SmartDelta

Automated Quality Assurance and Optimization in Incremental Industrial Software Systems Development

D5.1 – State of the Art on Software Quality Visualization

Submission date of deliverable: May 31, 2022

Edited by: Wasif Afzal (Mälardalen University, Sweden), Hakan Kilinc (NetRD, Turkey), Eray Tuzun (Bilkent University, Turkey), Sebnem Köken (Erste Software, Turkey), Benedikt Dornauer (University of Innsbruck, Austria), Michael Felderer (University of Innsbruck, Austria), Mircea-Cristian Racasan (c.c.com, Austria), Muhammad Abbas (RISE Research Institutes of Sweden AB, Sweden), Sarmad Bashir (RISE Research Institutes of Sweden AB, Sweden), Holger Schlingloff (Fraunhofer FOKUS, Germany), Abhishek Shrestha (Fraunhofer FOKUS, Germany), Robin Gröpler (IFAK, Germany), Björn Otto (IFAK, Germany), Jean Malm (Mälardalen University, Sweden), Andrea Pabón-Guerrero (Universidad Carlos III de Madrid, Spain), Julian Rojas-Bolaños (Universidad Carlos III de Madrid, Spain), Ioana Petre (BEIA, Romania), Daran Smalley (Alstom Transport, Sweden), Serge Demeyer (University of Antwerp, Belgium)

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	See affiliations in “Edited by” above.
Version number	V1.0
Work package	WP5
Work package leader	Sebnem Köken (Erste Software, Turkey)
Dissemination level	Public

Description *(max 5 lines)*	The goals of D5.1 (part of T5.1) are (i) to obtain a broad view and understanding of modern technologies that can be applied at all levels during the implementation of the SmartDelta Visualization components, and (ii) to do a state-of-the-art and practice analysis to identify existing solutions for visualization of software quality, their capabilities, and limitations.

Executive Summary

Visualizations are key to understanding complexities driven by loads of data generated by modern software-based systems. Modern ways to develop software-based systems tend to focus on delivering quick value to the end customer. Doing so is made possible by software development paradigms of agile and continuous development with the support of an ever-increasing set of supported tools. While the focus is on delivering quick value to the end customer, development teams are in a need to rapidly reduce wasteful effort, to anticipate problems in advance and to resolve quality issues efficiently. This deliverable is driven by this need to draw a picture of state-of-the-art and state-of-the-practice methods, tools and technologies for visualizing software development artefacts, software evolution and system properties. The contents of this deliverable are driven by SmartDelta's consortium partners representing some of the leading universities, technology, and complex system providers in Europe. In this deliverable, we cover visualizations supporting quality of requirements, design, code and testing as well as visualizations supporting monitoring and diagnostics through log analysis. We further focus on visualizations supporting system properties of performance & resource consumption and security & privacy. The deliverable also contains a snapshot of the existing technology stacks targeting visualizations. Overall, the deliverable provides a bird's-eye view of existing visualization research and enabling technology in the domain to enhance quality in the development of complex software-based systems.

Table of contents

Document Glossary

Acronym	Description
AI	Artificial Intelligence
ASD	Agile Software Development
API	Application Programming Interface
AWS	Amazon Web Services
CDN	Content Delivery Network
CI/CD	Continuous Integration / Continuous Delivery
CPN	Coloured Petri Nets
CPU	Central Processing Unit
CRISTA	Code Reading Implemented with Stepwise Abstraction
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
CVE	Common Vulnerabilities and Exposures
CVS	Concurrent Versions System
CWE	Common Weakness Enumeration
Concept	Comprehension of Net-Centered Programs and Techniques
DL	Deep Learning
DAST	Dynamic Application Security Testing
DevOps	Development (Dev) and Operations (Ops)
DOM	Document Object Model
DSM	Dependency Structure Matrix
EID	Ecological Interface Design
ELK	Elasticsearch, Logstash, and Kibana
EMC	Extended Misuse Case
GASP	Geometric Animation System, Princeton
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HWAS-V	Holistic Web Application Security Vulnerability Visualization
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JS	JavaScript
JSON	JavaScript Object Notation

Acronym	Description
LOC	Lines of Code
MBT	Model-Based Testing
ML	Machine Learning
MOOSE	Montreal Object-Oriented Slicing Environment
MTTR	Mean Time to Recover
MySQL	My Structured Query Language
NFP	Non-Functional Property
NPM	Node Package Manager
OWASP	Open Web Application Security Project
PHP	Hypertext Preprocessor
PIM	Personal Information Map
PPVM	Privacy Policy Visualization Modelling
QMs	Quality Models
Q-Rapids	Quality-Aware Rapid Software Development
RQA	Requirements Quality Analyzer
SAST	Static Application Security Testing
SHrimp	Simple Hierarchical Multi-Perspective
SIG	Softgoal Interdependency Graphs
SNMP	Simple Network Management Protocol
SPLE	Software Product Line Engineering
SMRC	System Requirements Capturing Model
STRATOS	STRATegic release planning Oversight Support
SVG	Scalable Vector Graphics
TSDB	Time Series Database
UI	User Interface
UML	Unified Modelling Language
URL	Uniform Resource Locator
V&V	Verification and Validation
VITRum	Visualization of Test-Related Metrics
WASC	Web Application Security Consortium
WMC	Weighted Method per Class
ZAP	OWASP Zed Attack Proxy

1. Introduction

The complexity of modern embedded systems is on the rise, e.g., in the automotive industry the number of computers per car has increased to hundreds [1]. Also, embedded system companies are developing systems using more rapid, continuous, and agile development methods and practices [2]. With continuous practices, there is an increased amount of information generated from the software process as the development and test cycles become more frequent [3]. Making sense of such information and data for quality enhancements can be both challenging and rewarding if done properly. For example, Ahmad et al. [4] argue that if test results data are not presented well, time and attention may be wasted while making critical decisions.

Visualization is a process of “representing data, information, and knowledge in a visual form to support the tasks of exploration ... and understanding” [5]. This is a mental process that occurs when interacting with visual elements. The visual elements are, in turn, representations of data that have been collected, stored, fetched, filtered, transported, and transformed to and from some data storage [2]. Software visualization offers insight of software system’s structure, algorithms as well as the analysis and exploration of software systems and anomalies. Some benefits of software visualization are that it allows you to combine and relate data of software systems that are not inherently related, point out technical debt, reveal changes to the source code and bug reports, and identify, measure, and improve code and feature quality [42].

In WP5 of SmartDelta, different solutions to visualize the software quality characteristics will be developed. In particular, the objective is to illustrate quality degradation and improvement trends over time so that end-users can quickly and effortlessly understand when and where in the system issues with respect to quality characteristics have occurred. The output of WP5 will thus be the SmartDelta visualization dashboard which gets its inputs from the solutions and services of WP2 to WP4. To support the technology selection for the dashboard and to design it, we here in D5.1 do a state of the art and practice analysis to identify existing solutions for visualization of software quality, their capabilities and limitations.

The table of contents of this deliverable is inspired by the feedback from a survey that was sent out to all the partners in SmartDelta (knowledge providers, technology providers and use case providers). The feedback of the survey was discussed in multiple SmartDelta consortium meetings, and this helped shape the contents of this deliverable.

The rest of the deliverable is organized as follows. Section 2 targets visualizations of software development artefacts and software quality attributes. This includes visualizations supporting quality of requirements, design, code, testing, monitoring, and diagnostics. This section also, amongst other topics, includes visualizations supporting system properties of performance, resource consumption, security, and privacy. Section 3 focuses on technologies for software visualization while Section 4 is on visualization styles. Section 5 summarizes the key recommendations from all the topics covered while conclusions are given in Section 6.

2. Visualizations of Software Development Artefacts and Software Quality Attributes

2.1. Visualizations Supporting Quality of Requirements and Design

Visualization of software development artifacts (e.g., requirements and design documentation) can be effectively used to support human insight and reasoning and increase software development productivity [77]. To minimize the development time and cost, early detection of the quality issues (i.e., inconsistencies, incompleteness, and incorrectness) in system requirements plays a vital role in enhancing the quality of the developed system [78].

Quality issues can be detected through semi-formal representation with graphical notations of the requirements. Osama et al. [79] proposed a semi-formal model called the SRCM to visualize the system requirements in a unified view. The model's objective is to identify a broader range of quality issues and capture a better understanding of textual requirements and their relations. SCRM comprises four main modules: system entities, system preconditions, system actions, and action sources. System entities represent all the unique domain entities of a system. System preconditions represent the components of a system. The action represents the change to happen in response to its precondition. Action Sources are a bridge between system preconditions and their respective actions. For quality checks, redundant requirements can be detected through SCRM by locating repeated action sources. This is because duplicate system components of the same type are stored only once in SCRM with unique keys, therefore, triggering the action sources to map the redundant requirements to the same keys.

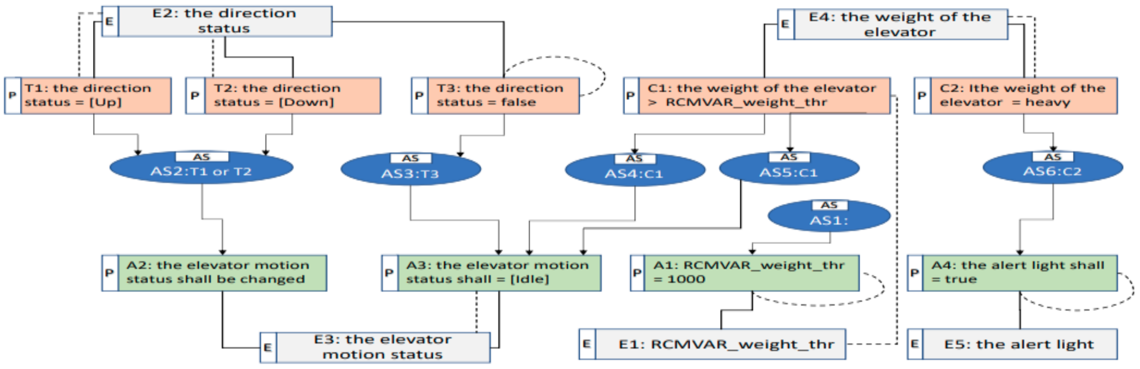
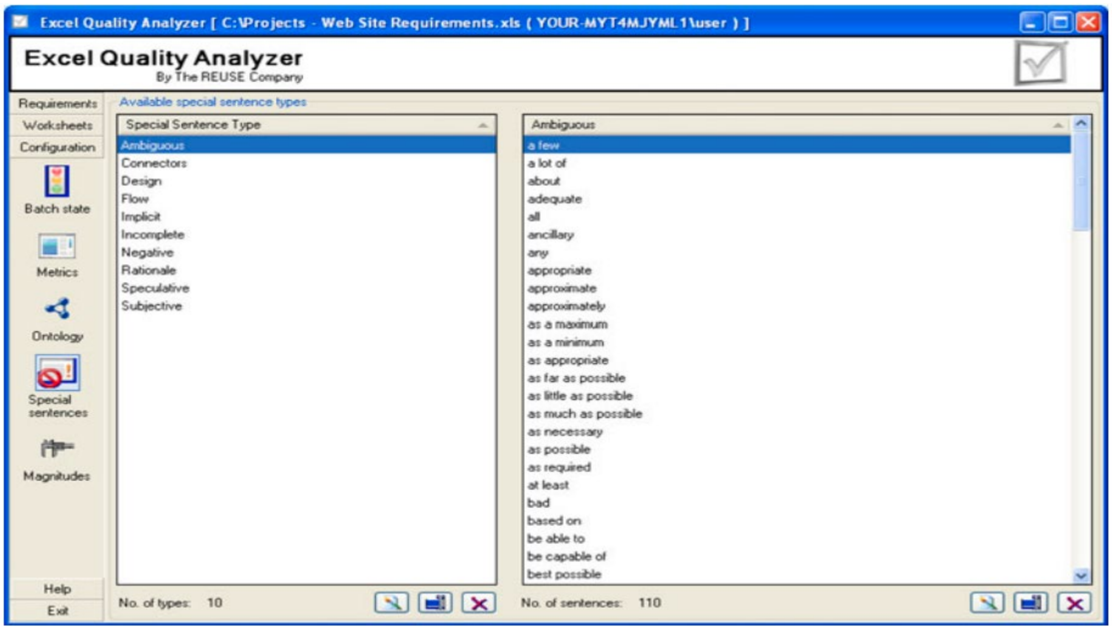

E2: the direction status
E4: the weight of the elevator
T1: the direction status = [Up]
T2: the direction status = [Down]
T3: the direction status = false
C1: the weight of the elevator > RCMVAR_weight_thr
C2: Ithe weight of the elevator = heavy
AS2:T1 or T2
AS3:T3
AS4:C1
AS5:C1
AS1:
AS6:C2
A2: the elevator motion status shall be changed
A3: the elevator motion status shall = [Idle]
A1: RCMVAR_weight_thr = 1000
A4: the alert light shall = true
E3: the elevator motion status
E1: RCMVAR_weight_thr
E5: the alert light

Figure 1. Requirements View as in [79].

SCRM provides visualization views for entities and requirements. The Entities View gives the complete dependency visualization of requirement entities to represent the effect of change in one entity requirement on another dependent requirement entity. The Requirements Views is a structural form of the all or subset requirements given a system, which increases the expressiveness of requirements for all the stakeholders. Moreover, Gonzalo et al. [80] proposed a framework to evaluate the quality of textual requirements in an automated fashion with the goal of improving efficiency in the requirements engineering process.

In addition, RQA [81] tool is used to implement the framework and evaluate the low-level quality indicators (e.g., requirement size, number of ambiguous terms, imperative verbal forms, and overlapping requirements). The RQA tool has the capability to implement quality measurement principles and techniques. The input data of the tool are the requirements to compute the quality metrics and give the recommendations as output.



