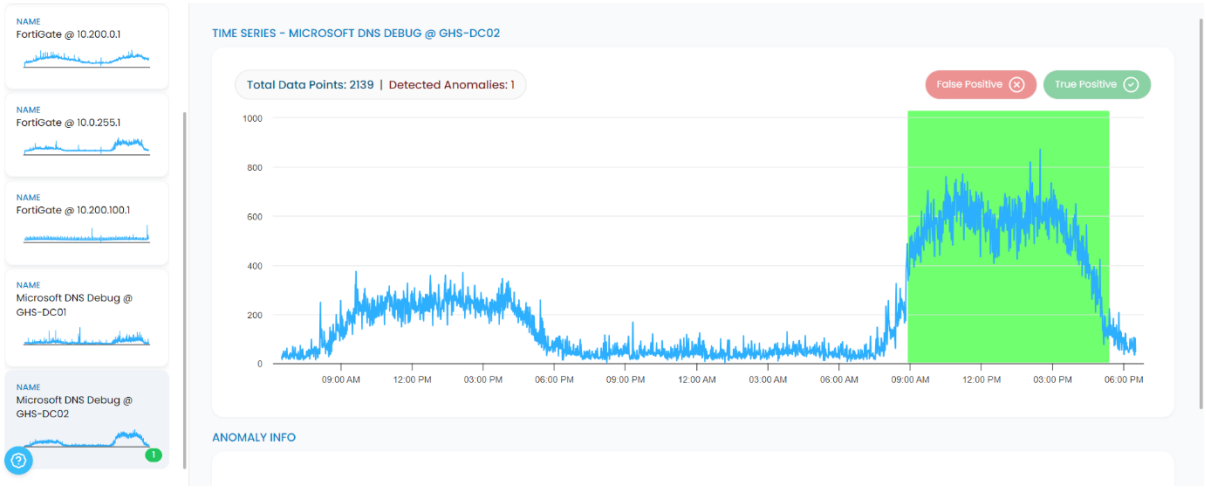


Figure 83: EPS tool

Offense Prioritization Tool

- *Purpose*: Helps SOC analysts prioritize incidents based on risk level, enhancing response efficiency.
- *Implementation*: Uses machine learning to assess offenses and assign risk scores, facilitating faster identification and escalation of critical incidents.
- *Impact*: Reduces analysis time significantly, allowing analysts to focus on high-risk incidents and improving overall SOC responsiveness.

d. Visualization

Figure 84: SOC Dashboard

Dashboard solution

Our SOC dashboards, powered by Grafana, integrate with ServiceNow APIs to visualize key performance indicators (KPIs) such as Mean Time to Detect (MTTD) and Mean Time to Respond (MTTR). These visualizations provide a real-time view of SOC performance, enabling analysts to monitor incident management progress effectively.

Visualisation requirements

The visualization setup is designed to highlight priority incidents, track response times, and provide historical and current trend analysis. The dashboard simplifies data interpretation, allowing quick decision-making for resource allocation and response strategies.

e. Evaluation setup

The SOC enhancement solutions implemented within this use case operate on on-premises servers and are designed to integrate seamlessly with existing SIEM infrastructure. The evaluation process is structured into three key stages: Pre-Processing, Processing, and Post-Processing.

Pre-Processing

In this phase, **log records (EPS)** and offense data are **collected from QRadar** and other integrated sources. The **EPS Anomaly Detection Tool** processes raw event logs, filtering out **false positives** before the data is passed into the machine learning pipeline. On the other hand, the offense prioritization tool process the related events of each offense. Key modifications during this stage include:

- **Normalization and parsing of log records** for structured analysis.
- **Application of anomaly detection models** to pre-filter noise.
- **Correlation of events** to establish potential security incidents.

Processing:

Once pre-processed, the data is transferred to the **AI-powered tools**, where ML models analyze event patterns and predict anomalies and the severity of security incidents. This stage involves:

- **Real-time correlation of security incidents** using SIEM capabilities.
- **Machine learning-driven anomaly detection** to prioritize threats.
- **Calculation of MTTD and MTTR improvements** based on historical benchmark data.

Post-Processing:

In the final phase, **evaluation metrics are generated** and visualized to assess the effectiveness of the implemented ML-based solutions. The results are aggregated and compared against baseline values to measure impact. Key deliverables at this stage include:

- **Performance reports showcasing reduction in false positives.**
- **Comparison of manual vs. automated offense prioritization.**
- **Visualized dashboards displaying achieved improvements in MTTD and MTTR.**

By following this structured evaluation setup, SOC analysts can leverage **SmartDelta's methodology** to assess the efficiency and accuracy of **ML-driven offense prioritization**, ensuring a more **responsive and adaptive security operations workflow**.

f. Evaluation results

Our tools are evaluated based on KPI targets from D1.4:
EPS Anomaly Tool: Successfully implemented in production, with significant improvements in anomaly detection accuracy.
Offense Prioritization Tool: In beta, this tool demonstrates a 60x faster average risk score assessment compared to analyst evaluations, effectively reducing Mean Time to Detect (MTTD) and Mean Time to Respond (MTTR) through prioritized and precise offense escalation.

Table 16: KPIs Overview

Req.	Tools	Solution partner	KPI Definition	Base Value	Target Value	Achieved Value
UC6.FR1	EPS and Offense app	Glasshouse	SIEM Solutions normaly use a metric of Ingested Events Per Second that define the volume of log records received to be analyzed. These Events (Log Records) are then analyzed and correlated by the SW engines built in to the SIEM to detect Security Incidents. GHS SOC measure Resource capacity to handle Security Incidents generated by 3500 EPS. We expect this capacity to grow by 15% with the introduction of the SIEM AI/ML Enhancement functionality.	0%	>= 15%	100% See Explanation
UC6.FR2	Offense app	Glasshouse	Upon receiving Log Records (Events) by the SIEM, the SIEM will perform parsing and correlation until it identifies a potential Security Incident. A SOC Analyst is then engaged to further triage and investigate this incident prior to escalating and initiating incident remediation activities. MTTD is measured as the time between Log Ingest and Security Incident alerting. We expect the MTTD to be reduced by at least 20%	0%	<= 20%	100%

UC6.FR3	Offense app	Glasshouse	Upon receiving Log Records (Events) by the SIEM, the SIEM will perform parsing and correlation until it identifies a potential Security Incident. A SOC Analyst is then engaged to further triage and investigate this incident prior to escalating and initiating incident remediation activities. While MTTD is measured as the time between Log Ingest and Security Incident alerting. MTTR is measured as the time between the Detection and the initiation of the Response. We expect the MTTR to be reduced by at least 20%	0%	<= 25%	100%

UC6.FR1 - EPS and Offense App:

The evaluation of **FR1** demonstrates that with the introduction of the **EPS Anomaly Detection Tool**, the **SOC analysts** can **identify anomalous events more efficiently**. While the **volume of EPS** remains consistent, the tool helps **filter out false positives**, significantly reducing the time spent investigating non-critical incidents. This results in a **better allocation of resources**, as analysts can focus their efforts on more relevant and higher-priority incidents, ultimately improving **efficiency in threat detection**.

UC6.FR2 - MTTD (Mean Time to Detect):

For **FR2**, the focus is on reducing the **Mean Time to Detect (MTTD)**, which is the time between **log ingestion** and **alert generation**. With the integration of **machine learning models**, the system now makes predictions like those of human analysts but is able to process data **60 times faster**. As a result, the tool can quickly detect and flag potential threats, significantly reducing the MTTD.

UC6.FR3 - MTTR (Mean Time to Respond):

For **FR3**, the goal is to reduce the **Mean Time to Respond (MTTR)**, which is the time between **incident detection** and the **initiation of the response**. Similar to MTTD (As response as the logical next step after alerting), the machine learning models predict the prioritization of incidents **faster than human analysts**, allowing the **SOC team** to respond to critical incidents more promptly.

g. Recommendation for industry adoption

The application of the SmartDelta methodology and the associated tools has significantly enhanced Security Operations Center (SOC) capabilities by improving incident response times, optimizing resource management, and enhancing overall operational efficiency. The structured guidelines provided by SmartDelta have proven instrumental in tackling key cybersecurity challenges, particularly in handling large volumes of security events in real time.

One of the key benefits of this methodology is its ability to automate anomaly detection and offense prioritization, reducing manual effort and enabling faster, more accurate threat detection. Traditional SOC workflows often rely on rule-based detection mechanisms, which can be time-consuming and prone to errors due to alert fatigue and evolving attack techniques. By integrating machine learning-driven solutions, SmartDelta has introduced a dynamic approach to cybersecurity, allowing SOC teams to quickly identify, categorize, and respond to threats with greater precision.

The tools developed within SmartDelta have demonstrated strong potential for industry-wide adoption. The ML-enhanced QRadar framework enables automatic anomaly detection and offense prioritization, leading to faster incident response times. By automatically ranking security incidents

based on threat severity, SOC analysts can prioritize critical events without being overwhelmed by a high volume of security alerts. This approach also improves Mean Time to Detect (MTTD) and Mean Time to Respond (MTTR), ensuring that security teams remain proactive rather than reactive.

Furthermore, the adaptive learning capabilities of SmartDelta's tools ensure that SOC teams are always equipped to handle emerging threats. Unlike static detection rules, which require frequent manual updates, the machine learning models continuously learn from new security events, refining their detection capabilities without human intervention. This results in an intelligent SOC framework that evolves in real time, maintaining high levels of accuracy while reducing false positives.
Beyond SOC environments, these tools hold potential applications in broader cybersecurity operations, including automated vulnerability management, intrusion detection, and security compliance monitoring. Future development could focus on enhancing data integration across multiple platforms, allowing for seamless interoperability with various security information and event management (SIEM) systems. Additionally, continuous improvements in real-time threat intelligence processing will ensure that these solutions remain adaptable to the ever-changing cybersecurity landscape.

By leveraging existing knowledge, data, and automation, companies can significantly reduce manual effort, lower operational costs, and improve overall cybersecurity resilience. The adoption of SmartDelta's methodology and tools positions organizations to enhance their security posture, accelerate incident response times, and maintain a competitive edge in today's rapidly evolving threat landscape.

11. Use-Case 9 from Izertis

a. Use-Case Description

Izertis keeps track of software developments in private repositories. These repositories represent a valuable asset for the company, yet their potential for reusing software artifacts and automating deployment and testing processes remains largely untapped. Currently, accessing and leveraging the information within these repositories is a manual process. During the testing phase of similar projects, teams must rely heavily on existing documentation and the expertise of the development team. This approach results in a significant investment of time and resources where new software development is undertaken.

To address these challenges, Izertis plans to develop a platform to enhance the efficiency or the testing process: SONATA (Smart SOftware TestiNg MAnagemenT PlAtform). The main goal of SONATA is to enable efficient searching within Izertis`s developed projects to find developments like a new project or a new delta. SONATA aims to suggest relevant test cases that should be included in the new development or delta to ensure its quality.

Story A: Software Artifact Repository Management

Development teams at Izertis regularly create and maintain multiple software projects, each with its own code repositories, requirements, and test cases. Currently, there's no systematic way to leverage this valuable knowledge base, leading to redundant work and inefficient resource utilization.

The development teams need to manually search through existing projects to find similar implementations or relevant test cases, which is time-consuming and often incomplete. This manual

process means that valuable existing solutions might be overlooked, leading to unnecessary recreation of similar components and test scenarios.

Management is concerned about this inefficiency as it directly impacts project timelines and resource allocation. Without a proper system to map and connect related projects, the company cannot fully capitalize on its intellectual property, leading to increased development costs and longer time-to-market for new projects.

Story B: Semantic Project Similarity Detection

When starting new projects, developers spend considerable time analyzing requirements and designing solutions that may already exist in similar forms within the company's portfolio. The current process relies heavily on individual knowledge and memory of past projects, making it difficult to identify potential code reuse opportunities.
The development team needs an efficient way to match new requirements against existing projects. Manual searching through documentation is time-consuming and often misses relevant matches, resulting in redundant development effort and inconsistent solutions across projects.

Management requires better visibility into code reuse opportunities to optimize resource allocation and maintain consistency across projects. Without automated similarity detection, the company cannot effectively leverage its existing codebase, leading to increased development costs and missed opportunities for standardization.

Story D: Automated Testing & Recommendations

When developing new features or modifications, testing teams must manually determine which test cases to include. This process is subjective and time-consuming, often leading to inconsistent test coverage across similar components.

Developers and QA teams need guidance on which test cases are essential for new developments, but currently rely on their experience and documentation review to make these decisions. This approach can miss critical test scenarios and lead to quality issues being discovered late in the development cycle.

Management needs assurance that new developments are adequately tested and consistent with established quality standards. Without automated test recommendations, there's a risk of insufficient test coverage and inconsistent testing approaches across teams, potentially leading to increased maintenance costs and quality issues in production.

b. Link to SmartDelta Methodology

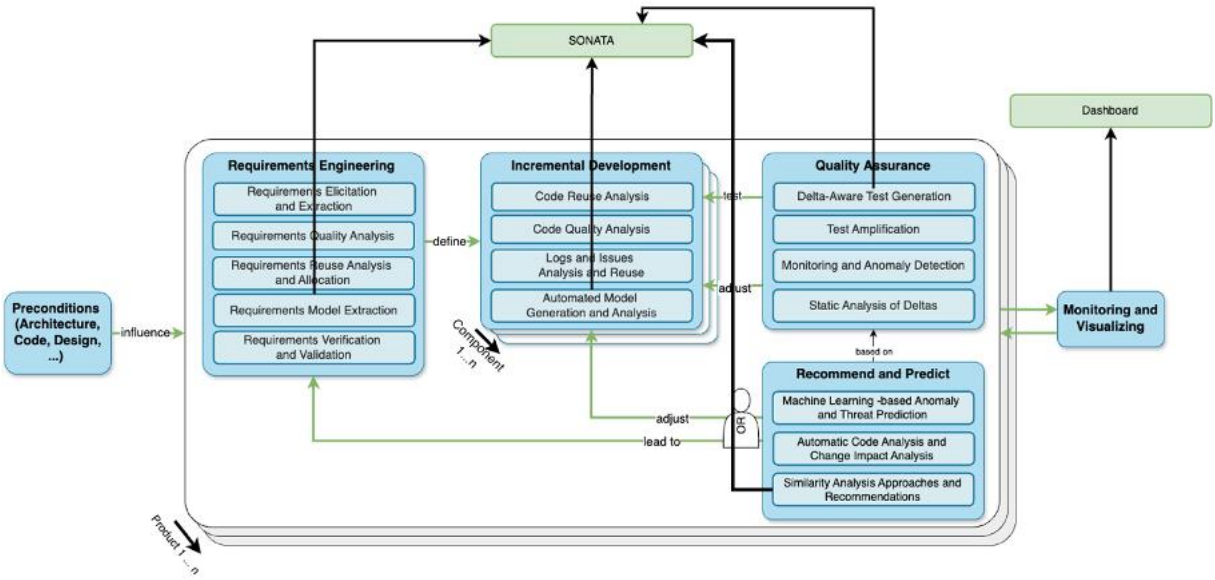


Figure 85: Tool Mapping with SmartDelta Methodology

In our use case, we employ the SmartDelta Methodology to effectively manage the deltas. Our use case makes use of all the stages of the methodology.

Requirements Model Extraction:

SONATA takes as input either a natural language query or a PDF document describing software requirements and uses Natural Language Processing (NLP) techniques and a predefined ontology of software-related concepts to extract and structure the key information of the inputs. From this information, SONATA generates a conceptual model represented as a knowledge graph.
The resulting knowledge graph encodes the main entities, relationships, and concepts relevant to the requirements. This graph is then compared against existing knowledge graph derived from previously completed projects. By applying semantic matchmaking techniques, the system identifies which past projects share the most similar requirements and conceptual structures.

Automated Model Generation and Analysis:
SONATA takes a Java code repository as input and uses ANTLR4 to perform lexical and syntactic analysis. During this process, it extracts classes, methods, attributes, interfaces, relationships, and other key structural elements from the codebase. Guided by an ontology that defines and categorizes software concepts, SONATA then stores and organizes this extracted information in a knowledge graph. By doing so, the system not only captures the logical structure of the code, but also provides a more comprehensive understanding of its components and their interconnections.

Delta-Aware Text Generation

SONATA operates by comparing the input project's code structure against previously analysed repositories. After mapping classes, methods, and components to the knowledge graph, SONATA identifies those that closely resemble or match existing artifacts from past projects. When such similarities arise, it checks whether corresponding tests are available and still relevant to the new project's context. If tests already exist, SONATA recommends them as a starting point for validation. By doing so, it aims to accelerate the testing phase and enhance code quality. However, it carefully considers differences in dependencies, configurations, and usage patterns, since even identical classes may behave differently in another environment. The system provides recommended tests that can be adapted or refined as necessary, considering library versions, framework variations, and any unique domain-specific requirements. Over time, as developers review and confirm which

tests prove most reliable and applicable, SONATA refines its recommendations, improving the overall accuracy and usefulness of this feature.

Similarity Analysis Approaches and recommendations

One effective approach to similarity analysis involves representing code structures as knowledge graphs and then relying on ontology-driven semantic mappings to relate concepts and entities. This means that classes, methods, and interfaces are not only extracted, but also placed within a conceptual framework that expresses their roles, responsibilities, and relationships. By encoding software artifacts in this manner, SONATA can compare new projects against previously analyzed repositories more accurately, since it moves beyond simple text matching and focuses on shared logic, patterns, and functionalities. Models that compute similarity scores can incorporate syntactic, structural, and semantic dimensions, ensuring that recommendations reflect not only surface-level similarities, but also deeper conceptual alignments. To refine these approaches, it is advisable to continually update and maintain the ontology so that it remains current with evolving technologies and architectural trends. It is also important to validate similarity metrics against expert feedback and project outcomes, adjusting the criteria as needed to improve precision and recall. Over time, this iterative process of refinement and validation can yield a more stable and reliable similarity analysis methodology that enhances SONATA's capacity to recommend the most relevant artifacts, tests, and best practices for given project.

Monitoring and visualization

A dedicated Grafana dashboard provides a visual interface for monitoring the quality metrics of the Java repositories under analysis. Integrating data derived from SONATA's knowledge graph and complementary static analysis tools presents key indicators. This interface gives developers and project managers a clear, real-time overview of the software's health and maintainability. As new repositories are processed and their metrics are integrated into the dashboard, teams can quickly are processed and their metrics are integrated into the dashboard, teams can quickly identify trends, detect potential issues, and make informed decisions to improve the codebase over time.

c. Tools descriptions

- SONATA

Purpose:

SONATA analyzes software repositories and requirements to identify similar projects and recommend relevant test cases. It addresses challenges in test case creation and code reuse by creating a semantic understanding of projects through knowledge graphs. The platform reduces manual effort in identifying appropriate test cases while ensuring testing coverage and quality standards across projects.

Implementation:

- Utilizes ANTLR4 for Java code parsing and extraction of classes, methods, attributes, and interfaces
- Implements Natural Language Processing for requirements analysis and similarity detection
- Stores project structures and relationships in Neo4j knowledge graph
- Provides web interface built with Angular for project management and analysis
- Delivers REST API services through FastAPI for backend operations
- Monitors quality metrics through Vaadin dashboard integration

Input data:

- Java source code repositories

- Software requirements documents
- Natural language feature descriptions
- Repository URLs for direct analysis

Impact:

SONATA integrates into the development workflow by allowing developers to upload new projects or connect repositories for analysis. The system automatically processes the input, compares it against existing projects in the knowledge base, and provides test case recommendations. Quality metrics are continuously monitored and displayed through the dashboard interface, enabling teams to track improvements and identify areas needing attention.
Through its ontology-based semantic modeling and similarity analysis, SONATA fosters efficient code comprehension, better testing strategies, and informed reuse of existing solutions. By recommending tests already proven effective in similar contexts, it reduces overhead, shortens the validation phase, and improves overall code quality.

d. Visualization

SONATA's visualization solution focuses on presenting comprehensive project similarity and quality metrics through an interactive dashboard. The dashboard is implemented using the Vaadin framework, providing a seamless web-based interface that enables teams to make data-driven decisions about test case selection and project planning.

Key features include:

- Project Similarity Analysis
  - Displays matching requirements between current and historical projects
  - Shows artifact distribution across similar projects by type (project management, code, documentation)
  - Presents defect patterns and classifications from comparable projects
- Quality Metrics Overview
  - Visualizes defect distributions by category (functionality, security, performance)
  - Tracks project costs and resource allocation across similar implementations
  - Enables custom metric definition and monitoring for specific project needs
- Interactive Reporting
  - Provides detailed artifact analysis with filtering capabilities
  - Presents bug reports and correlation with project characteristics
  - Offers customizable views for different stakeholder needs

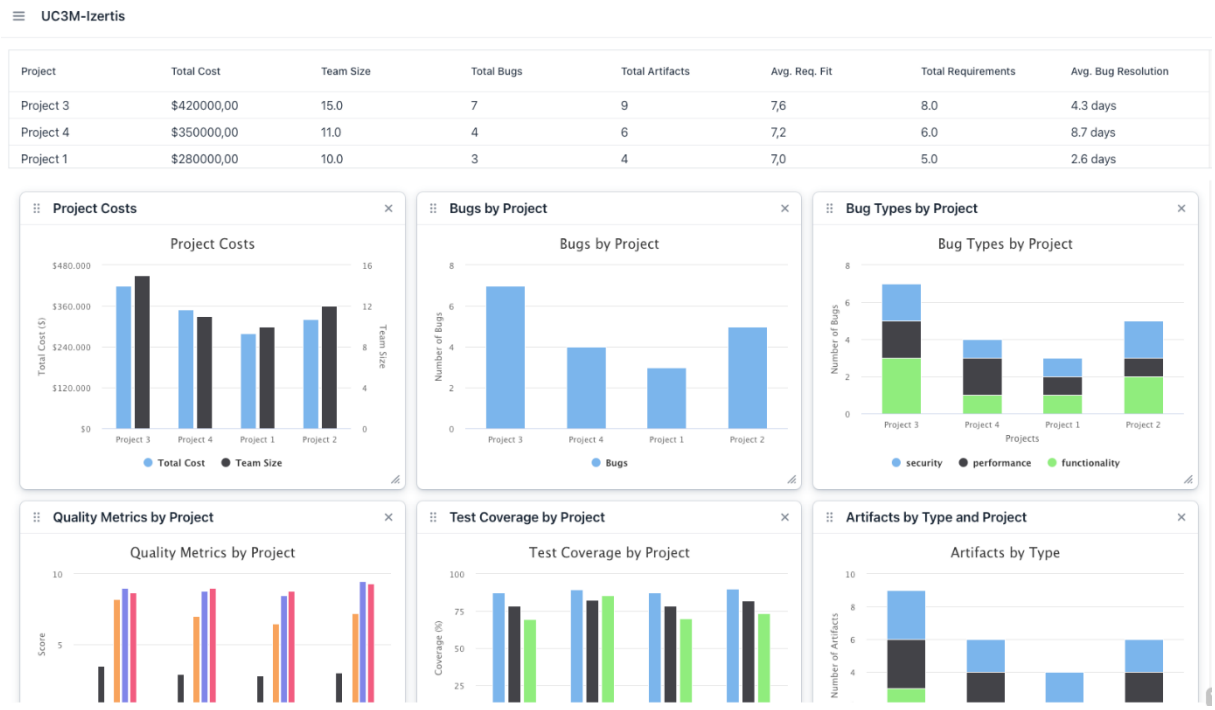


Figure 86: Dashboard key features

Table 17: Visualization requirements

Requ ID	Use Case Requirement	Requirement status
D5.UC9.1	The dashboard should show similar projects and the number of requirements that fit the customer's need	Finished
D5.UC9.2	The dashboard should show the number of artifacts developed by similar projects, grouped by type(project management, code artifacts, documentation)	Finished
D5.UC9.3	The dashboard should show similar projects with the number of artifacts developed and bugs reported	Finished
D5.UC9.4	The dashboard should display the number of defects from similar projects, grouped by type (Functionality errors, security bugs, performance defects)	Finished
D5.UC9.5	The dashboard should show the costs of similar projects	Finished
D5.UC9.6	The dashboard should display quality metrics for a similar project. The metrics can be defined by the user	Finished
D5.UC9.1	The dashboard should show similar projects and the number of requirements that fit the customer's needs	Finished
D5.UC9.2	The dashboard should show the number of artifacts developed by similar projects, grouped by type(project management, code artifacts, documentation)	Finished
D5.UC9.3	The dashboard should show similar projects with the number of artifacts developed and bugs reported	Finished

e. Evaluation Setup

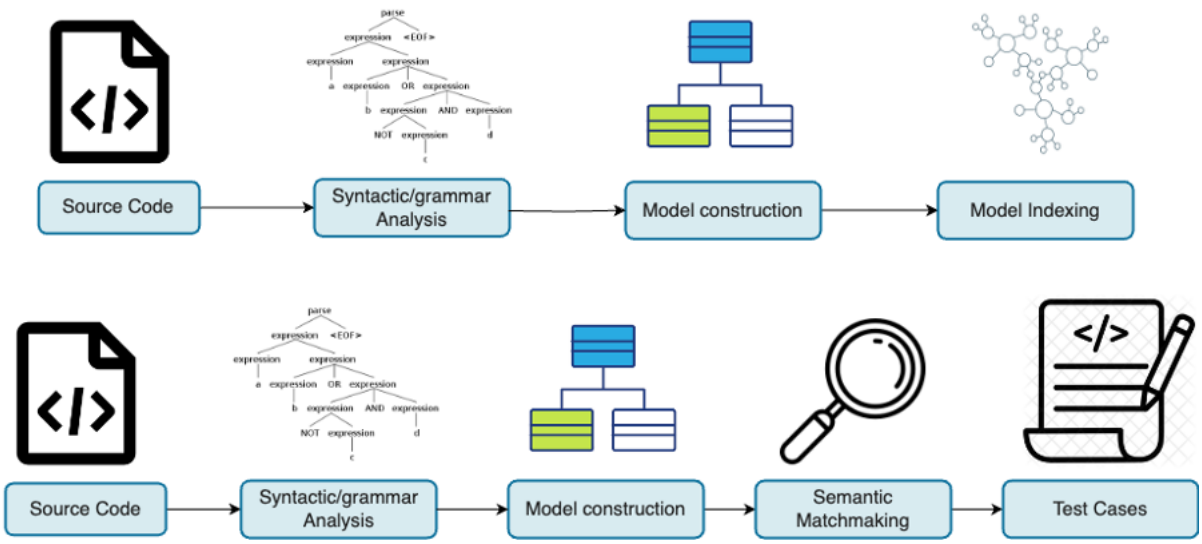


Figure 87: Evaluation setup

The evaluation of SONATA was conducted using a two-phase approach that tested both its core functionalities: model extraction and semantic matchmaking. The setup was designed around two key components, as illustrated in the diagrams:

Component 1 - Model Extraction and Indexing:

- Source code processing through syntactic/grammar analysis
- Model construction from the analyzed code
- Storage of constructed models in the knowledge graph
- Model indexing for future retrieval

Component 2 - Test Case Recommendation:

1. Source code analysis using the same syntactic approach
2. Model construction from analyzed code
3. Semantic matchmaking against stored models
4. Test case identification and recommendation

For the evaluation, we selected five Java projects with comprehensive requirements specifications and established conceptual models. This provided a baseline for comparing SONATA-generated models against real-world implementations. The evaluation was structured to address specific user stories and their associated KPIs:

User Story A Evaluation:

1. Focus on software artifact storage and indexing (UC9.FR1)
2. Assessment of feature identification from natural language requirements (UC9.FR2)
3. Comparison of SONATA-generated conceptual models against established baselines

User Story B Evaluation:

1. Testing of project suggestion accuracy (UC9.FR3)
2. Validation of similarity calculation between requirements and projects (UC9.FR4)
3. Assessment of conceptual model extraction accuracy (UC9.FR5)

User Story D Evaluation:

- Measurement of effort reduction in test environment setup (UC9.FR6)
- Analysis of test execution and reporting efficiency (UC9.FR7)
- Validation of test case suggestion accuracy (UC9.FR8)

- Implementation of visualization requirements (UC9.FR9.1-9.4)

The evaluation setup provided quantitative measures for each KPI, enabling objective assessment of SONATA's performance against target values.

f. Evaluation results

Table 18: KPIs Overview

Requirement	KPI Definition	KPI Base Values	KPI Target Values	KPI achieved Value
UC9.FR1	Percentage of software artefacts (including source code, use cases, requirements, etc.) from company projects successfully stored and indexed in the repository.	0%	90%	70%
UC9.FR2	Percentage of features from natural language requirements that are correctly identified and classified by the system.	0%	90%	80%
UC9.FR3	Percentage of development project suggestions accurately matching the given requirements in natural language.	0%	90%	80%
UC9.FR4	Automatic and correct calculation of the similarity degree between requirements and found projects.	0%	95%	90%
UC9.FR5	Accuracy of extracting conceptual models from source code	0%	90% accuracy in similarity calculation	90%
UC9.FR6	reduction in effort required to find possible test environments	100%	40%	40%
UC9.FR7	Reduction in effort required to execute and report the test results	100%	40%	40%
UC9.FR8	Percentage of accurately suggested test cases for reuse based on natural language requirements analysis and based on model similarities	0%	90%	90%
UC9.FR9.1-9.4	We need to visualize analysed data	0%	100%	100%

Further details about the KPI values are listed below:

- UC9.FR1: While targeting 90% of software artifacts to be successfully stored and indexed, SONATA achieved 70%. This gap likely reflects initial challenges in processing and indexing certain types of artifacts, particularly legacy code or non-standard documentation formats.
- UC9.FR2: The system achieved 80% accuracy in identifying and classifying features from natural language requirements against a target of 90%. This indicates the NLP components are functioning well but may need refinement for handling complex or ambiguous requirement descriptions.
- UC9.FR3: With an 80% achievement rate versus the 90% target for matching project suggestions to requirements, SONATA demonstrates strong but not yet optimal performance in project similarity detection. This suggests room for improvement in the semantic matching algorithms.

- UC9.FR4: The system reached 90% accuracy in calculating similarity degrees between requirements and projects, coming close to the 95% target. This high achievement validates the effectiveness of the semantic matchmaking approach.

g. Recommendation for industry adoption

The evaluation demonstrates that SONATA, powered by the SmartDelta Methodology, offers significant benefits for software development organizations seeking to optimize their testing processes and leverage existing code assets. The platform has shown promising results in automating test case identification and code reuse, particularly in enterprise-scale software development environments.
Challenges remain in several key areas. While SONATA demonstrates strong potential, the system currently indexes 70% of artifacts against the targeted 90%, particularly struggling with legacy systems and complex artifact structures. The natural language processing component shows promise but requires further refinement, currently achieving 80% accuracy when identifying features from ambiguous or complex requirements. Additionally, when processing large-scale repositories, performance optimizations are needed to maintain efficient operation at enterprise scale.

12. Use-Case 10 from Vaadin

a. Use-Case Description

Motivation

Vaadin is an open-source platform designed for building modern, collaborative web applications for Java backends. Its main products, Flow and Hilla, provide developers with frameworks for creating secure, scalable, and efficient applications. Vaadin Flow is a full-stack Java web framework that lets users build modern web applications without writing HTML or JavaScript. Hilla, on the other hand, is a full-stack framework combining Spring Boot, React, and UI components to offer both flexibility and productivity.

The applications developed using Vaadin’s frameworks are typically business-critical systems, catering to hundreds or thousands of users. These applications are characterized by their complexity and the need for reliable, high-performance solutions. Ensuring continuous improvement in quality, security, and performance is critical to maintain Vaadin's competitive edge in the market and to support its large community of developers.

Currently, Vaadin’s development processes involve several manual workflows that, while essential, are highly time-consuming. Issue triaging relies on manual categorization of reports, requiring developers to review and assign labels based on their understanding. Similarly, migrating applications between major versions demands significant manual analysis to identify and address API changes. Legacy code refactoring is often driven by expert insights rather than systematic tooling, which limits scalability. Additionally, performance and stability monitoring primarily depend on manual evaluations of test results, making it challenging to promptly identify and resolve regressions.

These processes, while effective to some extent, lack the streamlined automation necessary to scale with the increasing complexity of Vaadin’s frameworks and the growing needs of its developer community. Enhancing these workflows with automated solutions is essential to improve efficiency, consistency, and overall quality.