Special Sentence Type

Ambiguous
Connectors
Design
Flow
Implicit
Incomplete
Negative
Rationale
Speculative
Subjective

No. of types: 10

Ambiguous

a few
a lot of
about
adequate
all
ancillary
any
appropriate
approximate
approximately
as a maximum
as a minimum
as appropriate
as far as possible
as little as possible
as much as possible
as necessary
as possible
as required
at least
bad
based on
be able to
be capable of
best possible

No. of sentences: 110

Figure 2. Configuration of Quality Metrics: Define list of ambiguous sentences [80].

The RQA tool can be used by different collaborators in an organization to completely utilize the capabilities of the tool. The Quality department defines the quality metrics, weights, thresholds, etc. The Project manager evaluates the requirements of a given project, and the Requirement Engineer is facilitated by the tool to write requirements effectively in the first place, utilizing the defined metrics and procedures.

To address the concerns of quality-related (e.g., usability, security, and performance) requirements of a system, Rahimi et al. [82] proposed a data mining approach to extract and model the impact of quality-related requirements from an existing set of requirements. The proposed approach comprises detecting quality-related requirements by applying Machine Learning and generating goal graphs with the contextualization of domain-level information. Finally, the approach visualizes the results to support effective decision-making for stakeholders.

Visualization plays an important role in understanding the concerns related to the quality of requirements to support compliance verification further, generation of architectural design and assessment, supporting change impact analysis, and effectively communicating with customers.

SIG notation is used to model quality concerns and their trade-offs of a particular system. To reflect the interdependencies between different quality concerns, each concern is modelled as a soft goal and to satisfy each of the goals, multiple trade-offs must be made.

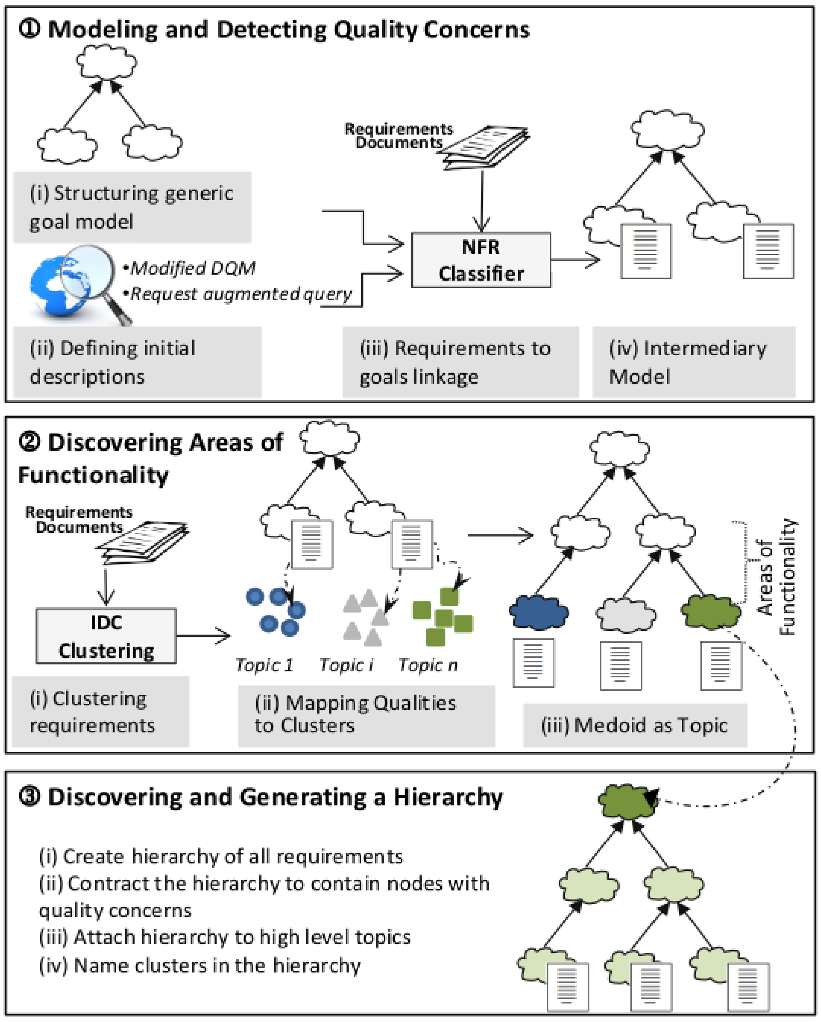

① Modeling and Detecting Quality Concerns
(i) Structuring generic goal model
•Modified DQM
•Request augmented query
(ii) Defining initial descriptions
Requirements Documents
NFR Classifier
(iii) Requirements to goals linkage
(iv) Intermediary Model
② Discovering Areas of Functionality
Requirements Documents
IDC Clustering
(i) Clustering requirements
Topic 1
Topic i
Topic n
(ii) Mapping Qualities to Clusters
Areas of Functionality
(iii) Medoid as Topic
③ Discovering and Generating a Hierarchy
(i) Create hierarchy of all requirements
(ii) Contract the hierarchy to contain nodes with quality concerns
(iii) Attach hierarchy to high level topics
(iv) Name clusters in the hierarchy

Figure 3. Overview of the SIG construction and visualization process [82].

In the first stage, quality concerns are modelled to classify and extract the related requirements from the requirements specifications. In the second stage, topic areas are clustered and assigned to their relevant quality concerns. In the final stage, candidate SIG is generated by organization requirements attached to each topic into a low-level hierarchy.

In a model-based design environment, models (e.g., in UML or Simulink) can also be regarded as high-level requirements. There are tools such as MES MXAM [148], which try to visualize quality problems by highlighting the respective parts of (the graphical description of) the model. By clicking on a design rule (e.g., “the control flow is from left to right or from top to bottom”), all parts of a model which do not follow this rule (i.e., control flow arcs which lead from right to left or from bottom to top) are highlighted. Other tools like MES MQC [100] use traditional methods like function graphs, pie charts and bar charts to present the evolution of certain quality parameters over time.

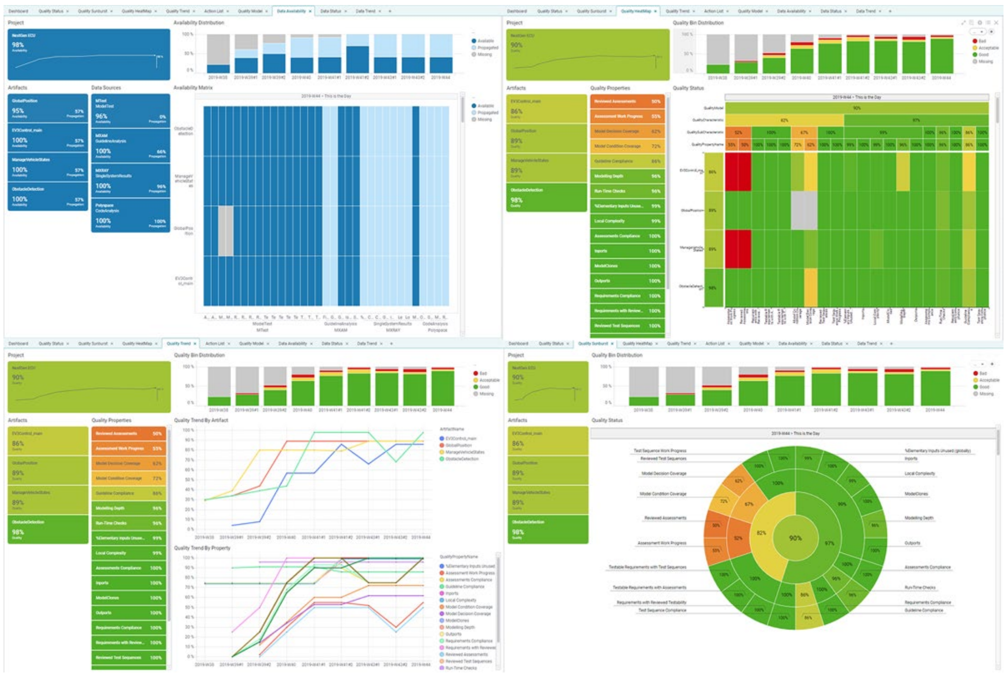


Figure 4. Quality dashboard of MES MQC [100].

The MATHWORKS requirement dashboard information is extracted from [101].

"*A traceability matrix displays links between items in Model-Based Design artifacts such as Requirements Toolbox™ objects, Simulink® model elements, Simulink Test™ objects, and MATLAB® code lines. You can apply filters and focus only on the items that you want to see. You can use the matrix to identify unlinked items and implement them in your design.*"

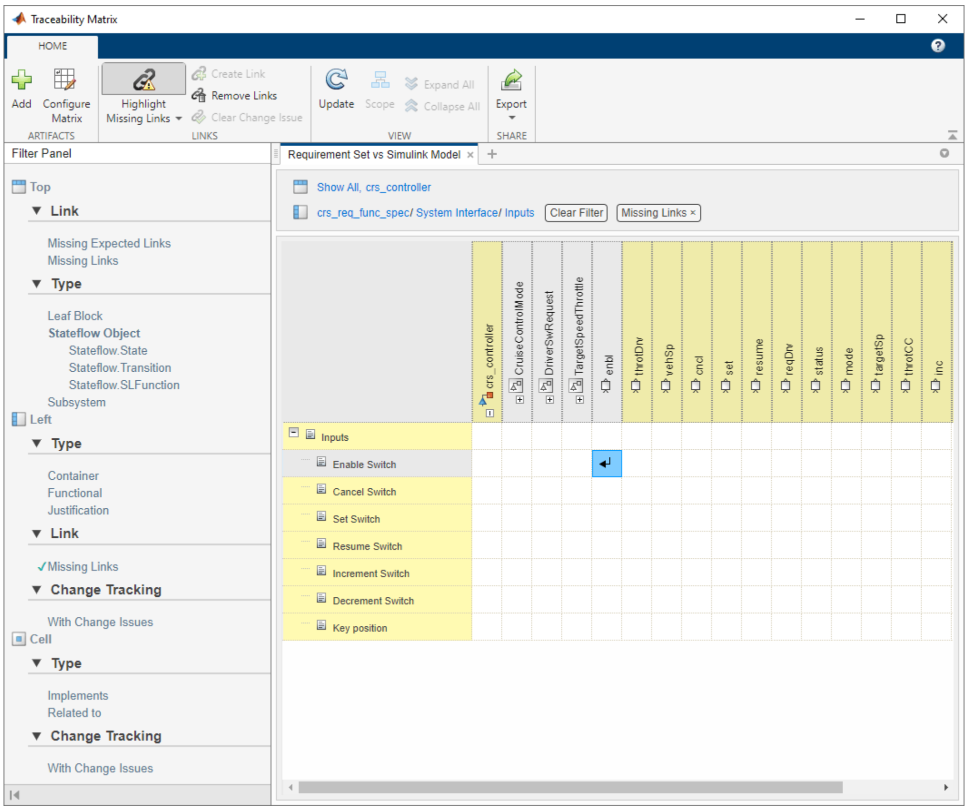
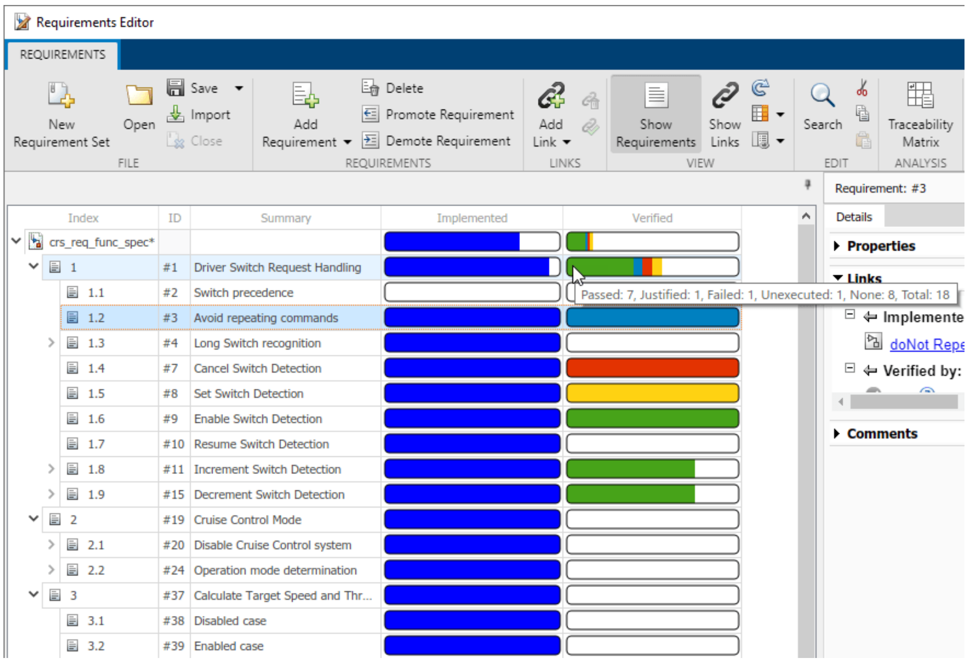


Figure 5. MATHWORKS requirements traceability matrix [101].

The MATHWORKS requirement verification status information is extracted from [102].

"*Review Requirements Verification Status: You can view the verification status of your requirements in the Requirements Browser and Requirements Editor. The verification status reflects results from simulation testing using Simulink® Test™ or property proving using Simulink Design Verifier™*".