Challenges

Vaadin faces several challenges in its software development lifecycle, which the solutions developed within the scope of the SmartDelta project aim to address:

Issue Triaging and Classification

- With thousands of GitHub issues and growing, manual triaging has become resource-intensive and inconsistent.
- Automating the classification of issues by type, severity, and impact is essential to streamline backlog refinement, improve planning accuracy, and provide insights into which parts of the product are affected by the issues.

Major Versions Feature Parity

- Migrating from older versions, such as Vaadin 7 or 8, to the latest versions often introduces breaking changes and feature mismatches. While framework developers address these changes at an architectural level, it is the application developers who must handle the practical implications in their codebases.
- Automating the mapping and replacement of APIs and ensuring parity across versions can significantly reduce manual effort and improve upgrade efficiency for application developers.

Legacy Code Refactoring

- Vaadin's codebase has evolved over two decades, with varying adoption of modern practices and tools.
- Identifying outdated code and recommending refactoring actions are critical to sustain maintainability, performance, and security.

Performance and Stability Monitoring

- Refactoring and updates can impact the performance of applications built with Vaadin frameworks.
- Establishing automated monitoring tools to track performance trends and correlate them with code changes can help prevent regressions and ensure stability.

The solutions developed within the scope of the project aim to provide actionable insights, streamline workflows, and enhance the overall development process to address these challenges.

b. Link to SmartDelta Methodology

The SmartDelta Methodology provides a structured approach for managing software quality and adaptability in incremental software development. It consists of six stages, each addressing specific challenges in delta management. Vaadin's use case aligns with these stages, leveraging tools and practices to address challenges in software development processes (See Figure 88). While the tools applied in each stage may not always produce tangible outputs for the immediate next stage, they contribute significantly to the overarching goals and improve the overall quality of subsequent processes.

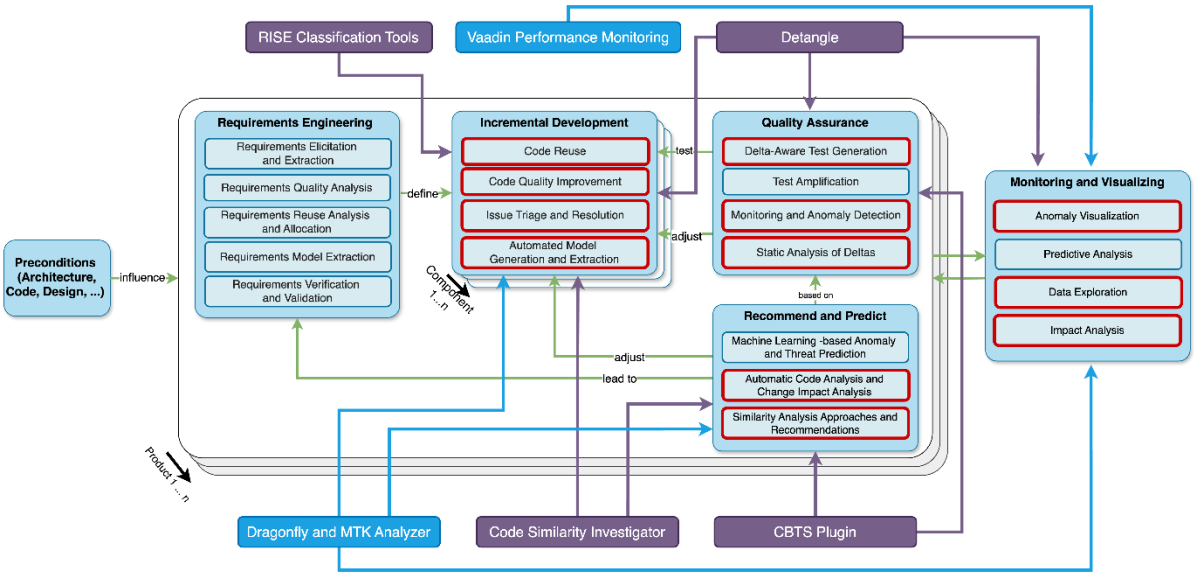


Figure 88: SmartDelta Methodology stages and elements used in the Vaadin use case.

Mapping Vaadin Use Case to SmartDelta Stages

Preconditions Consideration

Vaadin utilizes its established codebase, architecture, and user feedback to define baselines for product deltas. These preconditions ensure alignment with stakeholder needs and provide a strong foundation for planning and implementing incremental updates, without the need for external solutions such as those provided by the SmartDelta Methodology.

Requirements Engineering

Requirements for Vaadin's products, such as Flow and Flow Components, are documented as GitHub issues using predefined templates for consistency and completeness. The existing processes effectively support capturing and managing requirements without the need for external tools such as those provided by the SmartDelta Methodology.

Incremental Development

Vaadin's incremental development leverages modular architecture, reusable components, and iterative enhancements to ensure alignment with quality standards.

Classification tools developed by RISE contribute to labelling issues with type, severity and impact. These labels, assigned based on internal practices, enable prioritization by identifying high-impact and critical severity issues, which are given precedence. In addition, type classification provides insights about which aspect of the product is affected.

The Code Similarity Investigator from TWT supports identifying reusable code fragments and optimization opportunities, ensuring efficient updates. Moreover, for customer projects, tools such as Dragonfly and the MTK Analyzer are employed to support incremental development by automating legacy code migrations. Detangle complements these efforts by identifying areas of technical debt using meaningful KPIs, such as the Feature Debt Index and metrics reflecting collaboration challenges.

Quality Assurance (QA)

Incremental changes undergo a structured QA process, which includes unit tests, integration tests, visual tests, smoke tests, and platform-level validations. The Change-Based Test Selection Plugin (CBTS) contributes to this process by focusing testing efforts on affected components, reducing redundant testing. Complementary static analysis tools, such as SonarCloud for detecting code smells and vulnerabilities and code formatters for ensuring consistent coding style, ensure that code quality

standards are upheld throughout development. Additionally, Detangle is utilized within QA to analyze technical debt and offer metrics-driven insights that help maintain code quality and support continuous improvement.

Recommend and Predict

The Code Similarity Investigator contributes to identifying reusable components and improvement opportunities. This tool assists in locating previously optimized or fixed components that align with current development needs.

In customer projects, Dragonfly and the Modernization Toolkit Analyzer (MTK Analyzer) are leveraged to automate migrations, ensure feature parity across major versions, and support the integration of legacy features into new systems. Detangle complements this process by providing metrics-driven insights into architectural and technical debt hotspots, enabling informed decision-making on addressing improvements versus ongoing maintenance efforts. Furthermore, the CBTS plugin aids in predicting test impacts by selecting relevant test cases based on recent changes.

Monitoring and Visualizing

Vaadin's CI pipeline integrates dashboards that provide real-time insights into critical metrics, including test results and performance trends. These visualizations help monitor critical components, such as Flow and Grid, ensuring that bottlenecks and regressions are promptly identified and resolved. This approach supports continuous monitoring, maintaining the stability and reliability of the development process.

Conclusion

Vaadin's processes are aligned with the methodology. Tools developed for issue labelling, static analysis, code similarity investigation, and modernization support incremental development, quality assurance, recommendation, and monitoring. This alignment demonstrates a clear approach to managing updates and maintaining product quality.

c. Tools Descriptions

RISE Classification Tools/Solutions

Purpose

The RISE Issue Classification Tools are designed to automate issue triaging and categorization, addressing the challenges of manually managing a growing backlog of issues in the software development lifecycle. These tools aim to improve efficiency by reducing time spent on backlog refinement sessions and ensuring consistent issue categorization. Specifically, they help Vaadin by identifying the functionality areas, impact, and severity of issues, making planning and prioritization more efficient.

Implementation

The tools rely on a combination of fine-tuned language models and traditional machine learning approaches:

- **Issue Prediction**
  The tool is based on a fine-tuned large language model, i.e., BERT, for automated classification of issue reports with their category. Certain issue categories or labels are general, while others are specific to Vaadin.
- **Issue Severity**
  A fine-tuned generative large language model is used for recommending severity levels (blocker, major, minor) based on issue titles and descriptions. Preprocessing excludes code

snippets to reduce noise and improve accuracy, as including code was found to negatively impact performance.

- **Issue Impact**
  Logistic regression is employed to classify issues into high or low impact categories. While large language models were tested for this purpose, logistic regression provided better results due to the small, imbalanced dataset.

The tools are dockerized, and the models are accessed via API calls from a web-based application (see Figure 89).

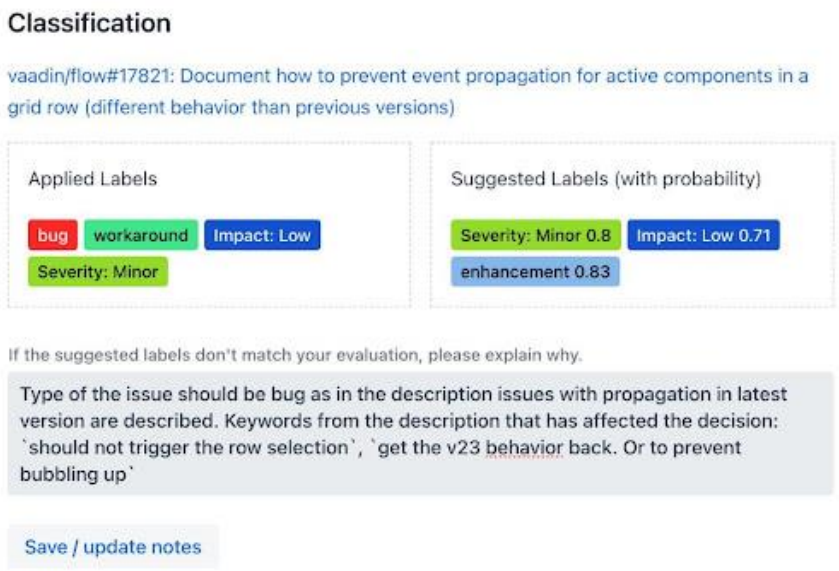


Figure 89: UI of the web-application used during evaluation.

Application in Use Case

In Vaadin's workflow, these tools are intended to be used during backlog refinement sessions by developers, product managers, and technical leads. The tools are integrated into a web application, which interacts with the dockerized models to fetch categorization suggestions for reported issues. Users can review the system-generated labels for functionality area, impact, and severity, and manually adjust them as needed.

Related KPIs

- UC10.FR_A1: Automatic suggestions for issue functionality area categorization based on the issue's minimal reproducible example.
- UC10.FR_A2: Automatic suggestions for issue impact categorization based on the issue's title and description.
- UC10.FR_A3: Automatic suggestions for issue severity categorization based on the issue's title and description.

Detangle by Cape of Good Code

Purpose

DETANGLE analyzes technical debt in software systems, targeting root causes such as architecture quality, code quality, process inefficiencies, and knowledge distribution. It aids refactoring by

identifying hotspots that hinder maintainability. In the Vaadin use case, it reduces manual effort in identifying improvement areas, enabling smoother feature implementation and lower bug density.

Implementation

The tool utilizes proprietary data mining and analysis techniques to compute metrics like System Effort, Maintenance Effort, and Primary Effort. By analyzing historical and current code changes at various levels (e.g., class, file, folder), DETANGLE predicts and quantifies the effort required to address technical debt. It also incorporates test coverage data to provide holistic technical debt analysis. These capabilities are delivered through meaningful visualizations such as graphs showing feature coupling, cohesion, and architectural extensibility trends.

Application in Use Case

In the Vaadin use case, DETANGLE simplifies managing the complexities of the Flow framework by analyzing feature and defect coupling across the codebase. It enables developers and technical leaders to identify tightly coupled modules and prioritize refactoring efforts. The tool provides visual feedback that highlights areas needing attention, with analyses performed as needed to generate updated insights.

Related KPIs

- UC10.FR_C1: Analysing, extracting and visualizing the nature of the code change (e.g., feature / refactoring / bug) and timeline of the change.

Code Similarity Investigator by TWT

Purpose

This tool automates the analysis of code to identify similarities and recommend reuse opportunities. The **Code Similarity Investigator (CSI)** focuses on identifying shared patterns and related structures within code sections. It addresses challenges in maintaining feature parity and refactoring legacy code efficiently, enhancing productivity and software quality in incremental development processes.

Implementation

- Uses Code Property Graphs (CPGs) to represent code sections.
- Applies graph-based algorithms to measure similarity.
- Identifies similar code blocks through structural analysis.

Input Data

- Source code from repositories or specific code segments.

Application in Use Case

The tool is integrated into Vaadin's software development workflow to streamline analysis and reuse processes. Provided in a dockerized format, it features a web interface for easy access by developers, enabling efficient examination of code for reuse opportunities or refactoring needs. This tool is intended to maintain high-quality standards and optimize the platform's evolution.

Related KPIs

- UC10.FR_B1: Analysing and listing references to code parts with similar functionality.
- **UC10.FR_C2:**

- **Original:**
  Automatic suggestions for design and code reuse based on natural language description of change requirements.
- **Revised:**
  Automatic suggestions for design and code reuse based on analysis of selected code segments, such as methods and classes.
- **Explanation of the Change:**
  The KPI has been updated to better reflect the underlying analysis process. Instead of relying on natural language descriptions, the revised requirement emphasizes that design and reuse suggestions are generated by analysing specific code segments. This approach leverages the identification of reusable patterns more precisely, thereby enhancing the overall accuracy and relevance of the suggestions provided.

Change-Based Test Selection (CBTS) Plugin by Fraunhofer FOKUS

Purpose

The CBTS Plugin is designed to optimize regression testing by analysing and classifying code modifications. Its primary goal is to minimize the effort required for regression testing by selecting the most relevant test cases. This is particularly beneficial for Vaadin products, where frequent updates, patches, and enhancements are common. The plugin ensures that these changes are thoroughly safeguarded against unintended consequences by identifying potential side effects.

Implementation

The CBTS Plugin uses **change impact analysis** techniques to assess modifications at a granular level (e.g., methods, classes, or components). It applies algorithms that map code changes to specific test cases, enabling the selection of those most likely to reveal issues caused by the changes (see Figure 90).

- **Data inputs:** The tool ingests code-level changes, such as commits, and analyzes the associated test cases from the repository.
- **Processing:** A traceability mapping between the codebase and test cases is generated, leveraging both static and dynamic analysis.

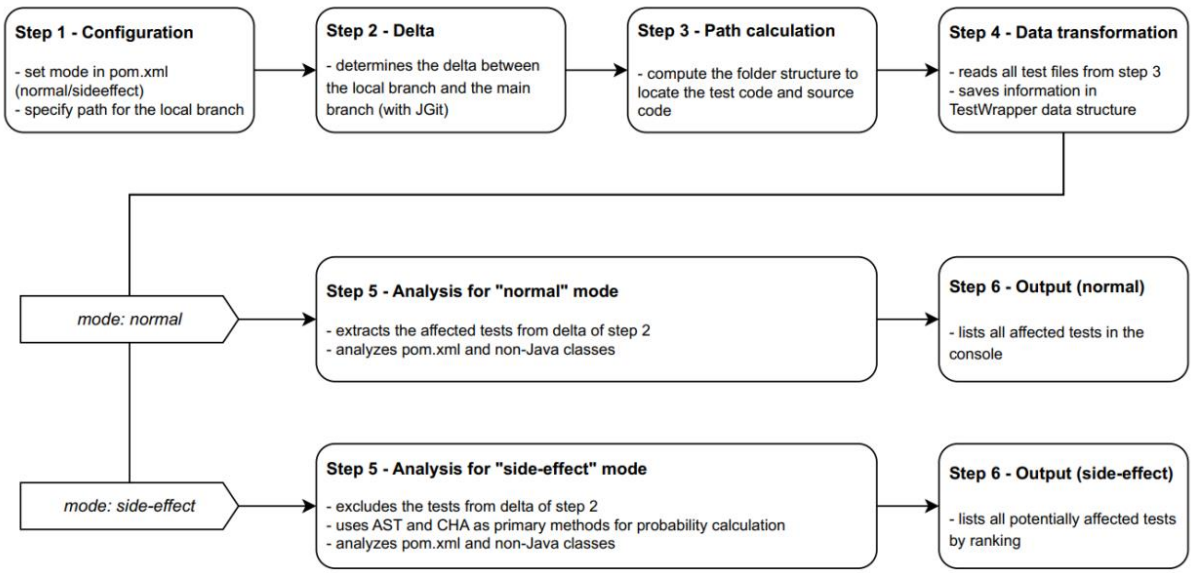


Figure 90: High-level workflow of CBTS

Application in Use Case

The CBTS Plugin is currently executed locally with specific configuration settings. This setup allows developers to evaluate its test selection capabilities in a controlled environment, although it is not fully integrated into a live CI/CD pipeline.

Related KPIs:

- UC10.FR_B3: Suggest tests for reuse based on feature description and/or code changes.

Dragonfly and Modernization Toolkit Analyzer (MTK Analyzer)

Purpose

Dragonfly and MTK Analyzer are designed to facilitate the migration of applications from older Vaadin versions (e.g., Vaadin 8) to the latest versions. These tools aim to automate the identification and transformation of API references that require updates, reducing the manual effort involved in migration and improving consistency. The main problems addressed include automating code updates, maintaining functionality parity across versions, and minimizing migration-related errors.

The tools were explicitly developed within Vaadin's use case to address feature parity across major releases, ensuring smoother transitions for users of Vaadin 7 or 8 to newer versions with web component-based front-end stacks. This is critical given the shift from GWT-based to web component-based architectures.

Implementation

Dragonfly and MTK Analyzer leverage the following technical components:

- **Parsing Framework:** The tools utilize Java Development Tooling (JDT) and Abstract Syntax Tree (AST) to analyze and transform Java source code.
- **Transformation Handlers:** Dragonfly incorporates handlers that parse compilation units (source files) into AST representations, apply predefined transformation rules, and save updated code. These rules are tailored to align older API references with the new Vaadin API.
- **Integration Modes:** Both tools can be used as plugins within Eclipse IDE or independently as Maven plugins.
- **Input and Output:**
    - **Input:** Java source code from applications developed using older framework versions, e.g. Vaadin 7 or 8.
    - **Output:** Updated source code compatible with the latest Vaadin versions or transformation coverage summaries.

Application in Use Case

In real-world scenarios, Dragonfly and MTK Analyzer are employed during migration projects where customer applications built on Vaadin 7 or 8 need to transition to newer versions.

- **Users:** The tools are used by developers and migration specialists within Vaadin and customer teams.
- **Workflow Integration:** They are integrated into the development workflow as plugins, enabling developers to analyse and transform project code directly within their development environment. The tools also output transformation summaries for review and validation.

- **Impact on Workflow:** These tools streamline the migration process by automating repetitive tasks, identifying migration gaps, and enabling high-level overviews of transformation coverage.

Related KPIs:

- UC10.FR_B2: Suggest API replacements based on the code/documentation analysis.

Internal Vaadin Performance Monitoring Solutions (Flow Fast Reload and Grid Performance Boost)

Purpose

The combined solution addresses key performance-related challenges in Vaadin's Flow framework and Grid component by targeting both our development workflow efficiency and the runtime performance experienced by end users. It aims to ensure optimal performance by:

- Automating the detection of performance regressions through integrated regression tests that continuously monitor key metrics, from reload times during development to the responsiveness of the Grid component at runtime.
- Improving development efficiency with faster reload times during code iterations.
- Enhancing the runtime performance of the Grid component, which is critical for applications handling large datasets.

The tools mitigate risks of performance degradation, reduce manual debugging efforts, and provide actionable insights for developers to maintain high performance standards in user applications.

Implementation

- **Flow Fast Reload**: This project targets faster reload times in Vaadin Flow applications, enabling developers to see code changes reflected in the browser almost instantly. It includes:
  - Automated tests that measure reload times for Flow apps of various sizes.
  - Tools to monitor and identify bottlenecks in reload processes for both small and large applications.
- **Grid Performance Boost**: Focused on enhancing the Grid component through:
  - Benchmarking tools that run daily, comparing the performance of Grid in different Vaadin platform versions.
  - Key metrics captured include render time, scroll-to-index time, expand time, and vertical scroll frame time.

Data inputs include continuous integration logs, benchmark results, and telemetry data collected during automated test runs. Dependencies include the CI infrastructure, and Java-based benchmarking scripts.

CI/CD Pipeline Enhancements: Continuous refinements have been implemented to streamline and automate the build and testing processes. These updates introduce additional automated checks and build configurations that reduce manual intervention and accelerate development cycles. The enhancements improve overall build reliability and facilitate proactive detection of performance issues, thereby ensuring a more efficient and stable deployment pipeline.

Application in Use Case

Vaadin's development teams use these tools to:

- Detect and prevent performance regressions during platform updates.
- Analyse trends in reload times and component performance metrics to identify areas for improvement.
- Test and optimize new features, such as the lazy columns functionality in Grid, before deployment.

Product managers and developers rely on the benchmark data and visualizations to prioritize fixes and enhancements. These tools are integrated into the CI/CD pipeline, ensuring consistent monitoring and proactive resolution of performance issues.

Related KPIs:

- UC10.FR_D1: Extract from CI logs the following data: test failures trends, performance trends.
- UC10.FR_D2: Analyse and list correlations between code changes and test failures/performance drops.
- UC10.FR_D3: Visualize performance trends over time.

d. Visualization

Dashboard Solution

The visualization solution for Vaadin's use case focuses on presenting statistical data derived from automated workflows and analysis tools. While the outputs of the SmartDelta tools are accessible directly within their respective environments, the statistical data they generate is utilized in the dashboard to provide actionable insights.

The visualization is implemented as a web application using the Vaadin platform (see Figure 91), incorporating the newly developed Vaadin Dashboard component. This approach ensures integration with existing tools while offering an intuitive and interactive interface for exploring key metrics.

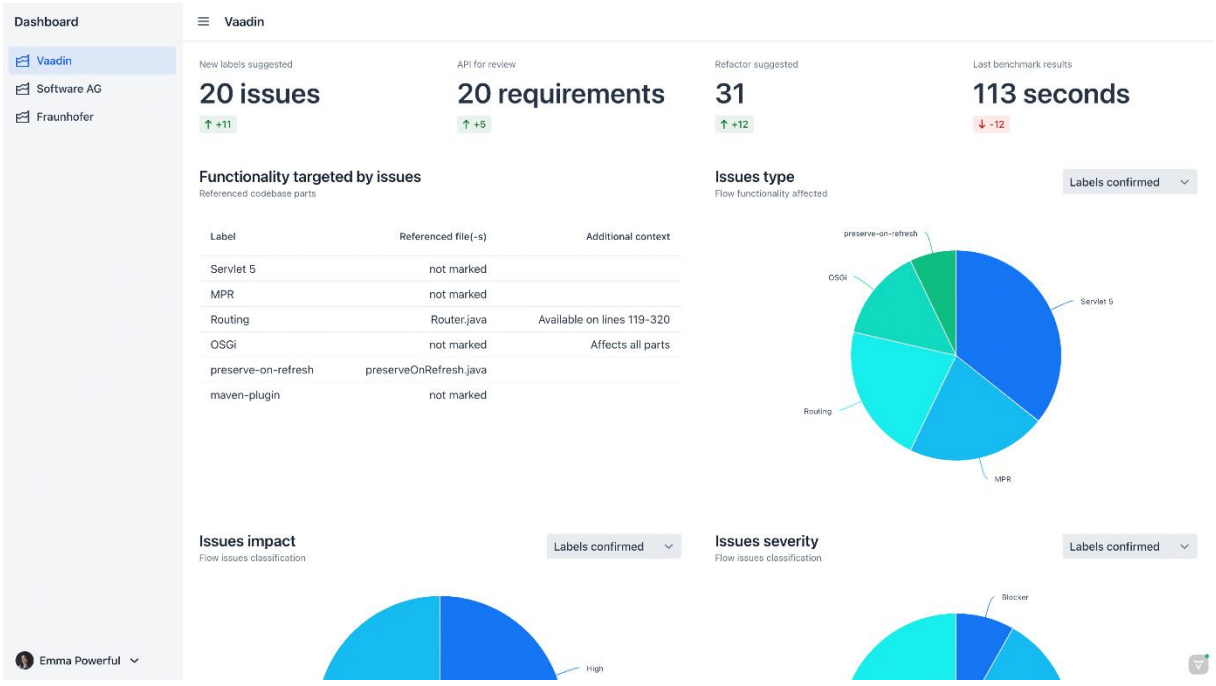


Figure 91: Dashboard web-application

Key features include:

- **Overview Metrics**: Displays summaries of issue classifications by type, severity, and impact, as well as progress on API migrations and refactoring activities.
- **Interactive Visualizations**: Pie charts, bar graphs, and spline charts enable users to monitor trends, identify bottlenecks, and make informed decisions.
- **Updates Summary**: Highlights changes and areas requiring attention, streamlining prioritization during development cycles.

Visualization Requirements

The dashboard aligns with UC10 requirements, designed to deliver meaningful insights to teams while supporting the integration of SmartDelta tools.

Table 19: Visualizations Requirements Overview

Req. ID	Title	Description	Status
D5.UC10.1	Management Dashboard - Overview	The management dashboard provides an overview of different metrics for a selected product. Detailed information could be opened via interaction with different components.	Finished
D5.UC10.2	Updates summary	Summary header containing information on the current state of the metrics and changes to those visualized with badges	Finished
D5.UC10.3	Issue classification by type	Summary of the labels available for classification by functionality affected with additional context of related files and notes. Pie chart visualizing the current distribution of the issues with possibility to switch between confirmed and suggested labels.	Finished
D5.UC10.4	Impact classification	Issue classification by impact visualized via pie chart.	Finished

Req. ID	Title	Description	Status
D5.UC10.5	Severity classification	Issue classification by severity visualized via pie chart.	Finished
D5.UC10.6	Functionality missmatch	Visualization of the comparison of the API between source and target versions: api aligned; api could be replaced; api missing; not confirmed; no action needed. Source and target version can be selected with the corresponding component.	Partial implementation
D5.UC10.7	Refactor suggestions	Visualization of the current state of suggested and proceeded refactorizations: refactored; under review; discarded.	Partial implementation
D5.UC10.8	Performance - Overview	Visualization of the performance trend of the framework.	Finished

By leveraging the Vaadin platform, including Vaadin Dashboard Components and Vaadin Charts, this visualization solution provides a user-friendly way to explore the data and insights generated by SmartDelta tools, enhancing the overall development and decision-making process.

13. Evaluation Setup

This section outlines the evaluation setups for each solution used in Vaadin's development workflow.

RISE Classification Tools/Solutions

Purpose
Automate issue categorization (functionality area, impact, severity) to reduce manual labelling effort and improve consistency.

Scope & KPIs

- Targets issues reported in the Vaadin components and Flow backlog.
- Related KPIs:
  - UC10.FR_A1 (functionality area),
  - UC10.FR_A2 (impact),
  - UC10.FR_A3 (severity).

Method

1. **Label Suggestions**: The tool provides label suggestions for new or unassigned issues.
2. **Expert Comparison**: Developers or product owners compare the automated labels with manually assigned labels.
3. **Data Collection**: Collect statistics on correct vs. incorrect classifications and log time saved vs. manual labelling.

Metrics

- **Accuracy**: Percentage of correct predictions (per label type).
- **Time Savings**: Reduction in median time spent on manual categorization.

Team

- Flow and Design System developers.

Detangle by Cape of Good Code

Purpose
Identify and visualize technical debt related to architecture, code quality, and historical change data.

Scope & KPIs

- Analyzes Vaadin's Flow codebase (including historical commits).
- Related KPI:
  - UC10.FR_C1 (nature/timeline of code changes).

Method

1. **Qualitative Analysis**: Tool outputs metrics like System Effort and Maintenance Effort.
2. **Review Sessions**: Developers and technical leads interpret DETANGLE's visualizations (e.g., hotspots, debt trends).
3. **Feedback Loop**: Observed findings guide refactoring priorities.

Metrics

- **Qualitative Feedback**: How helpful the insights/visualizations are in planning refactors.

Team

- Flow developers.

Code Similarity Investigator by TWT

Purpose
Automate detection of similar code fragments and suggest potential reuse opportunities.

Scope & KPIs

- Analyzes source code across Flow and adjacent modules.
- Related KPIs:
  - UC10.FR_B1 (listing code parts with similar functionality),
  - UC10.FR_C2 (suggest design/code reuse).

Method

1. **Code Pair Collection**: Select code snippets or classes known to be similar/dissimilar.
2. **Tool Execution**: Tools generate numeric similarity scores and simple descriptions (Low, Medium, High).
3. **Comparison**: Expert manual review checks alignment of tool's categorization with real functional overlap.
4. **Reuse Suggestions**: CSI indicates potential similar modules; experts validate relevance.

Metrics

- **Precision & Coverage**: For identifying truly similar code fragments.
- **Relevance of Reuse Suggestions**: How many suggestions developers find applicable.

Team

- Flow and Design System developers.

Change-Based Test Selection (CBTS) Plugin by Fraunhofer FOKUS

Purpose
Select the most relevant test classes/methods for code changes to reduce total regression testing effort.

Scope & KPIs

- Evaluates how well the tool addresses UC10.FR_B3 (suggest tests for reuse based on feature description/code changes).

Method

1. Test Identification: Define criteria for selecting tests covering both major and minor code changes.
2. Tool Execution: Run the CBTS Plugin to automatically select tests while recording performance metrics such as selection time and execution time reduction.
3. Evaluation: Experts assess the alignment between the tool's selections and anticipated coverage.

Metrics

- **Time Efficiency:** Combines the duration required for test selection with the subsequent savings in overall test execution time.
- **Test Reduction Efficiency:** The extent to which the tool reduces the overall regression test suite.
- **Defect Detection Capability:** The ability of the selected tests to capture critical defects compared to a full test run.

Team

- Fraunhofer FOKUS plugin developers.
- Flow developers.

Dragonfly and Modernization Toolkit Analyzer (MTK Analyzer)

Purpose
Streamline migration from older Vaadin versions (e.g., 7 or 8) to the latest architecture by automatically transforming outdated APIs.

Scope & KPIs

- Focus on real-world migration projects requiring API updates.
- Related KPI:
  - UC10.FR_B2 (suggest API replacements).

Method

1. **Project Selection**: Identify candidate applications built in Vaadin 7/8.
2. **Tool Execution**: Dragonfly or MTK Analyzer scans code for outdated references and applies transformations.
3. **Manual Review**: Developers confirm correctness of replacements and note any missed references.

Metrics

- **Coverage**: Fraction of outdated references automatically found vs. total references present.
- **Accuracy**: Fraction of correctly transformed references vs. total transformations applied.

Team

- Vaadin Modernization experts.
- Service Department developers involved in customer migration projects.

Internal Vaadin Performance Monitoring Solutions (Flow Fast Reload and Grid Performance Boost)

Purpose
Detect performance regressions and improve performance in Flow (fast reload) and Grid components (for example, rendering and scrolling).

Scope & KPIs

- Monitors continuous integration data for performance changes.
- Related KPIs:
  - UC10.FR_D1 (test failures and performance trends in CI logs),
  - UC10.FR_D2 (correlation between code changes and performance drops),
  - UC10.FR_D3 (visualize trends over time).

Method

1. **CI Integration**: Automated benchmarks run daily or per commit, tracking reloads times and grid metrics (for example, rendering and scrolling times).
2. **Trend Analysis**: Compare new runs to historical baselines; log significant regressions.
3. **Correlation**: When a drop is detected, the tool highlights recent commits for investigation.

Metrics

- **Reload Time Capture**: Assesses if the tool captures average or percentile reload times across different app sizes.

- **Grid Performance Capture**: Assesses if the tool captures key grid metrics, such as scroll latency and rendering time.
- **Performance Drop Linking**: Evaluates if the tool can link observed performance drops to corresponding code changes.
- **Trend Visualization**: Evaluates if the tool effectively visualizes performance trends over time.

Team

- Flow and Design System developers
- Product owners

14. Evaluation Results

Overview

Below is a table presenting the evaluation results against the predefined KPIs for the Vaadin use case within the project. The table compares the baseline and target values with the achieved results at the time of the final evaluation.