Index	ID	Summary
crs_req_func_spec*		
1	#1	Driver Switch Request Handling
1.1	#2	Switch precedence
1.2	#3	Avoid repeating commands
1.3	#4	Long Switch recognition
1.4	#7	Cancel Switch Detection
1.5	#8	Set Switch Detection
1.6	#9	Enable Switch Detection
1.7	#10	Resume Switch Detection
1.8	#11	Increment Switch Detection
1.9	#15	Decrement Switch Detection
2	#19	Cruise Control Mode
2.1	#20	Disable Cruise Control system
2.2	#24	Operation mode determination
3	#37	Calculate Target Speed and Thr...
3.1	#38	Disabled case
3.2	#39	Enabled case

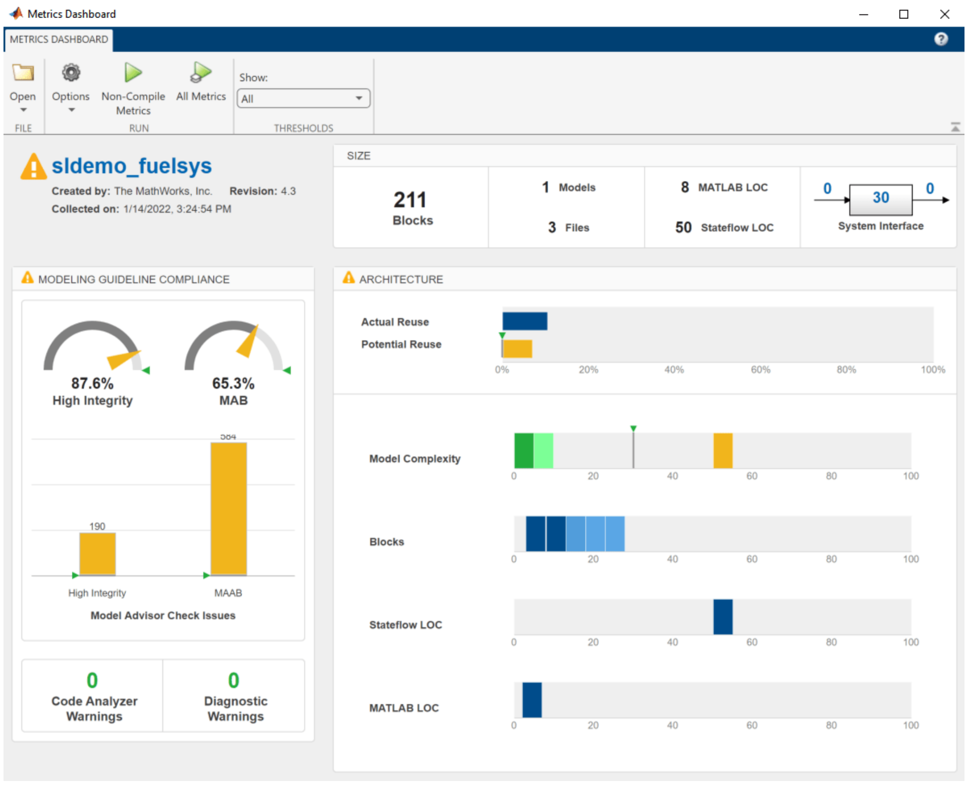
Passed: 7, Justified: 1, Failed: 1, Unexecuted: 1, None: 8, Total: 18

Figure 6. MATHWORKS requirements editor with design quality status information [103].

The MATHWORKS design quality status information is extracted from [103].

"*Collect Model Metric Data by Using the Metrics Dashboard*

To collect model metric data and assess the design status and quality of your model, use the Metrics Dashboard. The Metrics Dashboard provides a view into the size, architecture, and guideline compliance of your model."



sldemo_fuelsys
Created by: The MathWorks, Inc. Revision: 4.3
Collected on: 1/14/2022, 3:24:54 PM

211 Blocks
1 Models
3 Files
8 MATLAB LOC
50 Stateflow LOC
0 30 0 System Interface

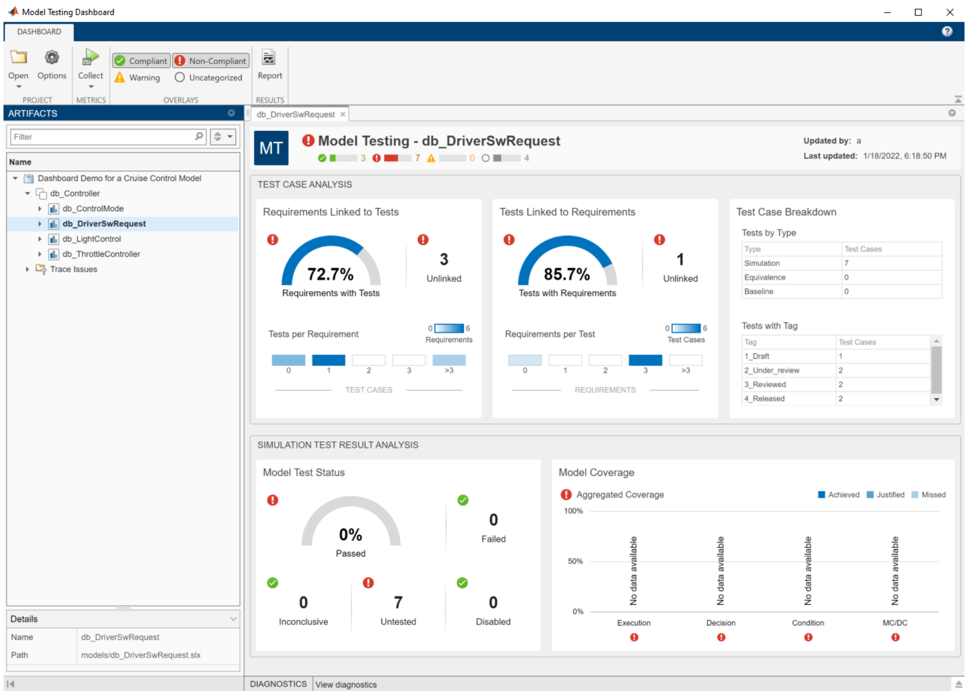
0 Code Analyzer Warnings
0 Diagnostic Warnings

Figure 7. MATHWORKS project metrics dashboard [104].

The MATHWORKS project dashboard information is extracted from [104].

"*Model Testing Dashboard collects metric data from the model design and testing artifacts in a project to help you assess the status and quality of your requirements-based model testing.*

The dashboard analyzes the artifacts in a project, such as requirements, models, and test results. Each metric in the dashboard measures a different aspect of the quality of the testing of your model and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C."



Updated by: a
Last updated: 1/18/2022, 6:18:50 PM

Tests by Type

Type	Test Cases
Simulation	7
Equivalence	0
Baseline	0

Tests with Tag

Tag	Test Cases
1_Draft	1
2_Under_review	2
3_Reviewed	2
4_Released	2

Details

Name	db_DriverSwRequest
Path	models/db_DriverSwRequest.slx

Figure 8. MATHWORKS model testing dashboard.

2.2. Visualizations Supporting Quality of Code and Testing

Visualization of code quality has mainly been concerned with architectural issues. There are several tools to visualize code with software architecture diagrams, e.g., Sotograph, Sonargraph, and Sotoarc. One common way to represent hotspots of an analysis is via so-called heat maps. The following picture is from [105].

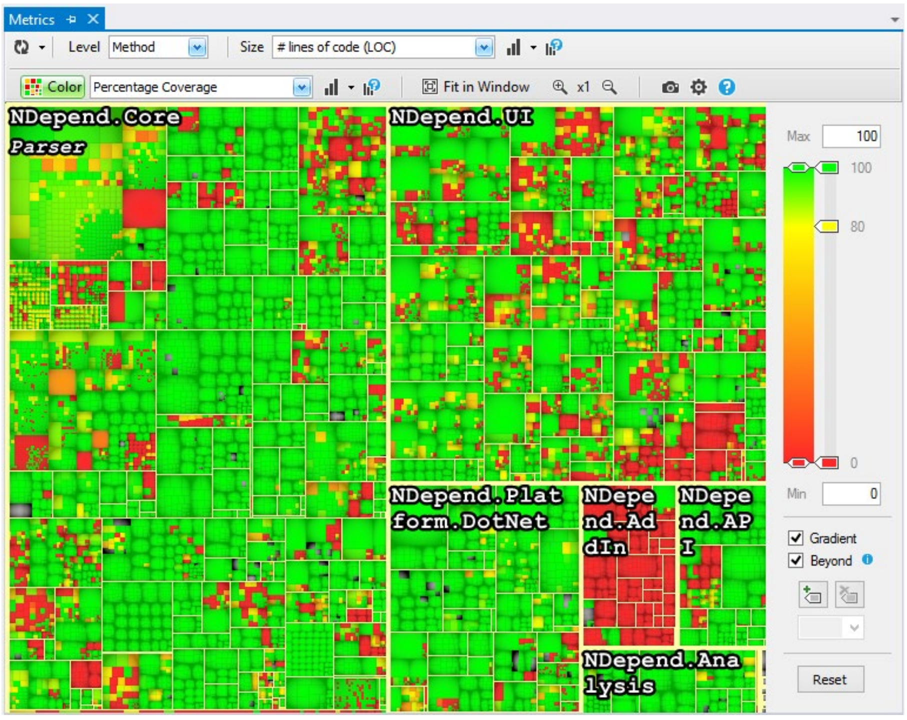


Figure 9. Software heat map [105].

Each rectangle represents a method, module or packet; the area is proportional to the size or lines of code of that particular program fragment. Larger rectangles indicate the code hierarchy: classes, namespaces and projects that group their child elements. The colour of a method rectangle represents the intended metrics, e.g., frequency of change or percentage of code coverage ratio. A 3D-extension of this idea leads to the idea of so-called “software cities”. The following example is from [106].

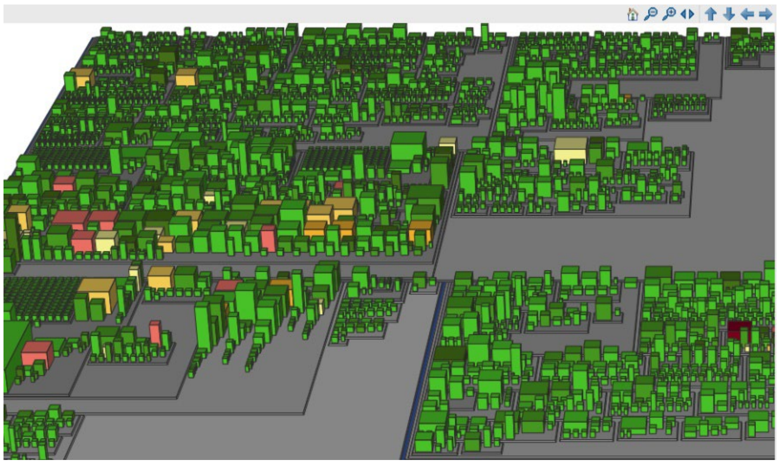


Figure 10. Software city [106].

In such a representation, again the ground area of each “building” is proportional to the size of a source file (measured in LOC). The height of the building reflects, e.g., the complexity of the code fragment, measured via some metrics such as McCabe or simply in form of average indentation. The colour is derived from the number of changes within a certain period.

Another way to look at a software architecture is via the graph of connections between modules. This allows, e.g., to spot cyclic dependencies between different parts. The simplest way is via a list of modules and their connections. The following picture is from B. Merkle "Stop the software architecture erosion: building better software systems", [107].

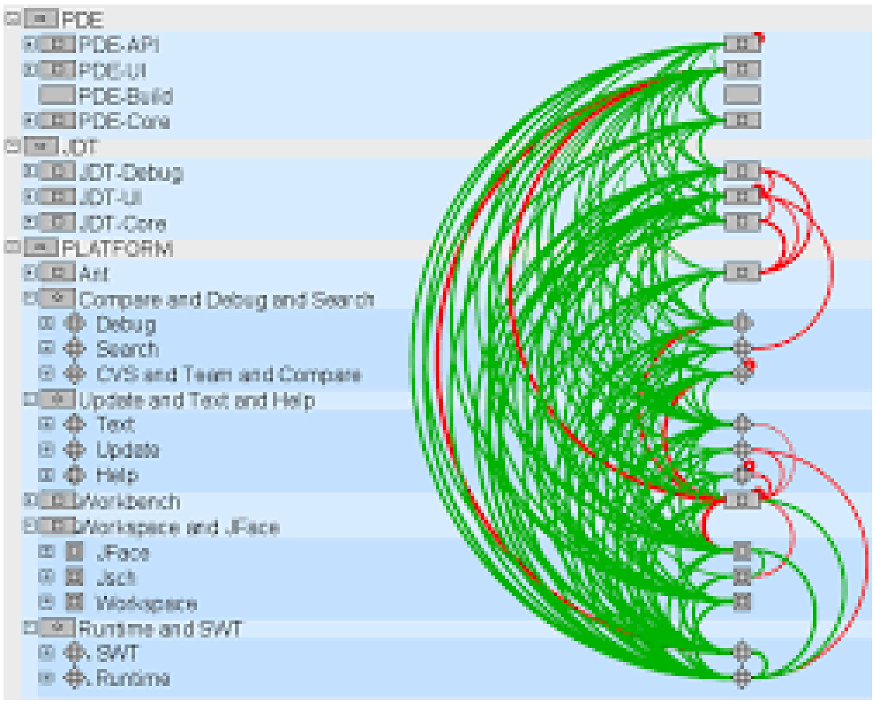


Figure 11. Software module dependency graph, linear view [107].

However, it is also possible to arrange the modules in a 2-Dimensional or 3-Dimensional world to identify "clusters" and "outliers". The following picture is taken from [108].