Table 20: KPIs Overview

Requirement ID	KPI Definition	Base Value	Target Value	Achieved Value
UC10.FR_A1	Reduction in time required for categorizing issues functionality areas, achieved using automatic suggestions based on the issue's minimal reproducible example	0%	70%	30%
UC10.FR_A2	Reduction in time required for assigning impact label to issues, achieved using automatic suggestions based on the issue's title and description.	0%	70%	30%
UC10.FR_A3	Reduction in time required for assigning severity label to issues, achieved using automatic suggestions based on the issue's title and description	0%	70%	30%
UC10.FR_B1	Analysing and listing references to code parts with similar functionality	0% (No mapping exists)	100% (List of code parts created)	100%
UC10.FR_B2	Suggest API replacements based on the code / documentation analysis	0% (Manual listing of the suggestions)	100% (Automatic suggestions of API replacements)	100%

Requirement ID	KPI Definition	Base Value	Target Value	Achieved Value
UC10.FR_B3	Suggest tests for reuse based on feature description and/or code changes	0% (Manual selection of tests)	100% (Automatic selection of existing tests based on feature description and/or code changes)	70%
UC10.FR_C1	Analysing, extracting and visualizing the nature of the code change (e.g., feature / refactoring / bug) and timeline of the change.	0% (Manual analysis using version control logs)	100% (Data is available and visualized)	100%
UC10.FR_C2	Automatic suggestions for design and code reuse based on analysis of selected code segments, such as methods and classes.	0% (Manual design and code re-use suggestions)	100% (Automatic design and code re-use suggestions)	100%
UC10.FR_D1	Extract from CI logs the following data: test failures trends, performance trends	0% (Unstructured log data)	100% (Analysable data structure)	100%
UC10.FR_D2	Analysing and listing correlations between code changes and tests failures / performance drops	0% (Unstructured log data; no correlation listing)	100% (Correlation lists are created; analysable data structure)	100%
UC10.FR_D3	Visualization of the performance trends over the time	0% (No data available)	100% (Data is available and visualized)	100%

RISE Classification Tools/Solutions

The RISE classification tools were evaluated in two rounds to assess their ability to automate issue categorization for functionality area (UC10.FR_A1), impact (UC10.FR_A2), and severity (UC10.FR_A3). The combined findings from both rounds are presented below.

First Evaluation Round

Experimental Evaluation (RISE Team)

- **Issue Prediction (Tool 1):** On a dataset of 4344 issues (15% for evaluation), the model achieved **75% accuracy** for classifying issue type (e.g., bugs, enhancements).
- **Issue Severity (Tool 2):** On 3472 issues (15% evaluation split), the tool reached **67% accuracy** for recommending severity (blocker, major, minor).
- **Issue Impact (Tool 3):** With a smaller dataset of 1753 entries, the tool achieved **62% accuracy** in classifying impact (high vs. low).

Use Case-Oriented Evaluation (Vaadin)

- **Data:** 185 new issues from Flow, Flow-Components, and Web-Components repositories.
- **Method:** Manual vs. automated label comparison.
- **Findings:**
  - **Flow Repository:** 81 issues, 18 misclassifications.
  - **Flow-Components Repository:** 59 issues, 7 misclassifications.
  - **Web-Components Repository:** 45 issues, 3 misclassifications.
  - Most discrepancies involved nuanced severity markers (e.g., memory leaks for blocker). Automated labels reduced discussion time, though manual intervention was still frequently required.

Log Time Summary

During the first evaluation round, the primary focus was on tool accuracy. The average time to classify an issue showed a 16% reduction, although extensive time evaluation was not the main focus.

Second Evaluation Round

A combined dataset of 151 issues was split so that 70% (105 issues) formed the test set. A few-shot prompting approach with a generative LLM was used for severity and impact predictions, while functionality area suggestions continued to be sourced from the previous model.

- **Severity Prediction:** Accuracy of **65%** for classifying issues as major, blocker, or minor.
- **Impact Prediction:** Accuracy of **73%** in distinguishing high vs. low impact.

Log Time Summary

During the second evaluation, detailed log times were captured to assess the impact of label suggestions on the classification process. Without suggestions, the average time to classify an issue was 98.7 seconds, while with suggestions it dropped to 73.4 seconds, representing an average reduction of approximately 25.65%. Similarly, the median classification time decreased from 90 seconds to 63 seconds (a 30% reduction).

These results demonstrate that label suggestions contribute to a more efficient classification process. The median time metric was selected for comparison against our target KPI due to the presence of outliers in the collected classification times.

Accuracy Summary

Across both evaluation rounds, the RISE classification tools provided automatic suggestions with the following accuracy:

- **Functionality Area (UC10.FR_A1):** Issue type classification averaged around **75% accuracy**
- **Impact (UC10.FR_A2):** Logistic regression and LLM-based evaluations showed **62%–73% accuracy**.
- **Severity (UC10.FR_A3):** The models achieved **65%–67% accuracy**

KPI Conclusion

The evaluation outcomes reveal a dual narrative. On one hand, the reduction in classification time, evidenced by the median log time improvements, indicates that the automated label suggestions are beneficial in streamlining the manual process. On the other hand, in-depth analysis has uncovered important insights into Vaadin's classification practices:

- Vaadin was provided with an automated similarity analysis results and identified 40 cases where similar issues were classified differently. Manual analysis concluded that these variations were largely due to missing context or varying human interpretations.
- A reevaluation of a subset (~105 issues) by a different group of experts resulted in a 7.6% discrepancy in labels, where experts were uncertain about assigning labels, underscoring the inherent challenges and subjective nuances in classification.

These findings indicate that, while automation effectively reduces manual effort, further refinement of the expert review process is needed to achieve the KPI target of a 70% reduction in manual classification time. Adjusting the training dataset will be a time-intensive process and should be managed outside of the project's current scope.

Detangle by Cape of Good Code

Using DETANGLE on the Vaadin Flow codebase yielded several key findings regarding technical debt, architecture quality, and code maintainability. The tool computed metrics such as **System Effort**, **Maintenance Effort**, and **Primary Effort (see Figure 92)** to capture and categorize the nature of code changes. Specifically, DETANGLE's analyses highlighted:

- **Nature and Timeline of Code Changes**
  DETANGLE successfully differentiated between feature-driven, bug-fixing, and technical-improvement (refactoring) changes. By visualizing these categories and their corresponding timelines, the tool provided clarity on when and where development effort was concentrated.
- **Hotspot Identification**
  Two critical "frontend" folders: **flow-server/src/main/java/com/vaadin/flow/server/frontend** and its corresponding test folder, exhibited high coupling and significant effort spent on both new features and bug fixes (see Figure 93). These folders together represent nearly 10% of the total codebase and show a notable overlap of maintenance and feature effort, indicating tightly coupled modules that can lead to unintended side effects.
- **Architecture Extensibility and Maintainability**
  Metrics such as **Primary Debt Index (PDI)**, **Defect Density**, and **Defect Impact** revealed that the identified hotspot folders have reduced architectural extensibility. They also experience recurring defect patterns, suggesting that new features implemented within these folders risk introducing further bugs and rework.
- **Refactoring Recommendations**
  By drilling down to the file level, DETANGLE provided insights for splitting large files, reorganizing code sections, and reducing inter-file dependencies. This granular view enabled focused discussions on how best to lower technical debt in the identified hotspots.

Figure 92: Detangle metrics

File Modules	LOC	Effort	Impact	Tech Debt Arch.	Tech Debt Code
flow-server/src/test/java/com/vaadin/flow/dom/ElementTest.java	2.5 K	66.00	HIGH	LOW	LOW
flow-server/src/test/java/com/vaadin/flow/server/frontend/AbstractNodeUpdatePackagesTest.java	897.0	433.00	MEDIUM	HIGH	LOW
flow-server/src/test/java/com/vaadin/flow/server/frontend/TaskRunPnpmInstallTest.java	729.0	373.00	MEDIUM	LOW	LOW
flow-server/src/test/java/com/vaadin/flow/server/frontend/AbstractUpdateImportsTest.java	572.0	178.00	HIGH	LOW	LOW
flow-server/src/main/java/com/vaadin/flow/server/frontend/TaskUpdatePackages.java	508.0	390.00	MEDIUM	HIGH	LOW
flow-server/src/test/java/com/vaadin/flow/server/frontend/FrontendUtilsTest.java	498.0	161.00	HIGH	LOW	LOW
flow-server/src/test/java/com/vaadin/flow/server/frontend/TaskUpdateThemeImportTest.java	434.0	76.00	MEDIUM	LOW	LOW
flow-server/src/main/java/com/vaadin/flow/server/communication/IndexHtmlRequestHandler.java	415.0	285.00	MEDIUM	LOW	LOW
vaadin-spring/src/test/java/com/vaadin/flow/spring/security/RequestUtilTest.java	402.0	150.00	HIGH	HIGH	LOW
flow-server/src/test/java/com/vaadin/flow/server/PwaRegistryTest.java	343.0	334.00	HIGH	LOW	LOW

Figure 93: Technical Debt Hotspots

Conclusion

In conclusion, the DETANGLE tool has successfully met the established UC10_FR_C1. It has provided valuable insights into potential areas for refactoring, thereby making a significant contribution to the overarching goal of Vaadin's user story. The tool's ability to identify hotspots and suggest improvements is commendable and aligns well with the objectives of enhancing code quality and design efficiency.

Code Similarity Investigator by TWT

Below are the consolidated findings for the CSI, derived from both automated analysis scores and detailed manual review of representative code pair comparisons provided by Vaadin. In this evaluation, similarity levels were categorized into four classes: None, Low, Medium, and High, and a numerical mapping was applied (None = 1, Low = 2, Medium = 3, High = 4) to quantify the difference between CSI's automated assessments and expert manual evaluations.

1. **Overall Accuracy and Matching**
    a. **High Alignment with Manual Review**: In most cases, the automated similarity scores closely matched expert assessments. Code pairs with nearly identical or near-identical logic (e.g., ViewAccessChecker vs. NavigationAccessControl getAccessDeniedException) were consistently classified as High by both CSI and manual review.
    b. **Moderate Cases**: Several code pairs, such as BuildFrontendMojo vs. VaadinBuildFrontendTask, were categorized as Medium by CSI. Despite differences in context, naming, or underlying build systems (Maven vs. Gradle), expert analysis confirmed the core similarity in functionality.
2. **Discrepancies**
    a. **Naming and Context Variation:** A few code pairs (e.g., BuildFrontendMojo vs. BuildDevBundleMojo) were flagged with discrepancies because CSI detected structural parallels but did not fully capture contextual nuances or divergent project goals.
    b. **Unavailable or Non-Comparable Methods:** In certain instances (e.g., ListDataProvider.fetch vs. CustomInMemoryDataProvider.fetch), no direct mapping could be established due to missing or dispersed method definitions across files.

Quantitative Evaluation Extract

A detailed quantitative assessment of the 29 code pair comparisons yielded the following results based on the numerical mapping:

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

Overall, 93% of the evaluated cases exhibited either exact matches or close agreement between CSI's automated assessments and expert manual evaluations.

Fulfilment of KPI UC10.FR_B1
CSI successfully identified and listed code segments with similar functionality, thereby meeting the KPI requirement. Although some discrepancies arose due to contextual or naming variations, the tool's performance in matching expert assessments underscores its effectiveness in supporting code reuse and refactoring efforts.

Fulfilment of KPI UC10.FR_C2
To assess automatic suggestions for design and code reuse, CSI was validated using the same set of code pairs from the manual similarity analysis:

- Reuse Suggestions: The tool provided suggestions that aligned with manual findings in all tested cases. In several instances, CSI recommended additional classes that were not initially part of the manual comparison but were later confirmed to be relevant after expert review.
- Performance Considerations: In most evaluation scenarios, the tool's analysis took over one hour, indicating potential performance bottlenecks for larger codebases or more complex queries.

Despite these performance challenges, CSI meets the KPI requirement for automatic design and code reuse suggestions. By providing relevant suggestions, the tool fulfills the core objective of UC10.FR_C2, demonstrating its potential to streamline design and code reuse in the development process.

Conclusion
The evaluation confirms that the Code Similarity Investigator aligns closely with expert evaluations, with 93% of cases showing exact or near-exact matches for similarity analysis. This strong alignment validates CSI's capability to reliably detect and suggest reusable code segments (KPI UC10.FR_B1). Additionally, its automated recommendations (KPI UC10.FR_C2) have proven useful, despite occasional performance constraints. Overall, CSI emerges as a promising solution for automating code similarity analysis and reuse suggestions in complex software systems.

CBTS Plugin (Change-Based Test Selection Tool) by Fraunhofer FOKUS

Qualitative Observations

During the evaluation, the CBTS Plugin was tested on Vaadin Flow modules with code modifications. The following observations emerged from developer feedback:

- **Parser Configuration:** An upgrade to a newer JavaParser library (3.26.3) and explicit configuration for Java 17 features were necessary to avoid parse errors in newer Flow components.
- **Fallback for Default Packages:** Tests without explicit package declarations caused exceptions, indicating a need for more robust handling.
- **Test Selection Accuracy:** The plugin reliably identified tests associated with significant modifications, such as changes in class inheritance, interface declarations, import statements, and dependency configurations. However, it did not suggest tests for subtle changes within existing methods, which are the most frequent modifications in daily development.
- **Developer Integration:** Developers acknowledged potential time savings from the automated listing of relevant tests. Nonetheless, as a TRL: 4 prototype, the tool's maturity limits its readiness for full integration into live pipelines.

Quantitative Observations

Evaluations across several Vaadin modules introduced a spectrum of modifications. Significant modifications, such as import statement changes or POM adjustments were effectively processed by the tool. In contrast, minimal modifications within existing methods (for example, minor bug fixes) were not captured by the tool, despite their frequency in day-to-day code maintenance.

Table 21: Evaluation of CBTS with Respect Test Components

Test componentent	amount of tests	type of side effects	execution time	number of executions
test-pwa-disabled-offline	3	Changes in import Dependencies from external libaries Changes to Interfaces POM Dependencies	1.4s	9
flow-polymer2lit	23	Changes in import Dependencies from external libaries Whitespace and minor changes POM Dependencies	3.2s	12
test-misc	10	Changes in import Dependencies from external libaries Java Changes to Class Inheritance POM Dependencies	2.9s	12
test-dev-mode	27	Changes in import Dependencies from external libaries POM Dependencies	3.1s	11

Further evaluation of the CBTS Plugin yielded the following performance metrics for the Vaadin Flow Framework (which comprises 18 core modules, 34 test modules, a suite of over 6680 tests with a total execution time of approximately 1.5 hours):

- **Regression Test Selection Time:** Approximately 30.23 seconds.
- **Number of Selected Regression Tests:**
  - *pwa-disabled-offline:* 3 out of 35 tests selected (~92% reduction)
  - *Flow-polymer2lit:* 23 out of 75 tests selected (~70% reduction)
  - *Flowmisc:* 10 out of 152 tests selected (~94% reduction)
  - *Flow-dev:* 27 out of 265 tests selected (~90% reduction)
- **Reduction in Execution Time:**
  - *pwa-disabled-offline:* ~95% reduction
  - *Flow-polymer2lit:* ~80% reduction

    - *Flowmisc:* ~94% reduction
    - *Flow-dev:* ~96% reduction
- Findings
    - None of the selected regression tests found a side-effect-caused defect.
    - The entire test suite detected only a few side-effect-caused defects; most defects found by the entire regression test suite were caused by direct changes applied to the code, not because of side-effects

Defect Detection Capability

While the tool meets the KPI for “Automatic selection of existing tests based on feature description and/or code changes” in cases of significant modifications, its overall defect detection capability is limited. Specifically, although tests for major changes (e.g., altered imports, dependency changes) are reliably suggested, the tool fails to capture tests relevant to minimal changes within existing methods. This limitation, affecting the majority of daily updates, results in an overall achieved defect detection capability of approximately 70% relative to the target value.

Conclusion

The final evaluation indicates that the CBTS Plugin demonstrates acceptable performance for identifying tests related to significant code modifications. However, its maturity currently limits comprehensive coverage, especially regarding the subtle changes within existing methods that dominate daily work. Despite these limitations, the tool provides valuable insights for regression test selection when run locally with appropriate configurations.