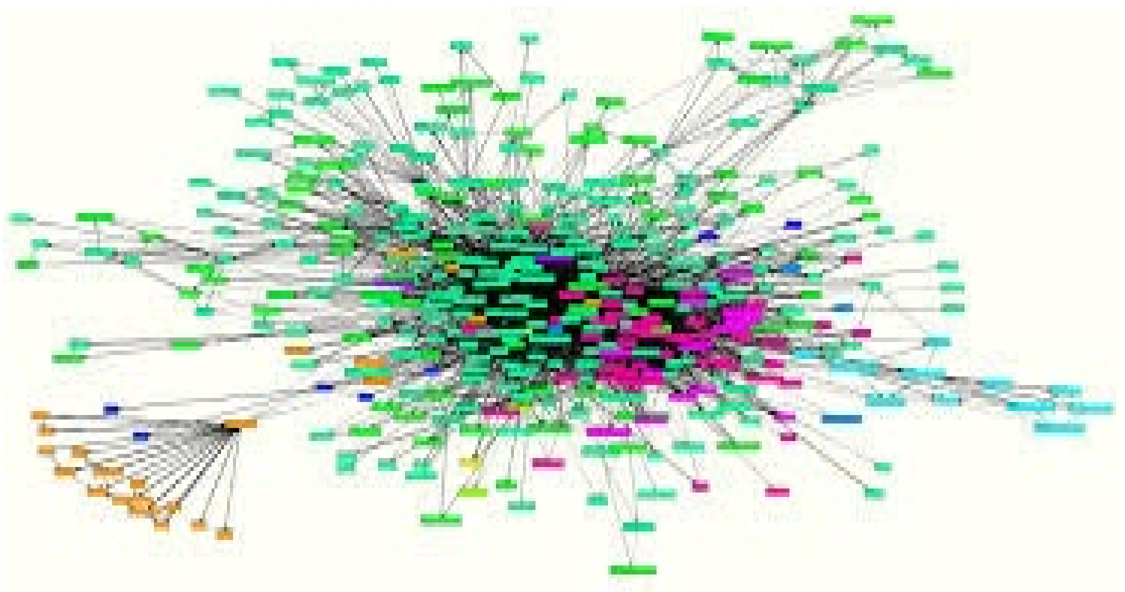


Figure 12. Software module dependency graph, explosion view [108].

The X-Ray public domain tool [109] by Jacopo Malnati provides a system complexity view and class and package dependency view for a given Java project. The height of a box indicates the lines of code, the width the number of methods in a class. Edges represent inheritance between classes. Nodes are arranged in vertical (top-down) trees that highlight inheritance hierarchy.

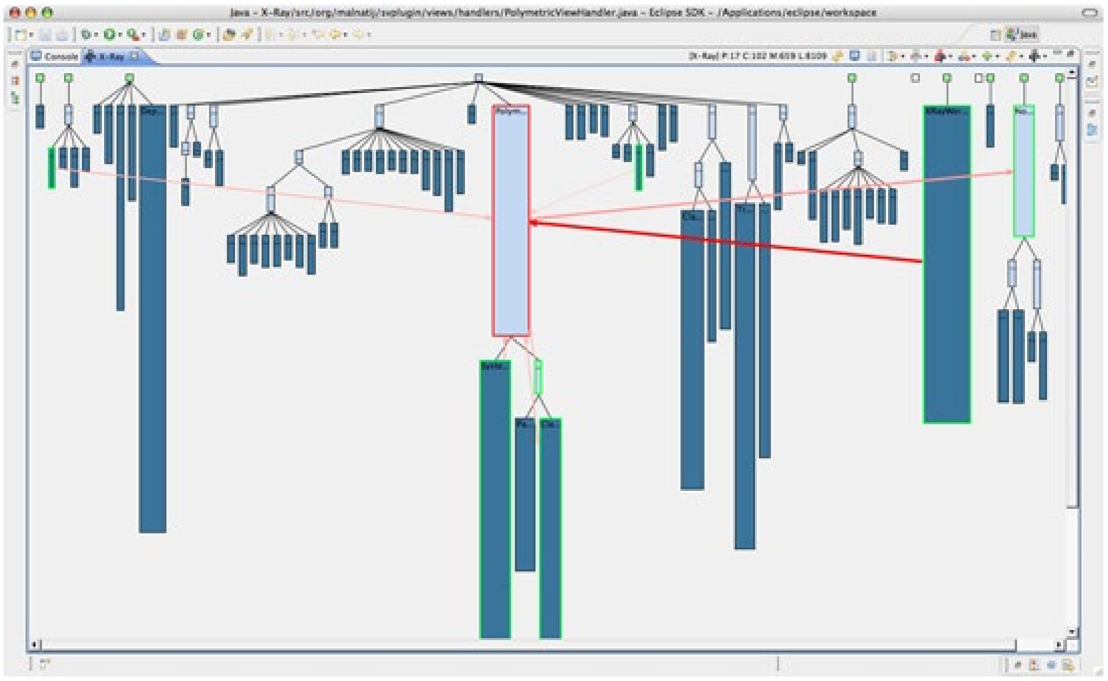


Figure 13. Class / package tree [109].

Class and package dependencies are shown by arranging the items in a circular fashion:

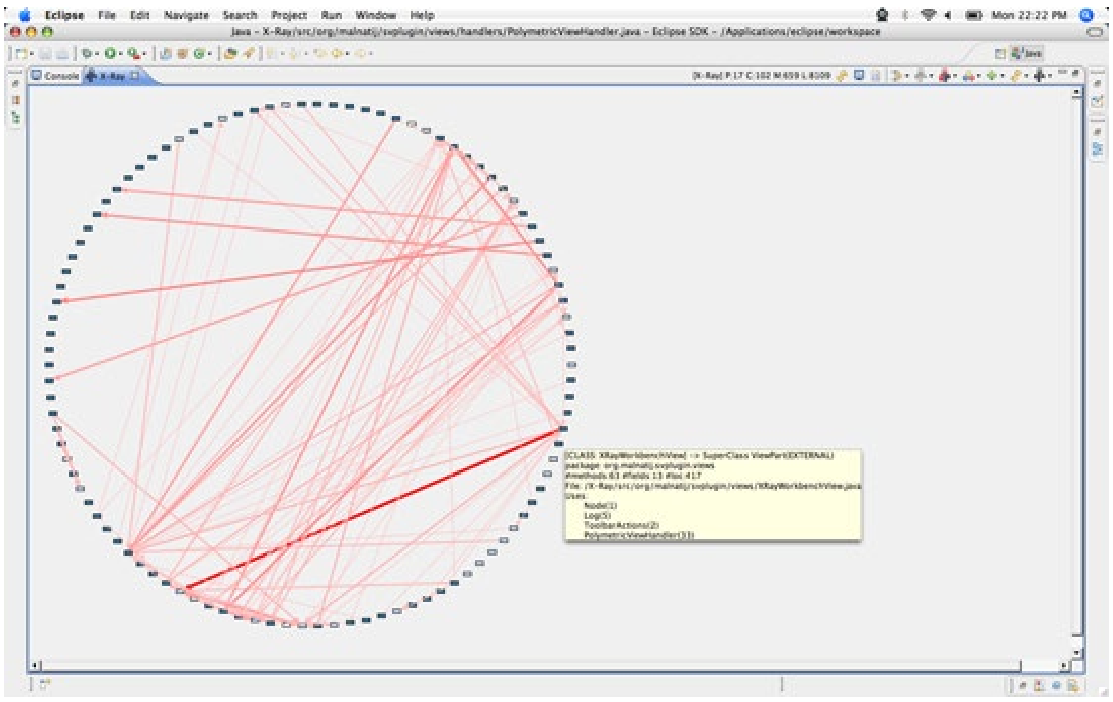


Figure 14. Class / package dependency graph, circular view [109].

In contrast to such dependency graphs, it may be simpler to view dependencies between modules as a software structure matrix: The following picture (taken from [110]) contrasts dependency structure matrix (DSM) and dependency graph of some software. The DSM shows immediately that the structure is layered because there is no cycle aggregated around the matrix diagonal. Also, it shows that some elements are more used than others through horizontal blue lines. The graph on the right is quite unreadable and doesn't provide such information.

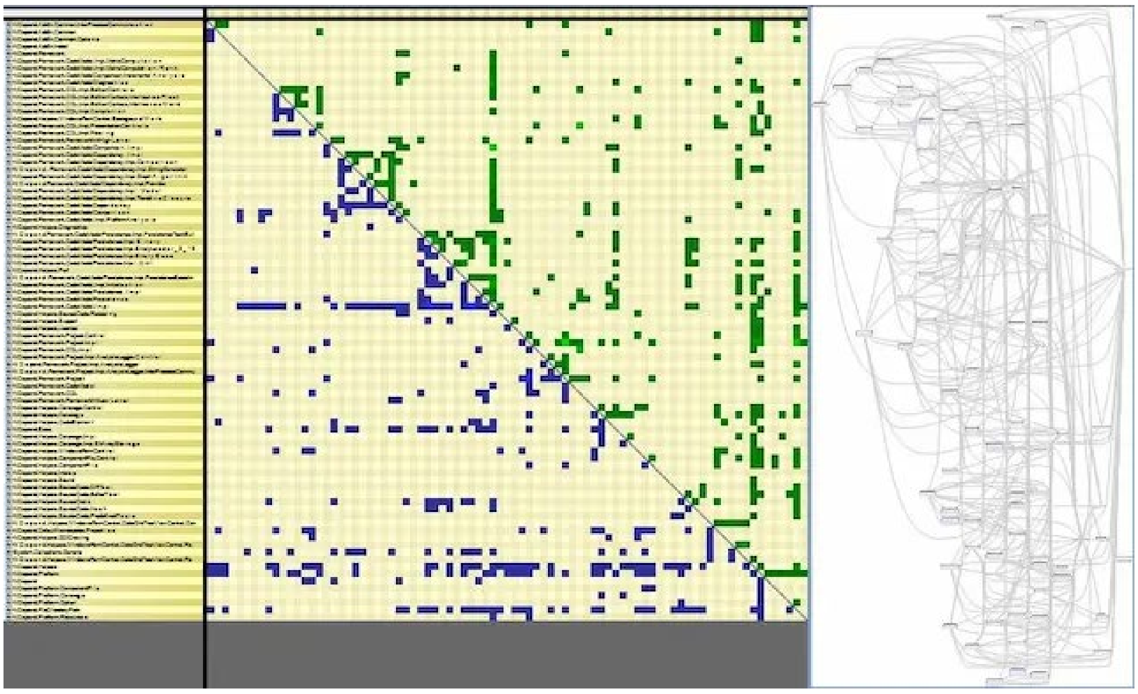


Figure 15. Software structure matrix compared to connection graph [110].

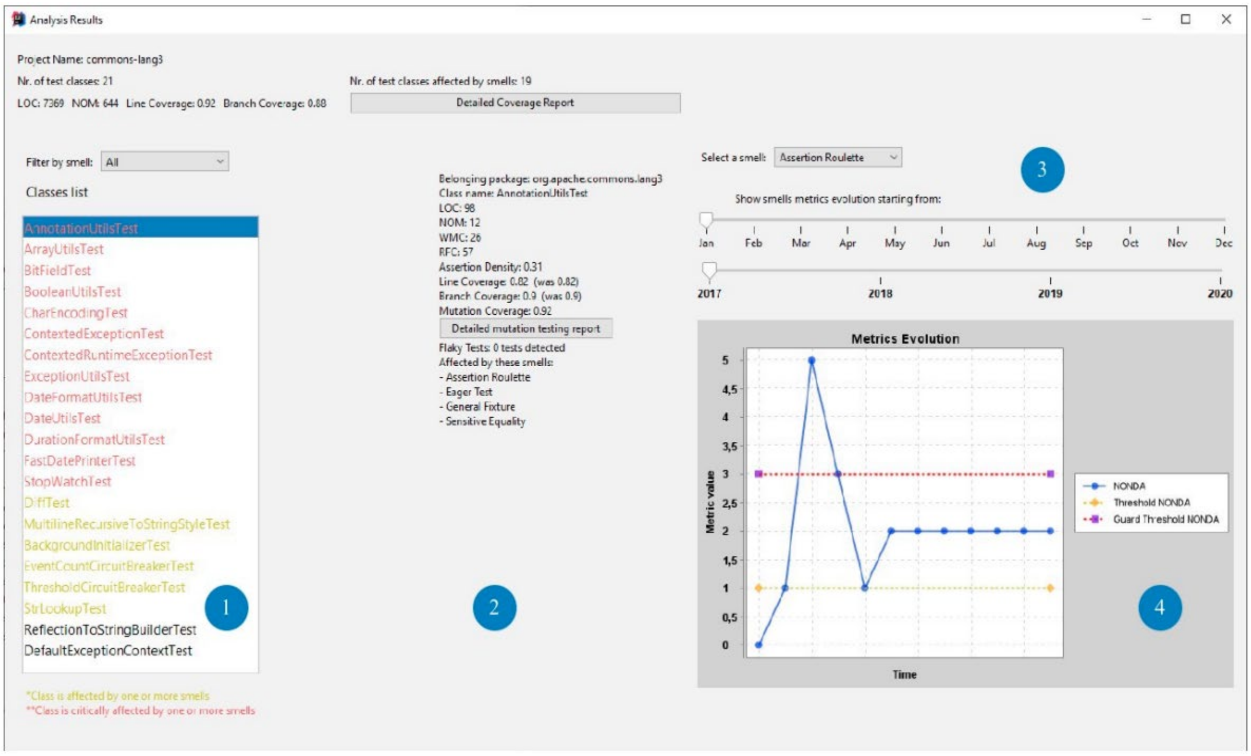
In a software testing context, the visualizations support hypothesis generation, decision-making, and can map to information needs [4]; e.g.:

- Is this the same bug as I saw yesterday?
- Has the bug already been fixed?
- Is this bug only present on some types of devices?
- What version of the test case did we run?
- Why was my new test case not selected for testing tonight?
- Is the level of quality sufficient for this feature to be considered completed?
- Can we release the software? etc.

Strandberg et al. [2] use the term *test results exploration and visualization* for when users interact with a system that generates visual elements to support answering these types of questions.

Visualization of source code coverage information helps developers better comprehend the production code and find critical bugs [83]. Software Testing has been identified as the main key to improving the quality and reliability of a software system, and the metrics are specifically important as they gauge the progress, quality, and health of software systems. Visualizing quality metrics aims to enhance the understanding and analysis of information for software systems [84].

To facilitate developers' analysis of source code and to investigate the capabilities of software tests, Fabiano et al. [85] proposed a tool named VITRum, which consists of an interface for several test code-related factors.