Dragonfly and Modernization Toolkit Analyzer (MTK Analyzer)

Below are the evaluation results obtained from analyzing three different service customer projects for migration from Vaadin 8 to the latest Vaadin versions. The statistics focus on method and constructor references, highlighting coverage rates (i.e., the proportion of references for which an automatic transformation rule was successfully applied).

```
Vaadin Minifinder version 202309011107
Run 2023-09-23
Location: [elided]
Found: 5 336 Vaadin 8 references and 0 Vaadin 7 references
.. in: 101 615 lines of Java
.. in: 466 Java files
.. in: 12 Java projects

Breakdown of references by syntactic construct:
Syntactic construct                          Coverage #         Total #      Coverage %
-------------------------------------------------------------------------------------
Method invocation                                2 411           3 536             68%
Constructor invocation                             237             296             80%
=====================================================================================

Top methods:
Declaring class                 Method                    Coverage #         Total #      Coverage %
-------------------------------------------------------------------------------------------------
AbstractComponent               setEnabled                       127             127            100%
                                setSizeFull                       73              73            100%
                                addStyleName                      72              72            100%
                                setWidth                          71              71            100%
                                setStyleName                      70              70            100%
                                others (23)                      308             349             88%
                               -------------------------------------------------------------------
                                total                            721             762             94%

AbstractOrderedLayout           addComponent                     154             154            100%
                                setComponentAlignmen              56              56            100%
                                setMargin                         54              54            100%
                                setExpandRatio                    52              52            100%
                                setSpacing                        34              34            100%
                                others (11)                      112             112            100%
                               -------------------------------------------------------------------
                                total                            462             462            100%

Grid                            getSelectedItems                  55              55            100%
                                getDataProvider                   33              33            100%
                                setSelectionMode                  13              13            100%
                                getColumn                         13              13            100%
                                deselectAll                       11              11            100%
                                others (34)                       46              83             55%
                               -------------------------------------------------------------------
                                total                            171             208             82%

MenuItem                        setVisible                         0              97              0%
                                setEnabled                         0              51              0%
                                addItem                            0               9              0%
                                isVisible                          0               5              0%
                                addItem                            0               3              0%
                                others (1)                         0               1              0%
                               -------------------------------------------------------------------
                                total                              0             166              0%

ComboBox                        setItems                          26              26            100%
                                setEmptySelectionAll              24              24            100%
                                setItemCaptionGenera              18              18            100%
                                setPageLength                     15              15            100%
                                addValueChangeListen               8               8            100%
                                others (6)                        13              16             81%
                               -------------------------------------------------------------------
                                total                            104             107             97%

others (114)                    total                            953           1 831             52%
```

Figure 94: MiniFinder report of project evaluation 2023-09-23

1. Project Evaluated on 2023-09-23

- **Scope**: 466 Java files, **5,336** Vaadin 8 references identified
- **Method Invocations**: 3,536 references; **68%** coverage
- **Constructor Invocations**: 296 references; **80%** coverage
- **Key Challenge**: Missing rules for `MenuItem` methods
- **Observation**: Overall solid coverage, with specific improvement areas around unhandled `MenuItem` transformations

```
Vaadin Minifinder version 202309011107
Run 2023-10-05
Location: [elided]
Found: 22 793 Vaadin 8 references and 0 Vaadin 7 references
.. in: 58 640 lines of Java
.. in: 203 Java files
.. in: 1 Java projects

Breakdown of references by syntactic construct:
Syntactic construct                          Coverage #         Total #      Coverage %
-------------------------------------------------------------------------------------
Method invocation                              11 106           13 330             83%
Constructor invocation                          2 457            3 165             77%
=====================================================================================

Top methods:
Declaring class                  Method                    Coverage #         Total #      Coverage %
-----------------------------------------------------------------------------------------------------
AbstractComponent                setSizeFull                      797             797            100%
                                 setWidth                         361             361            100%
                                 addStyleName                     268             268            100%
                                 setEnabled                       120             120            100%
                                 setVisible                        85              85            100%
                                 others (18)                      363             370             98%
                                ---------------------------------------------------------------------
                                 total                          1 994           2 001             99%

AbstractOrderedLayout            addComponent                     973             973            100%
                                 setComponentAlignmen             541             541            100%
                                 setExpandRatio                   313             313            100%
                                 setSpacing                        84              84            100%
                                 setMargin                         42              42            100%
                                 others (5)                        12              12            100%
                                ---------------------------------------------------------------------
                                 total                          1 965           1 965            100%

Grid                             addColumn                        676             676            100%
                                 addColumn                        111             111            100%
                                 addComponentColumn                60              60            100%
                                 getSelectionModel                 57              57            100%
                                 getEditor                         55              55            100%
                                 others (18)                      190             259             73%
                                ---------------------------------------------------------------------
                                 total                          1 149           1 218             94%

Column                           setCaption                       845             845            100%
                                 setWidth                         165             165            100%
                                 setRenderer                       55              55            100%
                                 setComparator                     45              45            100%
                                 setExpandRatio                    33              33            100%
                                 others (7)                        63              64             98%
                                ---------------------------------------------------------------------
                                 total                          1 206           1 207             99%

ComboBox                         setEmptySelectionAll             166             166            100%
                                 addValueChangeListen             163             163            100%
                                 setItems                         150             150            100%
                                 setItemCaptionGenera              48              48            100%
                                 setPageLength                     40              40            100%
                                 others (6)                        86             125             68%
                                ---------------------------------------------------------------------
                                 total                            653             692             94%

others (90)                      total                          4 139           6 247             66%
```

Figure 95: MiniFinder report of project evaluation 2023-10-05

2. Project Evaluated on 2023-10-05

- **Scope**: 203 Java files, **22,793** Vaadin 8 references identified
- **Method Invocations**: 13,330 references; **83%** coverage
- **Constructor Invocations**: 3,165 references; **77%** coverage
- **Notable Strength**: High coverage (94%) for `Grid` class references
- **Observation**: Marked improvement over the previous project. Lower-coverage classes indicated remaining rules to be refined.

```
Vaadin Minifinder version 1.4.4.202310190716
Run 2023-10-26
Location: [elided]
Found: 20 636 Vaadin 8 references and 0 Vaadin 7 references
.. in: 5 674 203 lines of Java
.. in: 40 366 Java files
.. in: 532 Java projects

Breakdown of references by syntactic construct:
Syntactic construct                                                    Coverage #    Total # Coverage %
-------------------------------------------------------------------------------------------------------
Method invocation                                                          6 775      10 489        64%
Constructor invocation                                                     1 334       1 514        88%
=======================================================================================================

Top methods:
Declaring class                  Method                                Coverage #    Total # Coverage %
-------------------------------------------------------------------------------------------------------
AbstractComponent                setWidth                                     234         234       100%
                                 setSizeFull                                  209         209       100%
                                 addStyleName                                 185         185       100%
                                 setId                                        171         171       100%
                                 setVisible                                   163         163       100%
                                 others (33)                                  668         833        80%
                                -------------------------------------------------------------------------
                                 total                                      1 630       1 795        90%

AbstractOrderedLayout            addComponent                                 565         565       100%
                                 setMargin                                    134         134       100%
                                 setSpacing                                   121         121       100%
                                 setExpandRatio                                86          86       100%
                                 setComponentAlignment                         51          51       100%
                                 others (7)                                    85          85       100%
                                -------------------------------------------------------------------------
                                 total                                      1 042       1 042       100%

Grid                             addColumn                                    195         196        99%
                                 getSelectedItems                              31          31       100%
                                 select                                        25          25       100%
                                 getDataProvider                               25          25       100%
                                 getColumns                                    15          15       100%
                                 others (29)                                   76         112        67%
                                -------------------------------------------------------------------------
                                 total                                        367         404        90%

Column                           setCaption                                   197         197       100%
                                 setHidable                                    47          47       100%
                                 setRenderer                                   33          33       100%
                                 setId                                         30          30       100%
                                 setWidth                                      19          19       100%
                                 others (12)                                   53          79        67%
                                -------------------------------------------------------------------------
                                 total                                        379         405        93%

UI                               getCurrent                                   134         134       100%
                                 addWindow                                     86          86       100%
                                 access                                         0          76         0%
                                 getPage                                       21          21       100%
                                 getSession                                    12          12       100%
                                 others (11)                                   24          40        60%
                                -------------------------------------------------------------------------
                                 total                                        277         369        75%

others (237)                     total                                      3 080       6 474        47%
```

Figure 96: MiniFinder report of project evaluation 2023-10-26

3. Project Evaluated on 2023-10-26

- **Scope**: 40,366 Java files, **20,636** Vaadin 8 references identified
- **Method Invocations**: 10,489 references; **64%** coverage
- **Constructor Invocations**: 1,514 references; **88%** coverage
- **Key Challenge**: High complexity of Vaadin API usage in certain classes
- **Observation**: While constructor references maintained strong coverage, the diversity of method usage affected the overall method-invocation coverage.

Conclusion

Across all three projects, the tools demonstrated a strong ability to **suggest and apply API replacements** in alignment with the **UC10.FR_B2** requirement of providing automated API recommendations based on code analysis. Constructor invocations were generally transformed with high coverage, reflecting robust rules for new-instance migrations. Method-invocation coverage varied according to project complexity but remained sufficiently high to significantly reduce manual migration

efforts. Overall, these results confirm that the tools effectively fulfil the KPI of **suggesting API replacements** and substantially support automated modernization from Vaadin 7 or 8 to the current architecture.

Internal Vaadin Performance Monitoring Solutions (Flow Fast Reload and Grid Performance Boost)

The internal Vaadin performance monitoring solutions for Flow Fast Reload and Grid Performance Boost have successfully transitioned from fragmented, unstructured log data to fully analyzable and visualized data (see Figure 29 and Figure 30). Below is a summary of how the solutions meet the stated KPIs and key findings from the monitoring setup:

1. **UC10.FR_D1:** Extract from CI logs the following data: test failures trends, performance trends.
    a. **Before**: Logs were captured but lacked a coherent format for systematic analysis.
    b. **After**: Performance metrics (e.g., refresh times, Grid rendering speeds) are now automatically collected in a consistent structure, enabling direct comparisons and trend analysis. Build information and test results are centrally stored and easily accessible via TeamCity dashboards.
2. **UC10.FR_D2:** Analyze and list correlations between code changes and test failures/performance drops.
    a. **Before**: There was no straightforward way to link performance variations to specific code commits or changes.
    b. **After**: Each TeamCity build run captures commit history alongside performance measurements. Users can quickly identify when and how a specific build correlates with performance improvements or regressions. The “Changes” tab in TeamCity provides an at-a-glance view of which revisions were included in each build, helping pinpoint relevant code modifications.
3. **UC10.FR_D3:** Visualize performance trends over time.
    a. **Before**: No consolidated or automated method existed to visualize performance trends.
    b. **After**: TeamCity graphs and dashboards display real-time data on test pass/fail rates and performance deltas relative to a baseline. These visualizations allow developers and stakeholders to quickly assess whether performance is improving, stable, or regressing.
    c. The system also reports on specific performance indicators, such as Grid rendering times or Flow reload times, providing actionable insights into potential bottlenecks.
4. **Key Observations and Outcomes**
    a. **Quantifiable Comparisons**: The build pipelines now measure performance against a known baseline, enabling clear identification of improvements or degradations.
    b. **Trend Monitoring**: Performance data can be tracked build-to-build, illustrating how incremental code changes affect system responsiveness.
    c. **Actionable Insights**: The automated measurements reduce guesswork in debugging and help maintain stable performance in Flow and Grid components, especially for applications handling large datasets.

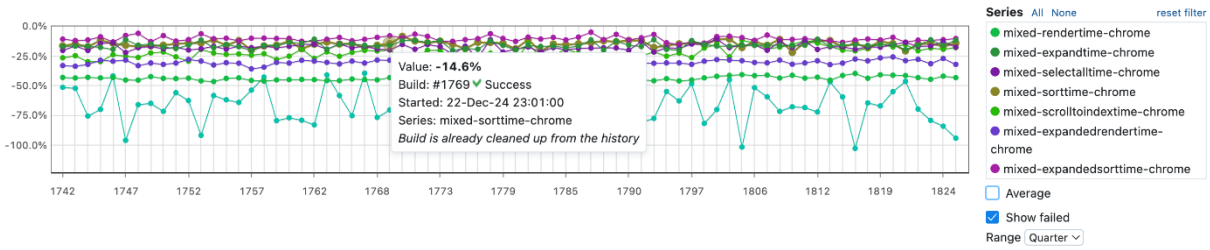


Figure 97: Grid's performance on recent platform version relative to the baseline version

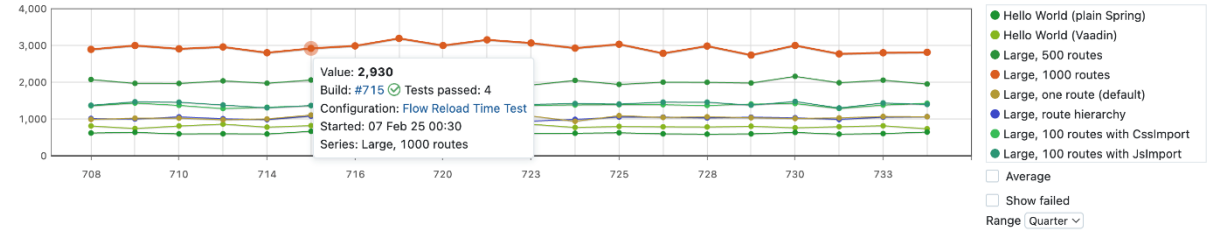


Figure 98: Vaadin 24.7 application reload times

Overall, these enhancements fulfil the KPIs by providing structured, correlated, and visualized performance data. The monitoring solutions allow Vaadin teams to proactively detect, analyse, and address potential regressions, ensuring both Flow and the Grid component remain performant and reliable.

a. Recommendation for industry adoption

The evaluation demonstrates that the SmartDelta Methodology, which integrates both partner solutions and Vaadin's internally developed tools, provides significant benefits for managing complex, business-critical software systems. Mature solutions such as Detangle and Modernization Tools offer substantial improvements by automating key processes such as code analysis, legacy refactoring and API migration. These tools have already proven to reduce manual effort and enhance quality assurance while enabling a seamless migration from legacy frameworks.

However, some limitations remain. The CBTS Plugin, while valuable for major code changes, is less effective in identifying the more subtle modifications that occur in routine development. Additionally, although the classification tools have provided useful insights into workflow practices and reduced the time required for manual processes, their effectiveness depends on an organization's ability to implement strong manual classification practices, which in turn produces a robust training set for these tools.

Overall, for industries facing complex software challenges, adopting this comprehensive methodology and toolset offers a clear path to improved productivity, quality, and scalability.

15. Use-Case 11 from Arcelik

a. Use-Case Description

Arçelik, a global household appliances manufacturer with extensive operations across 57 countries with 55,000 employees, has adopted the SmartDelta methodology to address inconsistencies in software quality and the maturity of Software Development Life Cycle (SDLC) processes. The initiative aims to enhance traceability, standardize processes, and generate actionable insights by monitoring key metrics such as the ratio of pull requests linked to bugs or user stories, and SonarQube issues linked to pull requests. Significant progress has been made in improving traceability and understanding the impact of code quality issues (Stories A and C), while challenges remain in root cause analysis (Story B), which are being addressed with the support of Large Language Models (LLMs). Through collaboration with user case partners and leveraging automated tools, Arçelik strives to reduce manual effort in report generation and achieve meaningful improvements in SDLC quality metrics, ultimately reflecting positive changes in software quality.

b. Link to SmartDelta Methodology

The tools and dashboards in this project aim to streamline the measurement and visualization of metrics across various development and validation processes. Different tools focus on distinct aspects, such as assessing tool performance, tracking application changes over time, and integrating data for meaningful insights.

A key component, the Metric Dashboard, consolidates data from multiple sources and provides an intuitive interface using the Qlik Dashboard tool. Team members can log in to explore detailed metrics, visualized as trendlines with dynamic color indicators that reflect performance trends. Future iterations will incorporate combined metrics for deeper analysis and more actionable feedback.

Additionally, individual tools serve specialized functions:

- ReLink helps match code changes to work items, enhancing traceability.
- PieR leverages AI for issue classification and analysis.
- Other tools focus on various aspects such as automated compliance checks, software evolution tracking, and impact analysis, ensuring a comprehensive and adaptable suite.

This ecosystem of tools enables better decision-making, more efficient workflows, and a data-driven approach to software quality and evolution.

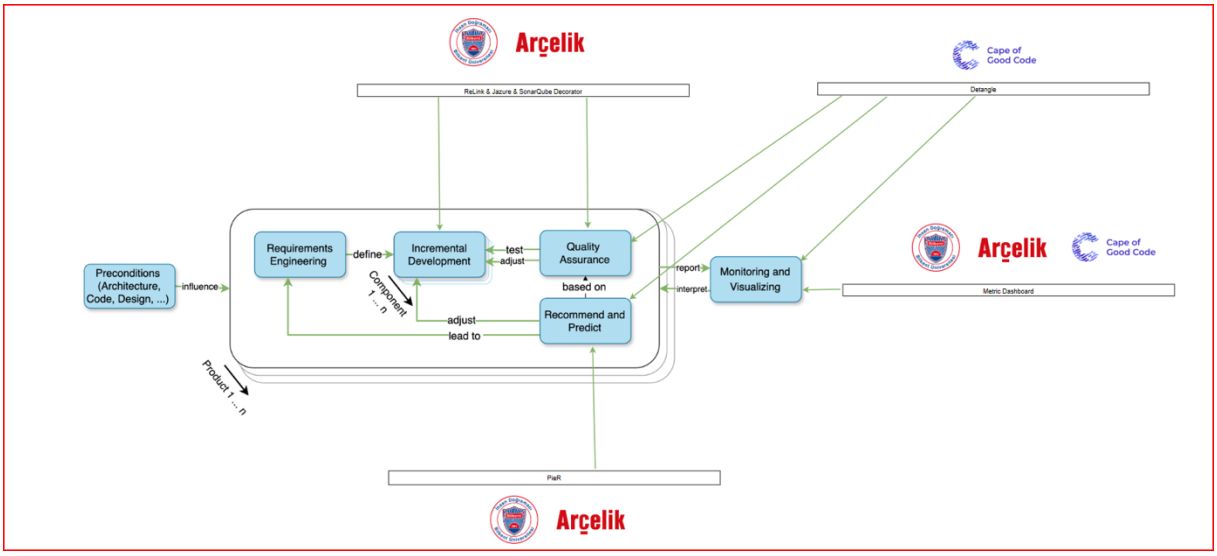


Figure 99: Arcelik and the SmartDelta Methology

c. Tools description

Table 21: Tools for Arcelik's Use-Case

Tool/Application	Purpose
ReLink	Matches Work Items with Code Changes using Machine Learning methods.
Jazure	Transfers Work Items from Jira to the Azure DevOps environment.
Policy for Matching Work Items with Code Changes	Ensures developers match code changes to corresponding Work Items in Azure DevOps.
PieR	Uses generative AI to classify issues and gain insights about code changes.
Detangle	Offers software analytics through a dashboard developed by Cape of Good Code.
Metric Dashboard	Pulls data from multiple sources, visualizes trends, and provides updates on application metrics.
Smellyzer	Analyzes and determines the process quality of code changes.

d. Visualization

Metric Dashboard

The Metrics Dashboard is designed using the Goal-Question-Metric (GQM) methodology to help our organization continuously improve software quality and operational efficiency. Each metric is aligned with a key question that maps to an overarching goal, ensuring that we track meaningful insights. By addressing process inefficiencies, we enhance product quality metrics such as code maintainability, security, and performance. Screenshots provided illustrate how the dashboard visualizes crucial indicators like code smells, vulnerabilities, technical debt, and test coverage, offering actionable insights for developers. Additionally, process-oriented metrics, such as incident resolution times and deployment frequency, help streamline workflows and optimize resource allocation. By leveraging automated analysis tools like SonarQube and Jira integrations, our dashboard facilitates data-driven decision-making, ultimately driving higher software quality, reduced risk, and improved developer productivity.

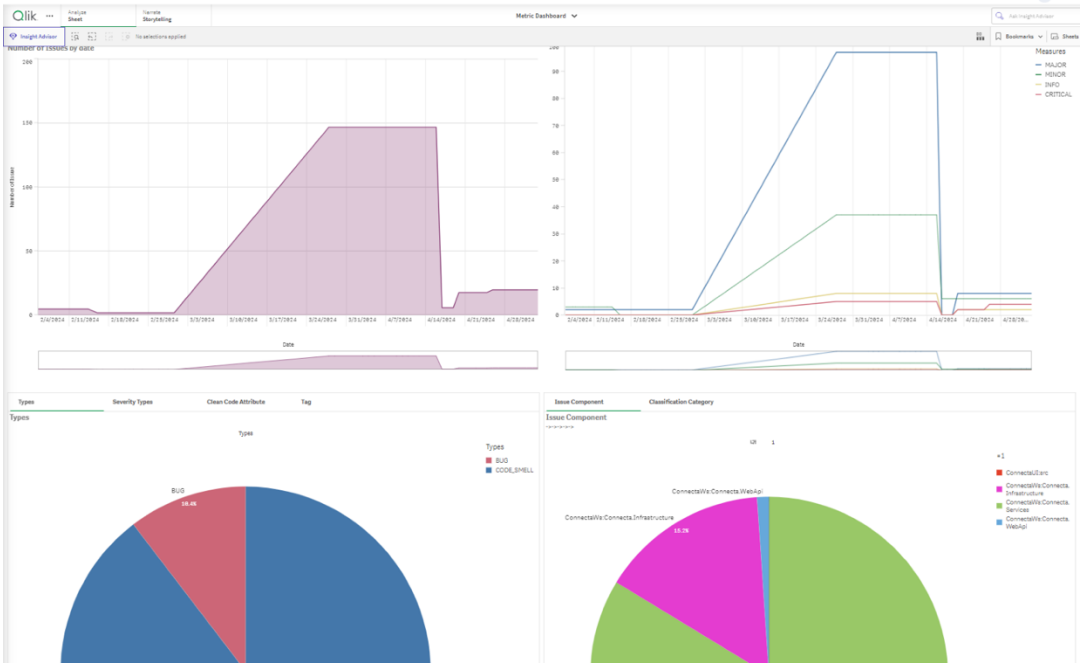


Figure 100: Metric Dashboard: The issues linked with the related pull requests

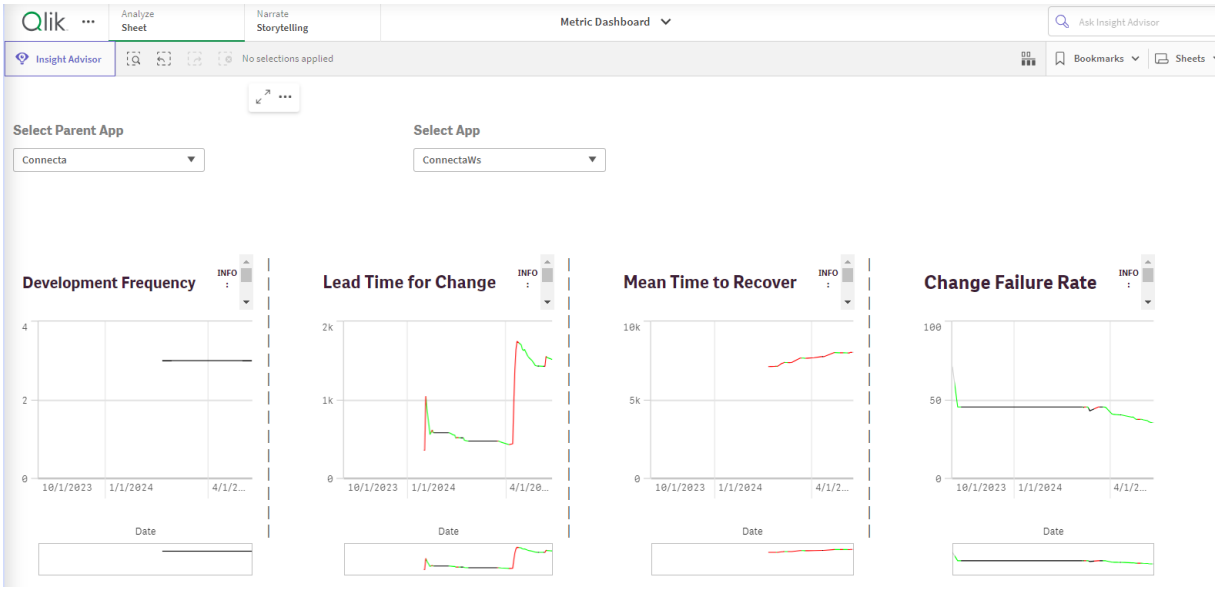


Figure 101: Metric vaad: Dora Metrics

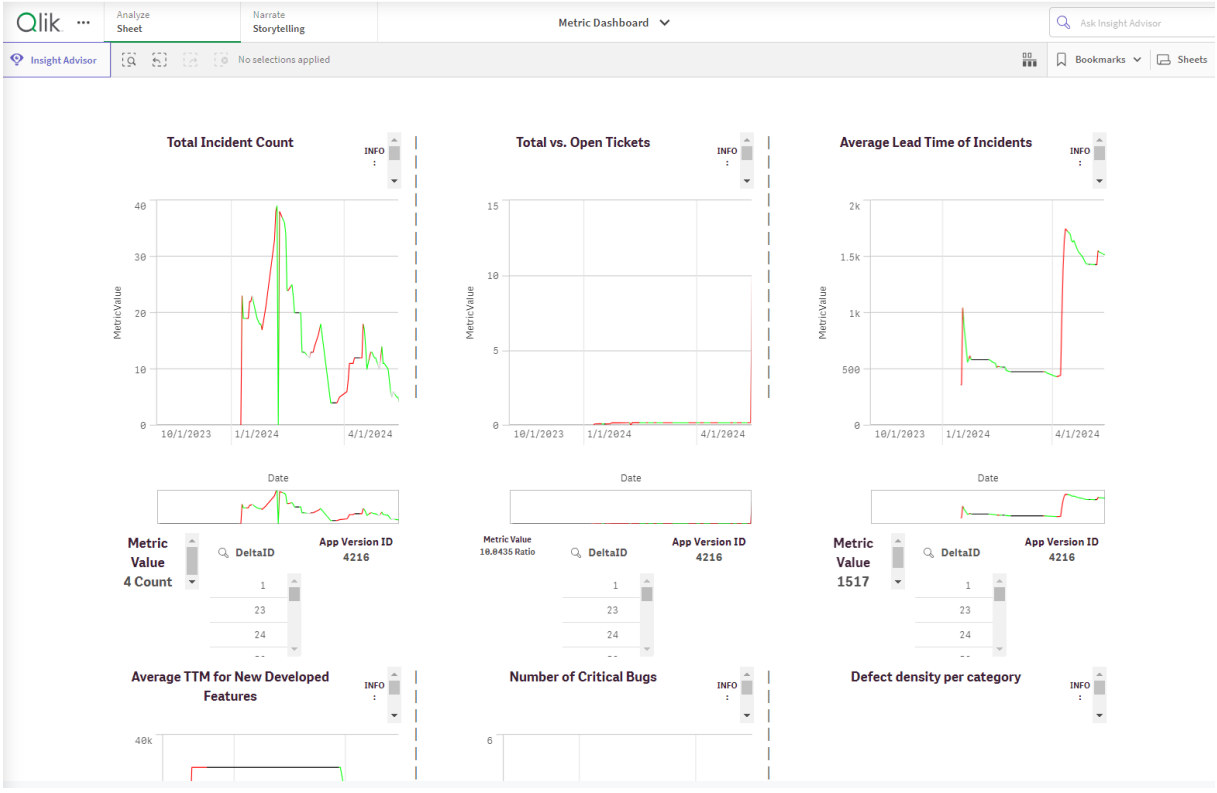


Figure 102: Metric Dashboard: Process Metrics

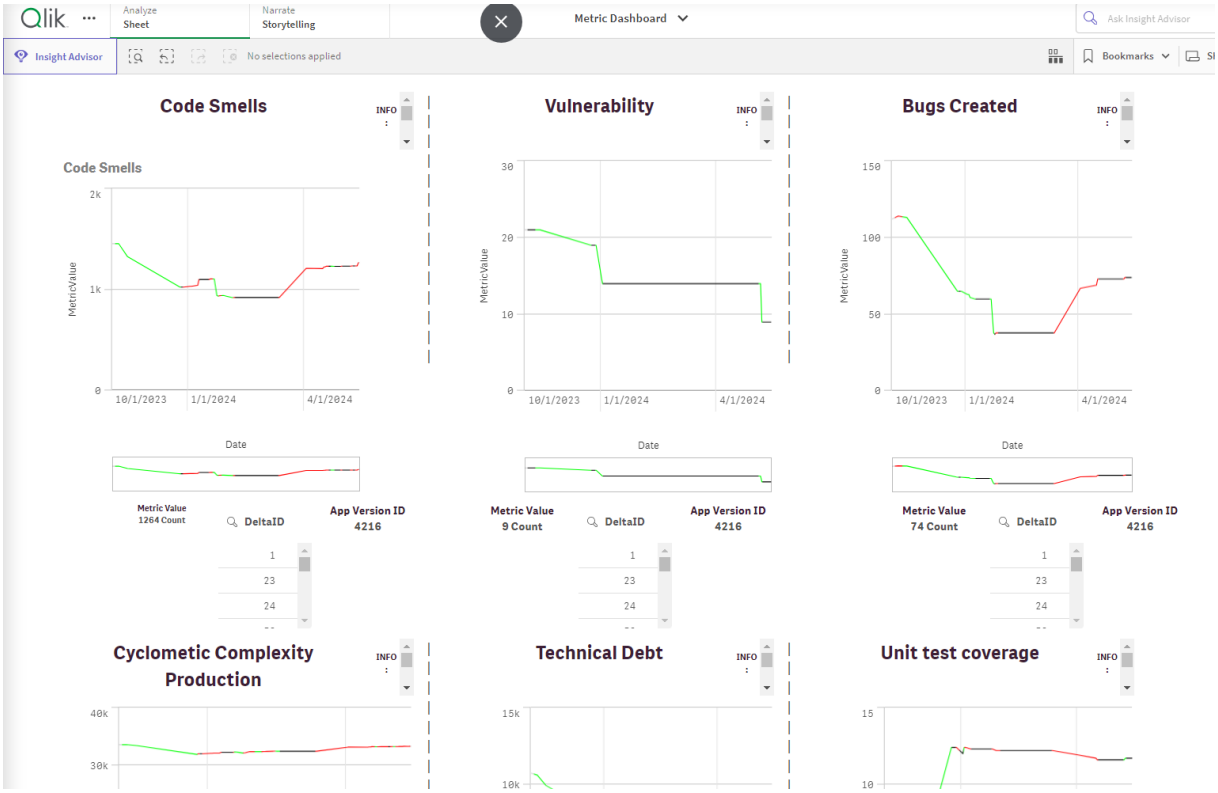


Figure 103: Metric Dashboard: Product Metrics

Visualisation requirements

Visualization requirements were defined based on metric definitions. For most metrics, we used an X-Y axis format, with the Y-axis representing the time delta and the X-axis representing the metric's unit of measurement, as displayed on the dashboard. This setup allowed us to visualize the delta as a trendline and observe how it evolved over time. To illustrate the linkage between PRs and Work Items, we utilized built-in pie charts and tables in Qlik. For summarizing the overall health of quality, radar charts and tables proved to be effective. For another application, we leveraged built-in JavaScript libraries to interpret and visualize the data.

e. Evaluation Setup

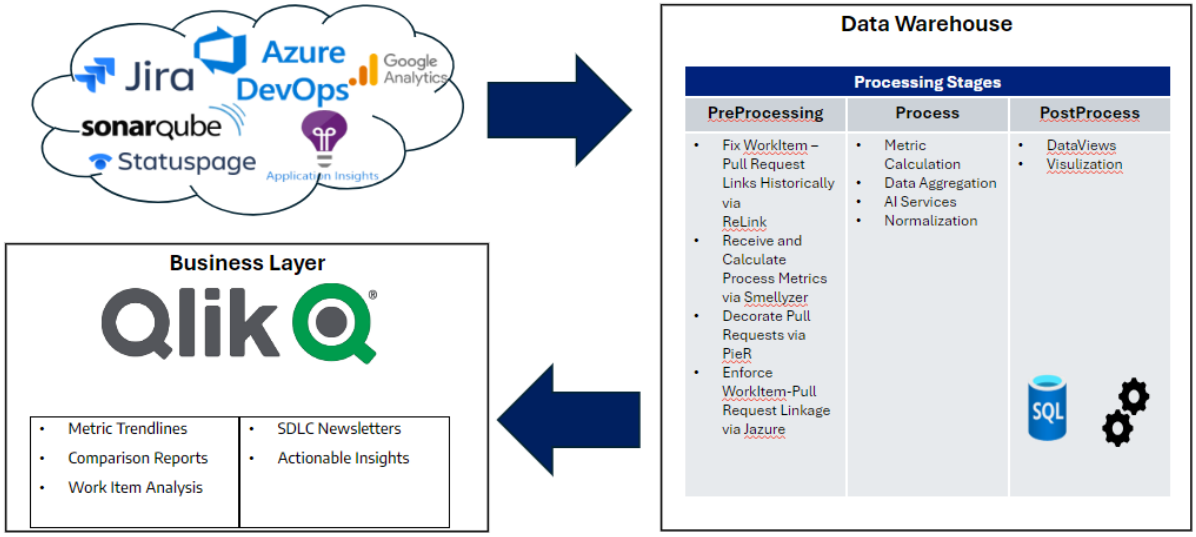


Figure 104: High Level Data Flow

The listed solutions in work packages operate on on-premises servers. Data is fetched from various sources, as illustrated in the figure 104 above. The data processing is visualized in three stages: Pre-Processing, Processing, and Post-Processing.