Analysis Results

Project Name: commons-lang3
Nr. of test classes: 21
LOC: 7369 NOM: 644 Line Coverage: 0.92 Branch Coverage: 0.88
Nr. of test classes affected by smells: 19

Filter by smell: All

Classes list

AnnotationUtilsTest
ArrayUtilsTest
BitFieldTest
BooleanUtilsTest
CharEncodingTest
ContextedExceptionTest
ContextedRuntimeExceptionTest
ExceptionUtilsTest
DateFormatUtilsTest
DateUtilsTest
DurationFormatUtilsTest
FastDatePrinterTest
StopWatchTest
DiffTest
MultilineRecursiveToStringStyleTest
BackgroundInitializerTest
EventCountCircuitBreakerTest
ThresholdCircuitBreakerTest
StrLookupTest
ReflectionToStringBuilderTest
DefaultExceptionContextTest

*Class is affected by one or more smells
**Class is critically affected by one or more smells

Belonging package: org.apache.commons.lang3
Class name: AnnotationUtilsTest
LOC: 98
NOM: 12
WMC: 26
RFC: 57
Assertion Density: 0.31
Line Coverage: 0.82 (was 0.82)
Branch Coverage: 0.9 (was 0.9)
Mutation Coverage: 0.92
Flaky Tests: 0 tests detected
Affected by these smells:
- Assertion Roulette
- Eager Test
- General Fixture
- Sensitive Equality

Select a smell: Assertion Roulette

Show smells metrics evolution starting from:

Figure 16. VITRum main panel [85].

Figure 16 shows the quality metrics report after computation. On the left side, Panel 1 represents all the ordered test classes based on the criticality of test smells of a system under analysis. Panel 2 contains all the calculated metric values for the selected test class. Panel 3 is for users to select the start date to consider for analysis. Panel 4 depicts a plot to analyse the test quality metrics over time since the start of the VITRum execution on a given software system.

Furthermore, there is another popular tool for automated code review; SonarQube [38]. SonarQube provides services, e.g., detection of bugs, vulnerabilities, and code smells in the code. It has the capability of continuous code inspection across software project branches and pulls requests by integrating with existing workflows.

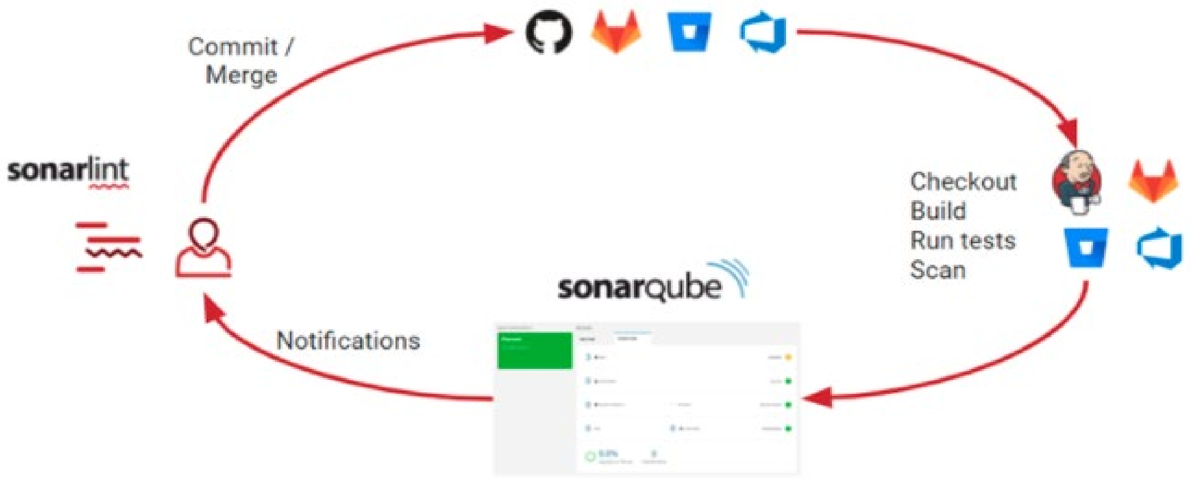
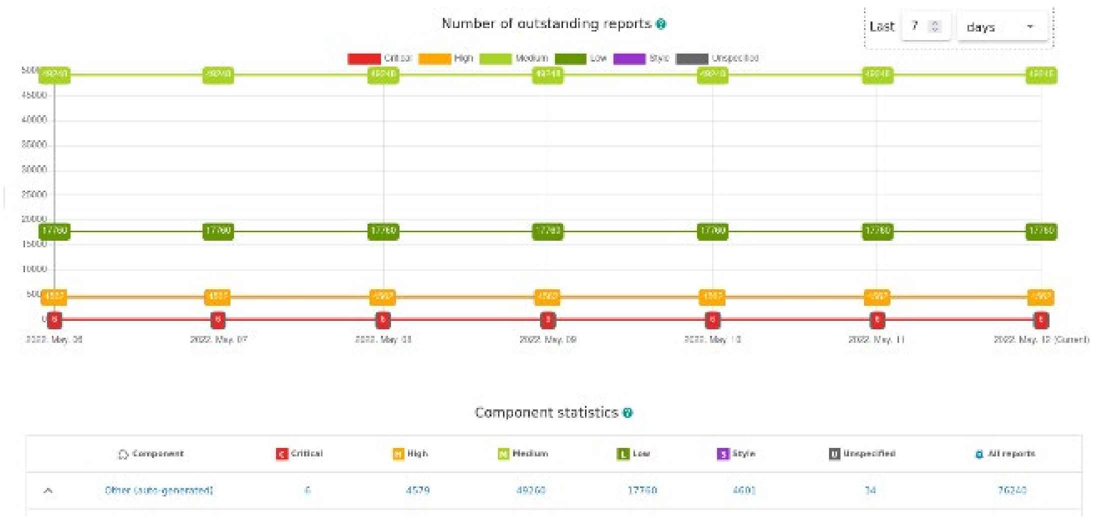


Figure 17. Overview of SonarQube [38].

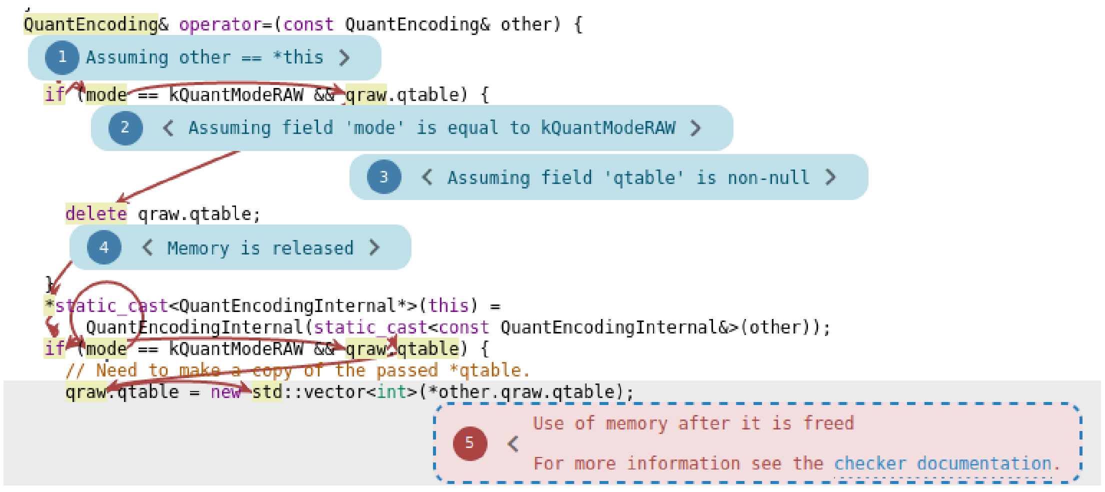
In a development process, developers commit or merge code to the DevOps platform through a Sonarlint IDE to receive instant feedback in the editor. In the second step, the CI tool is used to check out builds and run unit tests for visualization of results through an integrated SonarQube scanner. In the final step, SonarQube uploads the result analysis on the SonarQube server, providing feedback to developers through SonarQube Interface for visualization.

Different analysers cover different aspects of software quality. Hence the recommendation is to apply several tools. To collect analysis results in a common place, analyser frameworks such as CodeChecker are used. CodeChecker [111] is an open-source analysis framework primarily supplying a frontend to the *LLVM (low-level virtual machine) Clang* static analyser [112]. Additionally, it handles parsing and storing results from several other analysers targeting different languages. The analysis results are uploaded to a CodeChecker server, where warnings are presented in-line with the code for convenience. Some dashboard functionality is supported, showing the number of issues and review status.



Component	Critical	High	Medium	Low	Style	Unspecified	All reports
Other (auto-generated)	6	4579	49260	17760	4601	34	76240

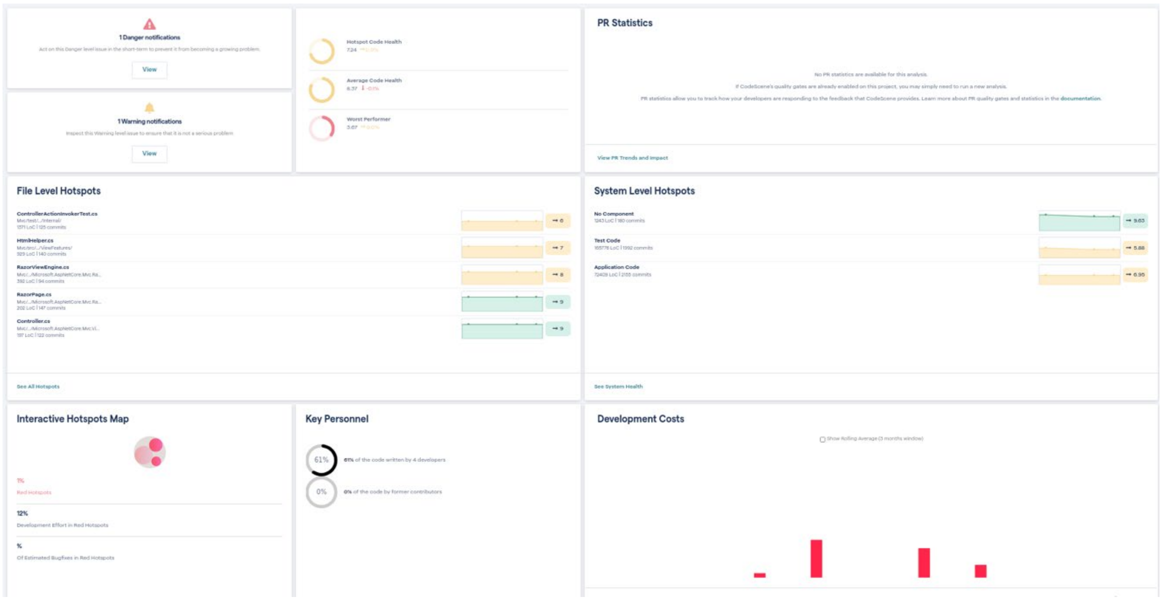
Figure 18. CodeChecker analysis statistics dashboard.



```
QuantEncoding& operator=(const QuantEncoding& other) {
  1 Assuming other == *this
  if (mode == kQuantModeRAW && qraw.qtable) {
    2 Assuming field 'mode' is equal to kQuantModeRAW
    3 Assuming field 'qtable' is non-null
    delete qraw.qtable;
    4 Memory is released
  }
  *static_cast<QuantEncodingInternal*>(this) =
      QuantEncodingInternal(static_cast<const QuantEncodingInternal&>(other));
  if (mode == kQuantModeRAW && qraw.qtable) {
    // Need to make a copy of the passed *qtable.
    qraw.qtable = new std::vector<int>(*other.qraw.qtable);
      5 Use of memory after it is freed
        For more information see the checker documentation.
```

Figure 19. Example of analysis report on CodeChecker server.

CodeScene [113] is a visualisation tool that gathers and presents information regarding technical debt. It scans a repository and extracts metrics related to code health based on review and metrics data (dependencies, complexity etc.). Additionally, it aggregates business-oriented information such as development cost estimation or key contributors. The results are presented in different dashboards that may be expanded based on interest. The visualisation highlights the hotspots found making them visible immediately. Finally, CodeScene presents mitigation recommendations to handle the issues found.



1 Danger notifications

Act on this Danger level issue in the short-term to prevent it from becoming a growing problem.

View

1 Warning notifications

Inspect this Warning level issue to ensure that it is not a serious problem

View

Hotspot Code Health
7.24

Average Code Health
8.37

Worst Performer
3.67

PR Statistics

No PR statistics are available for this analysis.

If CodeScene's quality gates are already enabled on this project, you may simply need to run a new analysis.

PR statistics allow you to track how your developers are responding to the feedback that CodeScene provides. Learn more about PR quality gates and statistics in the documentation.