Pre-Processing: This stage involves not only fetching data from APIs but also making modifications during the development process. The following steps are implemented:

- Repository policies were introduced.
- Work items from Jira were synchronized with Azure using the Jazure Application.
- ReLink was used to fix previously missing links between Pull Requests and Jira work items.
- PieR was utilized to classify, and tag Pull Requests.

Processing: At this stage, data is imported into the Data Warehouse. The Smelyzer and Metric Dashboard's background job analyse and calculate defined metrics. Additionally, The Metric Dashboard integrates with other APIs to connect with LLMs, identifying root causes, generating actionable insights, and normalizing the data.

Post-Processing: At this stage, data is prepared for visualization. Some metrics are aggregated to enhance the overall view, and this is achieved using QLIK.

As a result, SDLC members can utilize the Metric Dashboard to gain a unified view of the metrics. It provides actionable insights, delta reports, sprint newsletters, work item analysis, and metrics displayed as deltas.

f. Evaluation results

Table 22: KPIs Overview

Requirement	KPI Definition	KPI Base Values	KPI Target Values	KPI achieved Value
UC11.FR1	Ratio of pull requests linked with associated bugs/user stories	0.4	0.9	0.84 for past work PRs / 1 for current and future pull requests
UC11.FR4	Ratio of SonarQube code issues linked with related pull requests	0%	90%	100%
UC11.FR5	Ratio of bugs categorized accurately	0%	0,75	0,75
UC11.FR6 UC11.FR7 UC11.FR8 UC11.FR9	Effort reduction for creating a report with valuable insights into sw quality metrics	0%	<1	1

The implemented tools have delivered measurable impacts across various areas. The Software Metrics Dashboard analyzed over 100,000 data points from 10 projects and 72 repositories, offering detailed insights into software performance. PieR facilitated root cause analysis by tagging and analyzing 2,129 pull requests, enhancing process optimization with an average top-3 accuracy of 96.4%. Smellyzer detected over 5,000 process issues, providing actionable data for continuous improvement. Relink linked 84% of prior work items and pull requests using machine learning, significantly reducing manual workload. Finally, Jazure synchronized 13,737 Jira work items with Azure DevOps pull requests, improving cross-platform tracking and collaboration.

Jazure

Work item - Pull Request (PR) linkage has been established through ReLink for the T-1 moment of the software development process. For both the T and T+1 moments, we are enforcing a branch policy that mandates a work item's linkage for it to be merged into the master branch. This policy ensures that merging to the master branch without a linked work item is not possible. As a result, we have achieved a 100% success rate in creating relationships between work items and PRs.

PieR

The PieR tool supports us to achieve KPI UC11.FR5 (Ratio of bugs categorized accurately). We analyzed over 15K pull requests, and in our labeled sample set, the model achieved a top-1 accuracy of 63.6%, top-2 accuracy of 90.9%, and top-3 accuracy of 96.4%. Our target accuracy was 75%, which we were able to achieve with top-2 and top-3 accuracies.

Smellyzer

The Smellyzer tool supports us to achieve KPI UC11.FR1 (Ratio of pull requests linked with associated bugs/user stories). The Figures Smellyzer Figure 105 and 106 represent smell counts each year for bug tracking smells and code review smells, respectively in Connecta. Please see below for quantitative results:

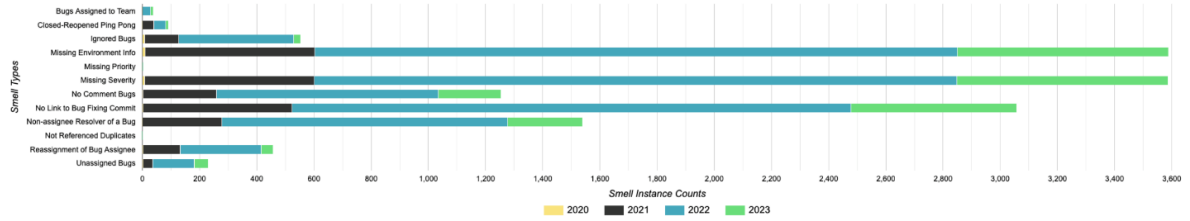


Figure 105: Smellyzer Figure X

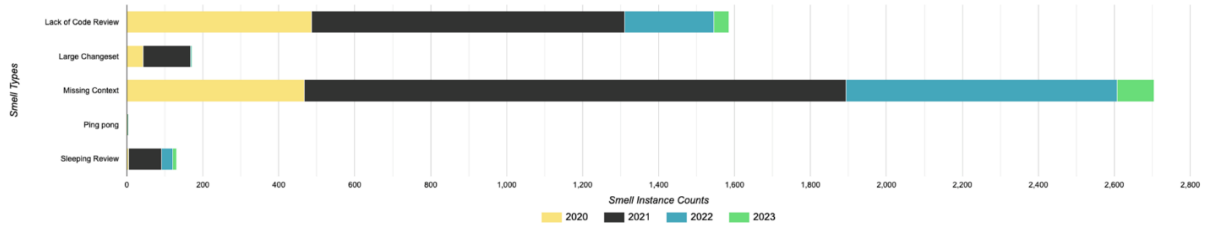


Figure106: Smellyzer Figure Y

Relink

The MRR for Connecta was found to be at a value of 0.84, which signifies that ReLink not only makes accurate recommendations but also ranks them effectively, making it easier for practitioners to identify and select relevant issues. Additionally, the top-1, top-3, and top-5 accuracy for Connecta reached 0.83, 0.85, and 0.86, demonstrating the system's proficiency in accurately associating PRs with the top-5 recommended issue.

g. Recommendation for industry adoption

The SmartDelta methodology has significantly improved software development by enhancing traceability, standardizing processes, and delivering actionable insights. Tools like ReLink and Jazure have strengthened PR-to-work item linkage, while dashboards provide clear visualizations for trend monitoring. AI-powered tools, such as PieR and Smellyzer, automate issue classification and quality assessment, reducing manual effort and driving efficiency.

However, challenges like root cause analysis and scaling the methodology to diverse environments remain. Future improvements should focus on refining analytical capabilities, enhancing tool integration, and adapting SmartDelta for evolving SDLC practices. With these advancements, SmartDelta can further solidify its role as a transformative methodology in the industry.

16. Industrial Use-Cases and Project KPIs results

10 Project KPIs are evaluated through the present evaluation. Based on the requirement evaluation for each use-case, we define the project KPI achievement.

Table 23: Evaluated project KPIs

KPI Number	KPI Description	Initial & expected target values
KPI_1.1	Automated translation of requirements into models.	Automated translation of 80% of requirements into models. The baseline includes no automated translation mechanisms for requirements.

KPI_1.2	Automated reuse of system artifacts based on quality trends.	50% reduction of mean time to resolution compared with manual reuse approach. The baseline time needed for reuse analysis for requirements is estimated by partners to be between 200 to 300 hours per project. F1-measure of 0.7 in classifying the artifacts. F1 is a measure of a test's accuracy in statistical analysis of binary classification.
KPI_1.3	Automated validation of delta artifacts and quality recommendations based on static analysis, testing and bug fixing.	Decreasing by 50% the mean time to detect anomalies and average time to patch bugs or vulnerabilities, today fixed in more than 245 days [11]. We also aim to have 80% of requirements to be tested automatically as well as 80% of fault coverage for test cases. We aim to reduce flakiness levels by 2-5%. The baseline for flakiness is reported to represent between 2 to 10% of all test cases between releases [12] and reflects the current state of use case considered.
KPI_1.4	Automated analysis and visualization of system execution results.	We aim to identify 50% more test scenarios or quality issues so that they can be taken into account to improve coverage measurements using different views on the data, such as visualizations per test case and per code branch, as well as statistics per release.
KPI_1.5	Integrated SmartDelta toolset with the current continuous development practices in industry for fast feedback loops	Integration of 5 industrially-common tool sets in order to introduce quality protection (operations) and optimization (development) tool chains in SmartDelta.
KPI_1.6	Demonstrable implementations.	Successful handling of several industrial case studies with at least 100k lines of code for both software and testware by SmartDelta framework leading to demonstrable implementations.
KPI_2.1	Maximizing SmartDelta Automation and Adoption	50% reduction of mean time to enable quality protection and optimization means, compared with the manual approach.

KPI_2.2	Product Delivery Effort Reduction	20% reduction of development effort (direct and indirect costs) in terms of total hours for specification work and testing in a period of one year (repeated twice during the project) with measurements performed per subsystem compared to two other closely-related variant projects. The baseline measurements will be performed during case study setup using historical information, but initial baseline information indicate a 1-3 year product delivery estimate.
KPI_2.3	Defect Prevention	Reducing the introduction of system-related defects by 20%. As a baseline, around 50% of the total number of defects are introduced at requirement and design phases as well as bad fixes during continuous development before the implementation of SmartDelta solutions. This will be measured via all defects closed over effort spent during development and calculated using the fixed defects for a 3 months window/net development effort for a 3 month window (repeated twice during the project). The metrics are giving a high-level view about average defect density which is changing slowly and allows teams to improve in the long term based on long term trends.
KPI_2.4	Quality Improvement	Quality Improvement in the range of 15-25%. We use the rate of build success and change failure rate (e.g., lead to service impairment, require a hotfix, a rollback, a fix-forward or a patch). The baseline shows a change failure rate of 16-45% [14] when the release frequency is between once per month and once e-very six months.

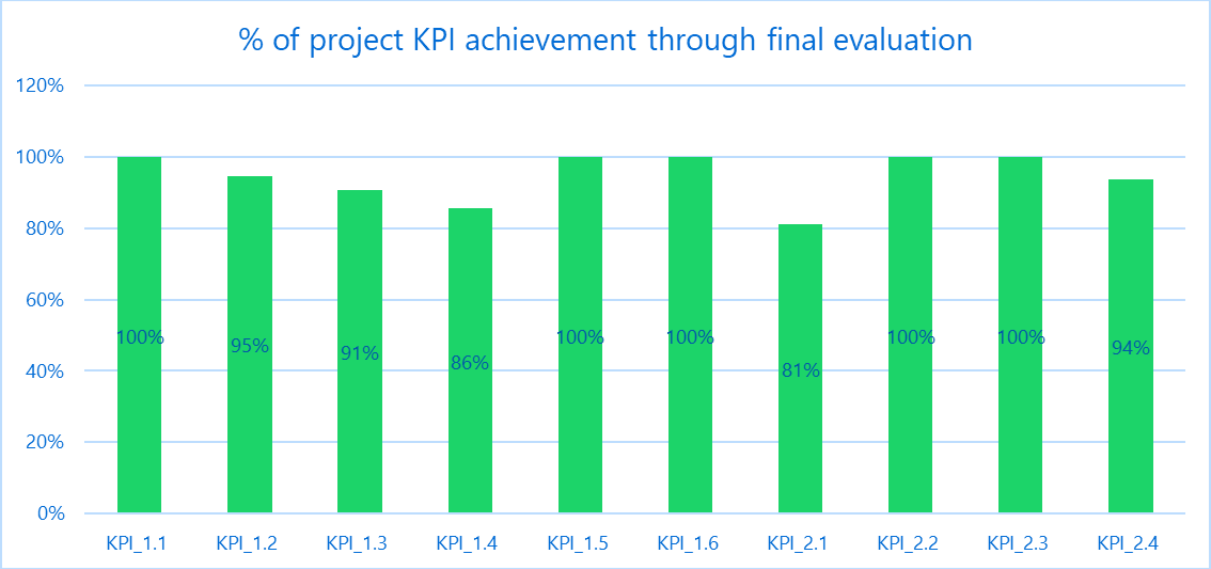


Figure 107: Project KPI achievements

2 KPIs have an achievement less than 90%:

KPI_1.4: 14 related requirements from 16 are met. The 2 partially satisfied requirements have shown improvements by 30% and 25% which are below the expected 50%.

KPI_2.1: 26 related requirements from 32 are met for this KPI. The 6 partially satisfied requirements have shown improvements between 30 and 40% which is below the expected 50%.

The final evaluation of the industrial use cases revealed that, in general, SmartDelta, its methodologies, and the tools it has developed have accomplished the anticipated objectives. However, the two KPIs with an achievement of less than 90% are noteworthy, as the decline in performance was only found in two use cases.

17. Implications for Industry

The SmartDelta project, which focused on automated quality assurance and optimization in the context of incremental industrial software development, has yielded promising results across a range of industry use cases.

Eleven participating organizations provided feedback on the project's methodology and associated tools, with a common theme being the significant improvement in efficiency and quality through automation.

Several participants highlighted the benefits of enhanced traceability and standardized processes, with tools such as ReLink and Jazure improving the connection between project requirements and work items, while dashboards facilitated trend monitoring. AI-powered tools such as PieR and Smellyzer automated issue classification and quality assessment, reducing manual effort.

One participant specifically noted the value of the SmartDelta Methodology in identifying relevant tools and solutions throughout the development lifecycle. Another participant emphasized the methodology's structured guidelines and its ability to suggest new approaches, particularly for comparing large volumes of software artifacts. Several organizations reported success in automating testing processes. The SONATA platform exhibited potential in automated test case identification and code reuse; however, further development is necessary to enhance artifact indexing and natural language processing accuracy. The TIGER+ tool demonstrated promise in accelerating testing

activities by reducing the number of tests while maintaining a high fault detection rate; nevertheless, its effectiveness is contingent on well-structured and traceable requirements.

In the domain of security, SmartDelta tools have been shown to enhance the capabilities of Security Operations Centers (SOCs) by improving incident response times and optimizing resource management. The machine learning (ML)-enhanced QRadar framework has been demonstrated to enable automatic anomaly detection and offense prioritization, leading to faster and more accurate threat detection. The AILA tool has been demonstrated to have the capacity to automatically assign security labels to issues, thereby expediting expert assignment and improving software security and quality. The integration of AISA and CSI with SmartDelta recommendations establishes a framework for artifact reuse, code quality assessment, and error prevention.

In the banking sector, participants anticipate enhanced efficiency, performance, and quality of their primary banking software through the facilitation of metric tracking, architectural quality analysis, and technical debt calculation. However, challenges persist, including the management of voluminous performance datasets and the assurance of IT security approval for shared data. Other use cases have highlighted the potential for microservice analysis and monitoring, as well as the use of dashboards for health monitoring and AI-driven predictions in charging systems.

While the project has shown substantial progress, challenges remain.

Several participants have mentioned the need for further development in areas like root cause analysis, scaling the methodology to diverse environments, refining analytical capabilities, and enhancing tool integration. One participant has emphasized the need for continuous improvement in real-time threat intelligence processing. Another participant highlighted the importance of training engineering teams on the effective use of the tools. Despite the aforementioned challenges, the overall outlook is positive, with participants acknowledging the transformative potential of the SmartDelta methodology and its associated tools in improving productivity, quality, and scalability in industrial software development.

18. References

[1] SmartDelta Methodology Users and Developers Guidelines
https://itea4.org/project/workpackage/deliverable/document/download/433/SmartDelta%20D2.4%20-%20SmartDelta%20Methodology%20Users%20and%20Developers%20Guidelines.pdf

[2] K. J. Kevin Feng (2024). Cocoa: Co-Planning and Co-Execution with AI Agents

preprint arXiv:2412.10999, 2024•arxiv.org