View PR Trends and Impact

File Level Hotspots

ControllerActionInvokerTest.cs
Mvc/test/.../Internal/
1371 LoC | 125 commits
→ 6

HtmlHelper.cs
Mvc/src/.../ViewFeatures/
929 LoC | 140 commits
→ 7

RazorViewEngine.cs
Mvc/.../Microsoft.AspNetCore.Mvc.Ra...
392 LoC | 94 commits
→ 8

RazorPage.cs
Mvc/.../Microsoft.AspNetCore.Mvc.Ra...
202 LoC | 147 commits
→ 9

Controller.cs
Mvc/.../Microsoft.AspNetCore.Mvc.Vi...
197 LoC | 122 commits
→ 9

See All Hotspots

System Level Hotspots

No Component
1243 LoC | 180 commits
→ 9.63

Test Code
165776 LoC | 1992 commits
→ 5.88

Application Code
72409 LoC | 2155 commits
→ 6.95

See System Health

Interactive Hotspots Map

1%
Red Hotspots

12%
Development Effort in Red Hotspots

%
Of Estimated Bugfixes in Red Hotspots

Key Personnel

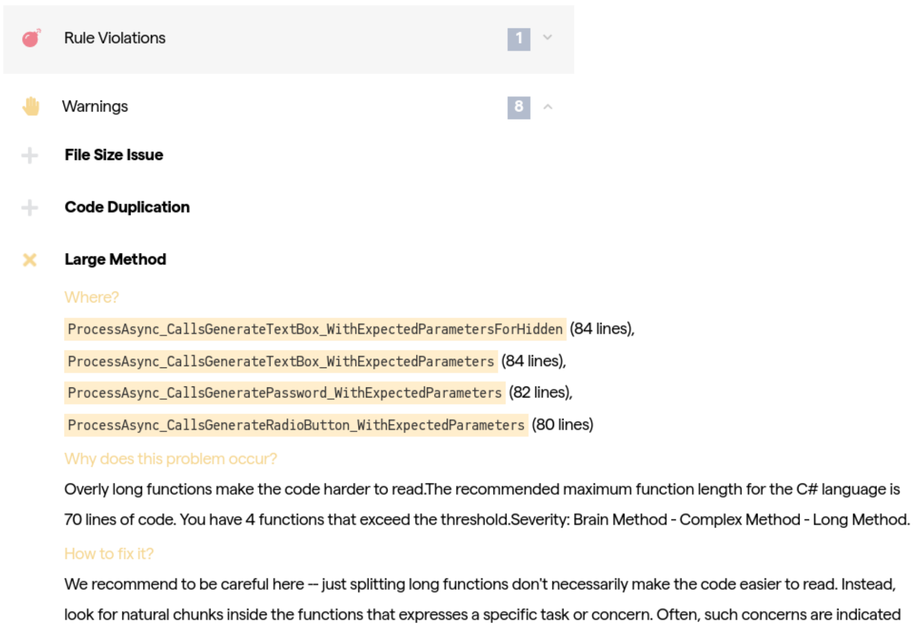
61% — 61% of the code written by 4 developers

0% — 0% of the code by former contributors

Development Costs

Show Rolling Average (3 months window)

Figure 20. CodeScene dashboard overview of code- and developer issue hotspots. Image captured from product showcase applied on ASP.NET MVC Project [114].



Rule Violations 1

Warnings 8

File Size Issue

Code Duplication

Large Method

Where?

`ProcessAsync_CallsGenerateTextBox_WithExpectedParametersForHidden` (84 lines),

`ProcessAsync_CallsGenerateTextBox_WithExpectedParameters` (84 lines),

`ProcessAsync_CallsGeneratePassword_WithExpectedParameters` (82 lines),

`ProcessAsync_CallsGenerateRadioButton_WithExpectedParameters` (80 lines)

Why does this problem occur?

Overly long functions make the code harder to read.The recommended maximum function length for the C# language is 70 lines of code. You have 4 functions that exceed the threshold.Severity: Brain Method - Complex Method - Long Method.

How to fix it?

We recommend to be careful here -- just splitting long functions don't necessarily make the code easier to read. Instead, look for natural chunks inside the functions that expresses a specific task or concern. Often, such concerns are indicated

Figure 21. Example of code quality warning and mitigation recommendation in CodeScene.

2.2.1. Software Metrics

Software metrics can be obtained from various sources, i.e., artifacts produced for and during the software development process [115].

One class of metrics is derived from the source code itself. The simplest form is the number of LOC. While this metric is easy to calculate, it only gives a very rough estimate of the code complexity. Therefore, many metrics exist which aim to increase the expressiveness. E.g., some methods weight the code lines in a certain way or take additional artifacts into account. The number of bugs per line of code is an example for this approach. Moreover code-based metrics can also be used to give recommendations for refactoring. This can be done by e.g., measure the coupling / cohesion of classes and packages.

Metrics can also be calculated by tracing the execution of the software. A simple representative of the category is the measurement of the average software execution time. Other parameters can be measured as well like the average storage consumption or the average request delay in the context of web applications. While these metrics can be collected during the normal execution of the program, it can also be desired to calculate them during the execution of tests. The measurement of test coverage is an example for this which is already considered a best practice for software development.

Apart from the software development artifacts, related processes yield valuable data for metrics, too. A widely used metric of this category is the Cycle Time. This is the average time elapsing from a feature request to the actual feature deployment. It therefore can be used to assess the effectiveness of the software development process.

2.2.2. Families of Metrics

A single metric can only be used to measure one very specific aspect of the software development process. While this can be useful for making concrete decisions, it may complicate the assessment of the overall process. To overcome this, different metrics are often grouped as family.

A popular metric family are the DORA (DevOps Research and Assessment) metrics [116] by The DevOps Research and Assessment team (hence the name). DORA metrics aim to assess the quality of the development process by measuring four different categories: Deployment frequency, lead time for changes, mean time to recovery and change failure rate.

The SPACE (Satisfaction, Performance, Activity, Communication, and Efficiency) metrics [117] aims to measure the productivity of software development. This is done by rating it in five different categories, namely satisfaction, performance, activity, communication, and efficiency.

2.2.3. Tools

The collection and calculation of metrics has been integrated into countless tools. These range from free and open-source tools over on-premise commercial tools to full software-as-a-service platforms. Most of these tools include methods to visualize the metrics, too. In the following, a non-representative list of commercial tools is given.

LinearB [118] is a web platform for the collection of DORA metrics. It collects them from various tools like code repositories, CI platforms and issue trackers. The metrics are presented using graphs.

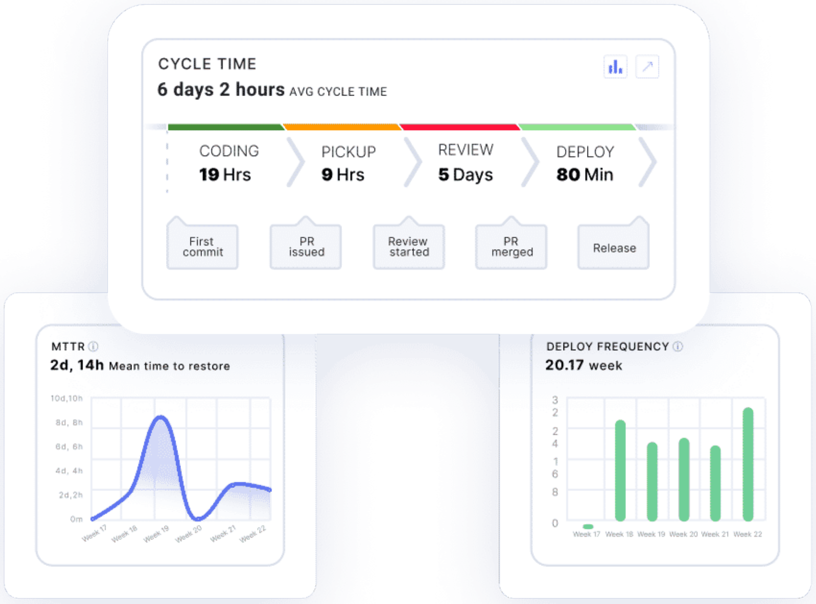


Figure 22. LinearB displays metrics from various tools and repositories.

Haystack [119] integrates with a Git-based development workflow to collect metrics. These can be visualized in a configurable dashboard as shown in Figure 23.

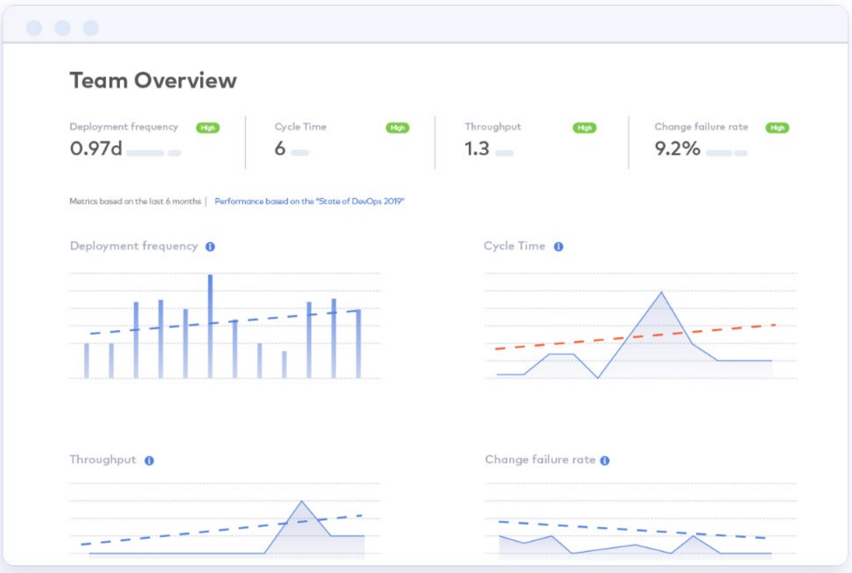

Team Overview
Deployment frequency High 0.97d
Cycle Time High 6
Throughput High 1.3
Change failure rate High 9.2%
Metrics based on the last 6 months | Performance based on the "State of DevOps 2019"
Deployment frequency
Cycle Time
Throughput
Change failure rate

Figure 23. Haystack displays Git data.

Swarmia is a web platform to collect metrics “aligned with DORA and SPACE metrics” [120].

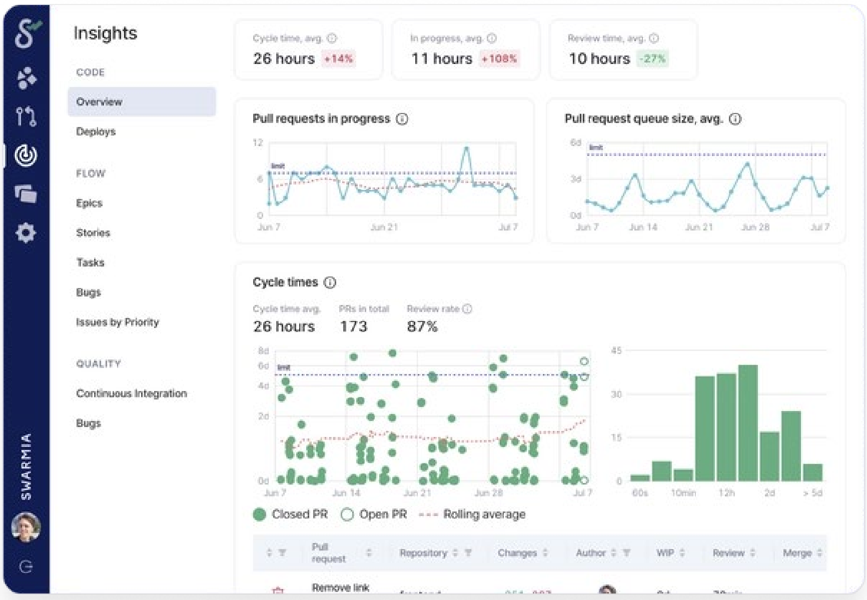
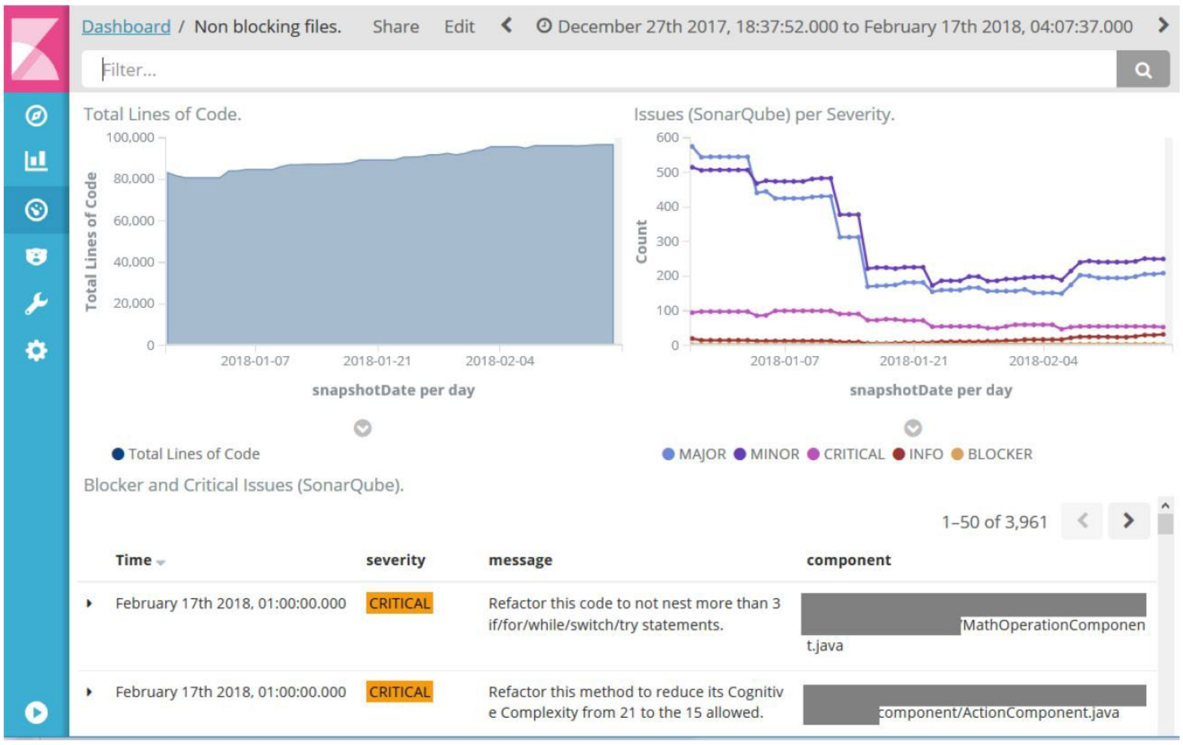


Figure 24. Swarmia shows metrics for the development pipeline.

The tool Q-Rapids [122] offers software analytics capabilities that integrate QMs to assess and improve software quality in the context of ASD. Its main functionalities are: (a) real-time gathering of various types of data related to the development and usage of a software system, which are the input for a QM; (b) real-time modelling of this data in terms of a QM in order to reason about aggregated quality-related strategic indicators; (c) presenting this QM and data to decision makers in a multi-dimensional and navigational dashboard for use during ASD events, such as sprint planning or daily stand-up meetings.

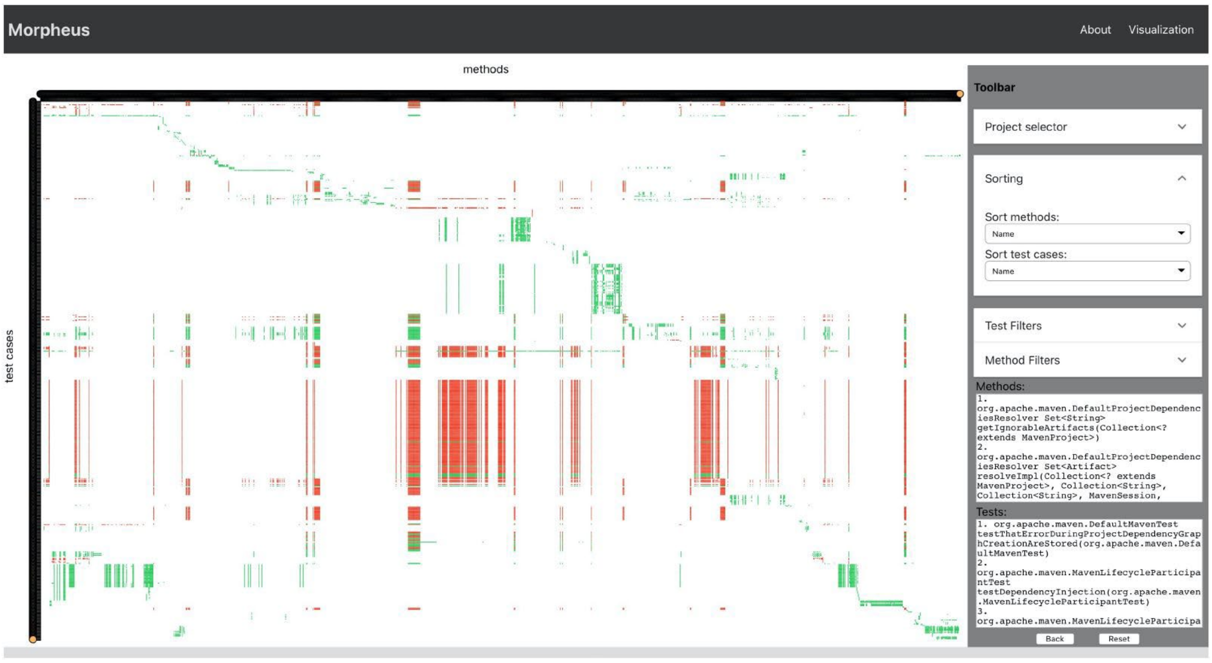


Blocker and Critical Issues (SonarQube).

1–50 of 3,961

Time	severity	message	component
February 17th 2018, 01:00:00.000	CRITICAL	Refactor this code to not nest more than 3 if/for/while/switch/try statements.	[redacted] MathOperationComponent.java
February 17th 2018, 01:00:00.000	CRITICAL	Refactor this method to reduce its Cognitive Complexity from 21 to the 15 allowed.	[redacted] component/ActionComponent.java

Figure 25. Q-Rapids tool with Raw Data Dashboard for blocking and critical files [122].



Sort methods: Name

Sort test cases: Name

Methods:

```
1.
org.apache.maven.DefaultProjectDependenc
iesResolver Set<String>
getIgnorableArtifacts(Collection<?
extends MavenProject>)
2.
org.apache.maven.DefaultProjectDependenc
iesResolver Set<Artifact>
resolveImpl(Collection<? extends
MavenProject>, Collection<String>,
Collection<String>, MavenSession,
```

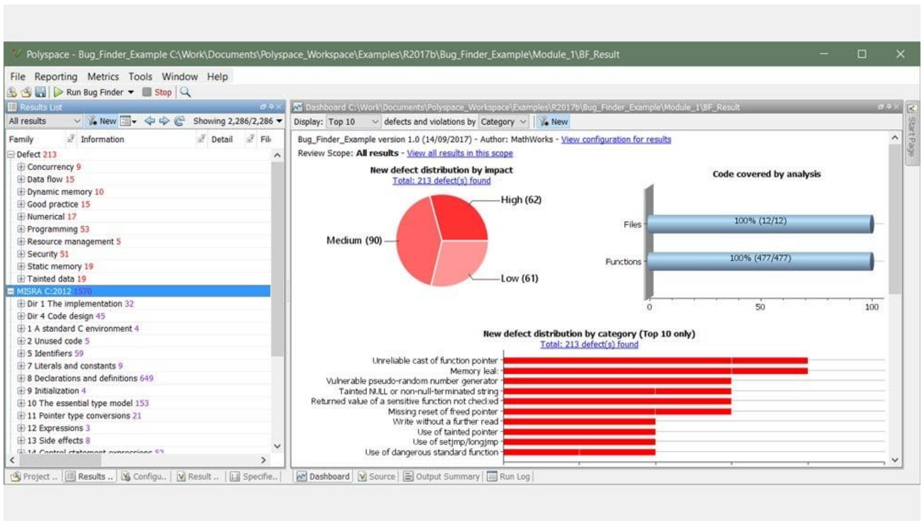
Tests:

```
1. org.apache.maven.DefaultMavenTest
testThatErrorDuringProjectDependencyGrap
hCreationAreStored(org.apache.maven.Defa
ultMavenTest)
2.
org.apache.maven.MavenLifecycleParticipa
ntTest
testDependencyInjection(org.apache.maven
.MavenLifecycleParticipantTest)
3.
org.apache.maven.MavenLifecycleParticipa
```

Figure 26. Morpheus Web Application UI [121].

Considering the sheer number of tools, the demand for the calculation of metrics and especially the visualization of those is huge. This is mainly caused by their ability to measure the quality of the software development process in a concise and quantifiable way. Consequently, expressive visualizations of those can be an essential guide for decision making.

MATHWORKS dashboard, to determine the code quality from Polyspace taken from [6], generates code metrics to measure and improve the quality of the source code and to compare results against quality thresholds.


Polyspace - Bug_Finder_Example C:\Work\Documents\Polyspace_Workspace\Examples\R2017b\Bug_Finder_Example\Module_1\BF_Result
File
Reporting
Metrics
Tools
Window
Help
Run Bug Finder
Stop
Results List
All results
New
Showing 2,286/2,286
Family
Information
Detail
Dashboard C:\Work\Documents\Polyspace_Workspace\Examples\R2017b\Bug_Finder_Example\Module_1\BF_Result
Display: Top 10
defects and violations by Category
New
Start Page
Project ..
Results ..
Configu..
Result ..
Specifie..
Dashboard
Source
Output Summary
Run Log

Bug_Finder_Example version 1.0 (14/09/2017) - Author: MathWorks - View configuration for results

Review Scope: **All results** - View all results in this scope

```
Defect 213
  Concurrency 9
  Data flow 15
  Dynamic memory 10
  Good practice 15
  Numerical 17
  Programming 53
  Resource management 5
  Security 51
  Static memory 19
  Tainted data 19
MISRA C:2012 1570
  Dir 1 The implementation 32
  Dir 4 Code design 45
  1 A standard C environment 4
  2 Unused code 5
  5 Identifiers 59
  7 Literals and constants 9
  8 Declarations and definitions 649
  9 Initialization 4
  10 The essential type model 153
  11 Pointer type conversions 21
  12 Expressions 3
  13 Side effects 8
```

New defect distribution by impact
Total: 213 defect(s) found
High (62)
Medium (90)
Low (61)
Code covered by analysis
Files
100% (12/12)
Functions
100% (477/477)
0
50
100
New defect distribution by category (Top 10 only)
Total: 213 defect(s) found
Unreliable cast of function pointer
Memory leak
Vulnerable pseudo-random number generator
Tainted NULL or non-null-terminated string
Returned value of a sensitive function not checked
Missing reset of freed pointer
Write without a further read
Use of tainted pointer
Use of setjmp/longjmp
Use of dangerous standard function

Figure 27. Dashboard in Polyspace Desktop User Interface.

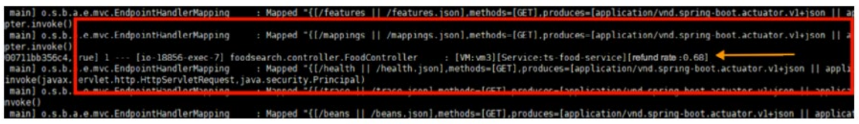
2.3. Visualizations Supporting Monitoring and Diagnostics Through Log Analysis

In debugging processes, debugging applications at three maturity levels are used with qualitative and quantitative analysis approaches. In qualitative analysis approach, for each fault condition, three developers are selected to debug applications at different maturity levels. The tools provided for different maturity levels are as follows [58].

- Basic Log Analysis. Developers use command line tools to retrieve and analyze logs.

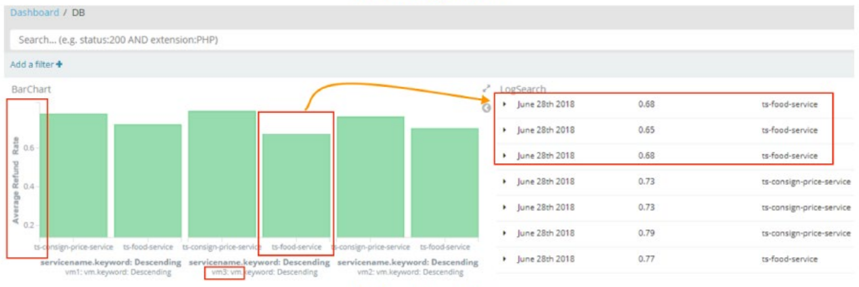
- Visual Log Analysis. Developers use the ELK stack [53] such as Logstash for log collection, Elasticsearch for log indexing and retrieval, and Kibana for visualization.
- Visual Trace Analysis. Developers use both the ELK stack and tools like Zipkin [52] for debugging.

Figure 28 presents sample snapshots of basic log analysis, visual log analysis, and visual trace analysis, respectively.



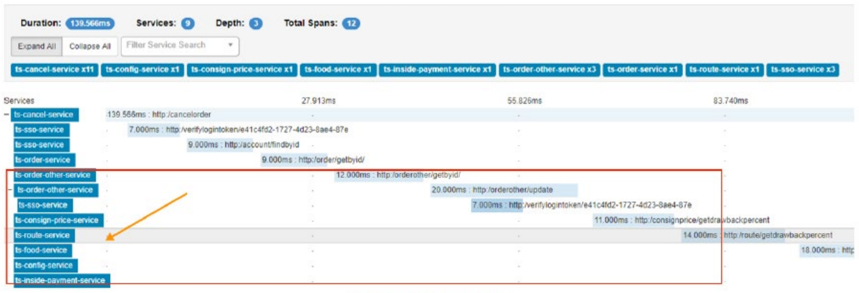
```
main] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/features || /features.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || a
pter.invoke()
main] o.s.b. .e.mvc.EndpointHandlerMapping     : Mapped "{[/mappings || /mappings.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || a
pter.invoke()
00711bb356c4, rue] 1 --- [io-18856-exec-7] foodsearch.controller.FoodController     : [VM:vm3][Service:ts-food-service][refund rate:0.68]
main] o.s.b. .e.mvc.EndpointHandlerMapping     : Mapped "{[/health || /health.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || appli
invoke(javax. ervlet.http.HttpServletRequest,java.security.Principal)
main] o.s.b. .e.mvc.EndpointHandlerMapping     : Mapped "{[/trace || /trace.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || applic
nvoke()
main] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/beans || /beans.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || applica
```

(a) Basic Log Analysis


Dashboard / DB
Search... (e.g. status:200 AND extension:PHP)
Add a filter +
BarChart
Average Refund Rate
0.6
0.4
0.2
ts-consign-price-service
ts-food-service
ts-consign-price-service
ts-food-service
ts-consign-price-service
ts-food-service
servicename.keyword: Descending
vm1: vm.keyword: Descending
servicename.keyword: Descending
vm3: vm.keyword: Descending
servicename.keyword: Descending
vm2: vm.keyword: Descending
LogSearch

June 28th 2018	0.68	ts-food-service
June 28th 2018	0.65	ts-food-service
June 28th 2018	0.68	ts-food-service
June 28th 2018	0.73	ts-consign-price-service
June 28th 2018	0.73	ts-consign-price-service
June 28th 2018	0.79	ts-consign-price-service
June 28th 2018	0.77	ts-food-service

(b) Visual Log Analysis



```
ts-cancel-service          139.566ms : http:/cancelorder
ts-sso-service             7.000ms : http:/verifylogintoken/e41c4fd2-1727-4d23-8ae4-87e
ts-sso-service             9.000ms : http:/account/findbyid
ts-order-service           9.000ms : http:/order/getbyid/
ts-order-other-service     12.000ms : http:/orderother/getbyid/
ts-order-other-service     20.000ms : http:/orderother/update
ts-sso-service             7.000ms : http:/verifylogintoken/e41c4fd2-1727-4d23-8ae4-87e
ts-consign-price-service   11.000ms : http:/consignprice/getdrawbackpercent
ts-route-service           14.000ms : http:/route/getdrawbackpercent
ts-food-service            18.000ms : http
ts-config-service
ts-inside-payment-service
```

(c) Visual Trace Analysis

Figure 28: Qualitative comparison of different maturity levels [58].

The quantitative analysis approach includes the entire debugging process and the time used for each step, in addition to the maturity levels in the qualitative approach [58].

Software infrastructures constantly generate log data to provide performance insight. Being able to visualize and monitor any log data in real time, such as syslog, SNMP traps, or Windows event logs, they help provide the insights needed for effective troubleshooting when problems arise and quickly uncover root causes of problems. However, these logs come in multiple formats from different sources, making their consolidation, analysis, and visualization difficult. The proliferation of distributed applications, the increasing amount of log data, the speed of incoming log data, and CI/CD applications that require multiple development environments (each with its own logs) make visualization difficult [96].

There are some commercial products such as Solarwind [90], ManageEngine[91] and open-source products such as ELK Stack, Fluentd [92], Graylog [93], Nagios [94].

As a result, visualization makes institutions more responsive and agile. The power of visualizations comes from people's ability to recognize visual patterns. It is difficult to spot a pattern among the raw log data, but it becomes easier when you can visualize that data. When an experienced DevOps engineer looks at the log visualization, they can easily see where the anomaly is, even if they have no prior knowledge of how the system is working or where the anomaly is coming from [95].

2.4. Visualizations Supporting System Properties

2.4.1. Performance & Resource Consumption

The following are the research that relate energy consumption and executed software, a relevant aspect if we take into account that the study by Pang et al. [62] shows that more than 80% of developers do not take into account energy consumption when developing software.

Jagroep et al. [61] proposes a method to compare the energy produced by different deltas of a software product. The method was validated through an empirical experiment in which they studied the energy consumption of DOCGEN 7.3 and DOCGEN 8.0 software which are presented in the following image highlighting in red the difference between them.

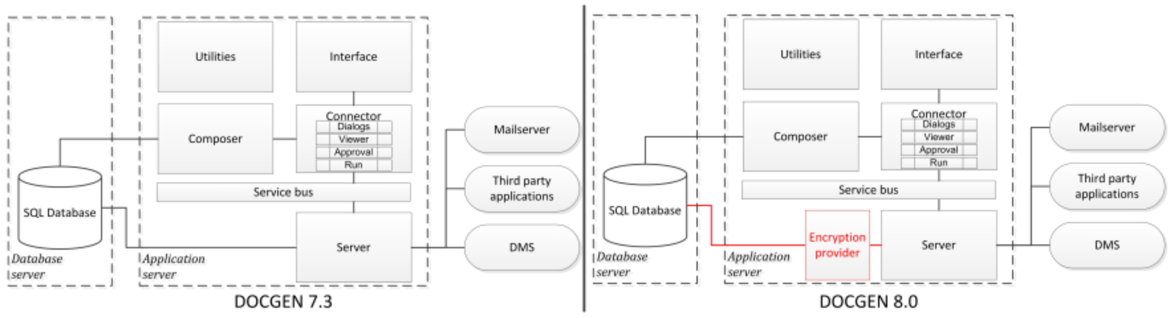
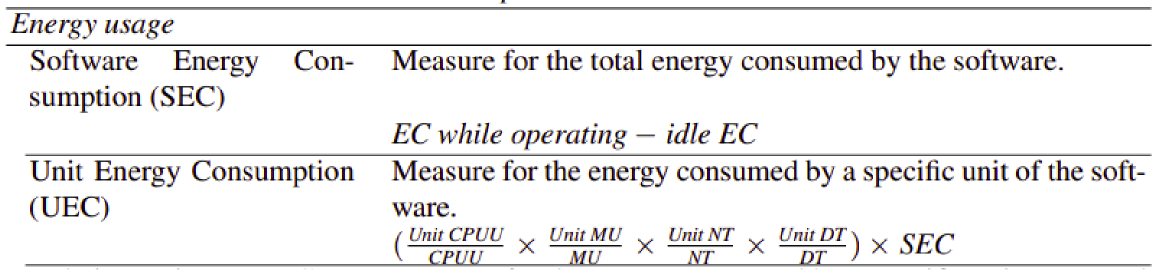


Figure 29. The change in red for the functional architectures for DOCGEN 7.3 and 8.0 [61].

In the experiment, the SEC and UEC metrics were measured as shown in the following image:



Energy usage	
Software Energy Consumption (SEC)	Measure for the total energy consumed by the software.
	$EC\ while\ operating - idle\ EC$
Unit Energy Consumption (UEC)	Measure for the energy consumed by a specific unit of the software.
	$\left(\frac{Unit\ CPUU}{CPUU} \times \frac{Unit\ MU}{MU} \times \frac{Unit\ NT}{NT} \times \frac{Unit\ DT}{DT}\right) \times SEC$

Figure 30. Energy consumption measures used.

The following hardware resources are monitored:

- Hard disk: disk bytes/sec, disk read bytes/sec, disk write bytes/sec
- Processor: % processor usage
- Memory: private bytes, working set, private working set
- Network: bytes total/sec, bytes sent/sec, bytes received/sec
- IO: IO data (bytes/sec), IO read (bytes/sec), IO write (bytes/sec)

The authors state that the method detailed in this research can be replicated by companies to measure electricity consumption in different deltas of a product. Among the relevant contributions of this research is a regression model that allows predicting electricity consumption from system performance and consumption.

Chowdhury and Hindle [63] present GreenOracle, a model that allows estimating the energy consumption (jules) of the application. The model is based on big data taking as a data source energy measurement of previous applications to estimate the energy consumption of any application, the model also takes the dynamic traces of system calls and CPU utilization. The proposed model does not require any power measurement of the application under test, so it can save developers time by mitigating the complexity of measuring the actual power consumption using hardware instruments. When a developer modifies the source code, he can rerun the energy model for the new version and find out if the energy consumption exceeds his predefined threshold.

GreenAdvisor [64] provides information about the change in energy consumption between commits and indicates to the developer which part of the code may be causing the change. The GreenAdvisor tool uses the application's system call profile to warn developers about possible changes in the application's energy consumption profile by using general rules introduced defined in previous research. The authors are aware that their results may be biased because the evaluation was performed with students who were assigned the same project and most of them were inexperienced developers.

Johann et al. [98] presents a generic metric for measuring software and a method for applying it in a software engineering process. This method uses a white-box measurement approach in which by having access to the code it is possible to determine where in the software there is potential for energy savings, here the source code instrumentation technique was used, which is often used in the context of software profiling, e.g., for runtime analysis.

They defined the metric as:

$$EnergyEfficiency = \frac{UsefulWorkDone}{UsedEnergy}$$

The following image presents the measurement environment used:

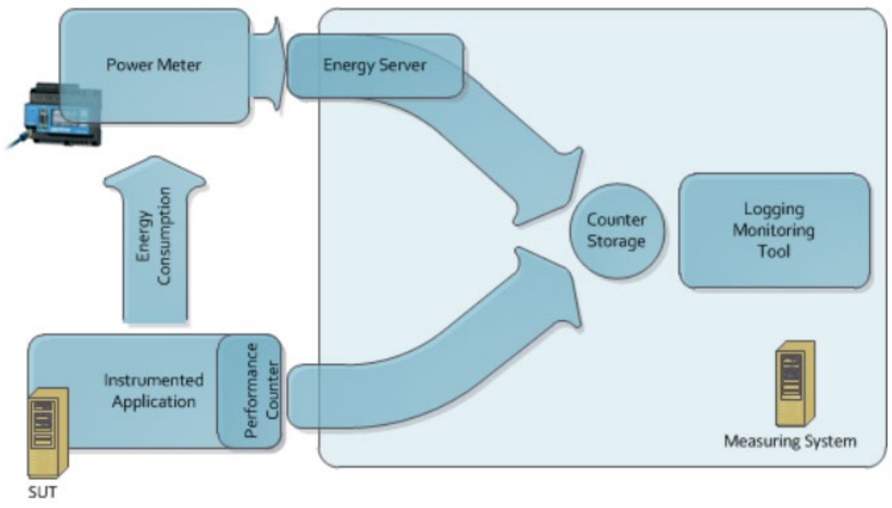


Figure 31. The measurement environment used in [98].

the authors describe the environment as follows “*The system under test runs an instrumented application and is connected to a power meter. The power meter is connected to the measuring system, where an Energy Server converts the values to performance counters. The instrumented application also writes performance counters. These can be read, e.g. by logging or monitoring tools to analyze the software regarding its energy consumption. To accomplish the instrumentation the set-up builds on an API provided by Intel to perform the source code instrumentation. Using this set-up, existing source code can be expanded by counters. These numerical values can be used e.g. for counting loop cycles, marking entry and exit points of code fragments, counting the amount of how often parts of software are frequented, etc.*”

The authors conclude that the procedure shown in this article is feasible for research but too complex for daily use, so they propose to implement this method in a software development project in the future.

Muhammad Afaq [99] presents a monitoring framework for virtual machines, the following image shows the proposed architecture, which is composed of three layers: configuration layer, monitoring layer and visualization layer.

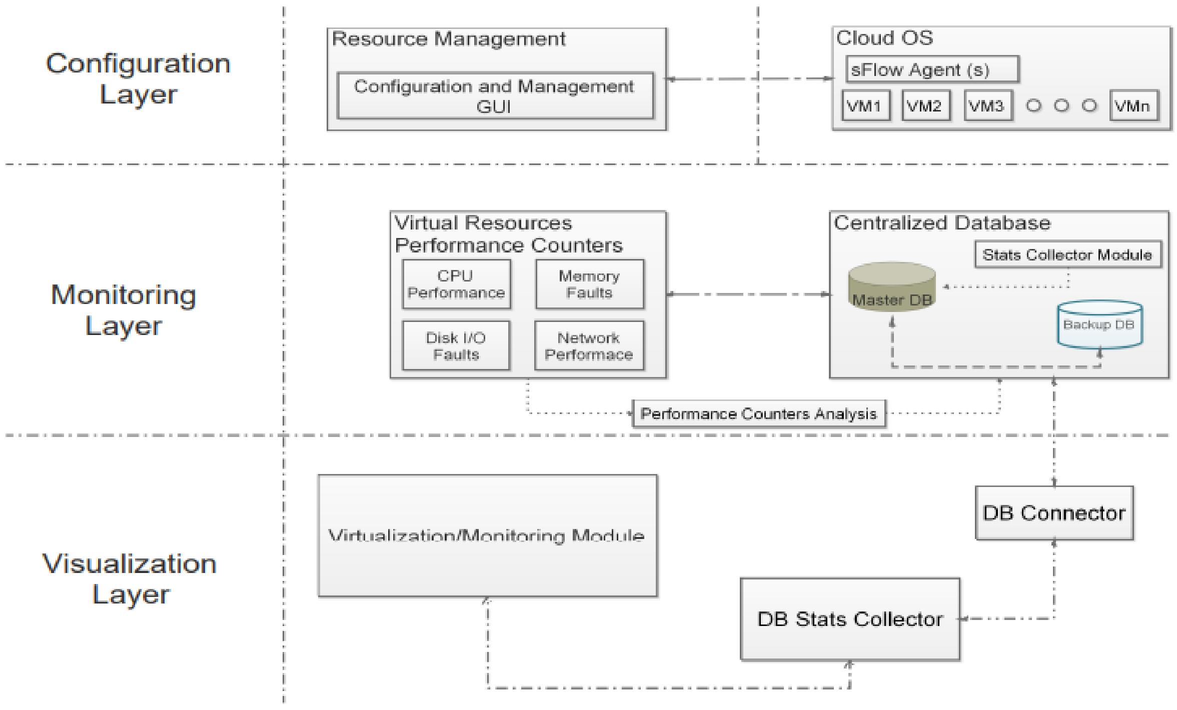


Figure 32. The system architecture proposed in [99].

The monitoring layer consists of a centralized database in which the performance counters collected by the Host sFlow agent are stored. The visualization layer consists of a visualization module that provides a graphical representation of the performance counters to the administrator. The visualization layer consists of a visualization module that provides a graphical representation of the performance counters to the administrator. The authors describe the operation as follows: "The Host sFlow agent collects the performance counters and sends them to the monitoring layer. In addition, the performance counters are sent to the visualization layer, where they are displayed graphically. This can greatly help administrators monitor and manage infrastructure resources to improve overall system performance." The following graph shows an example of visualization of the graphs presented to the user; in this case the graph represents network performance.

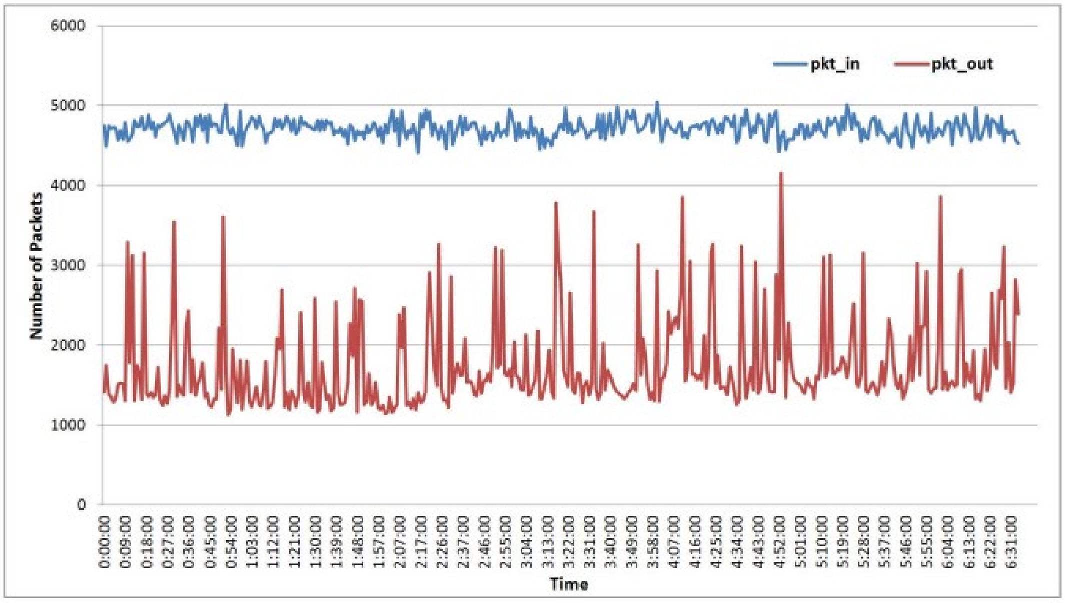
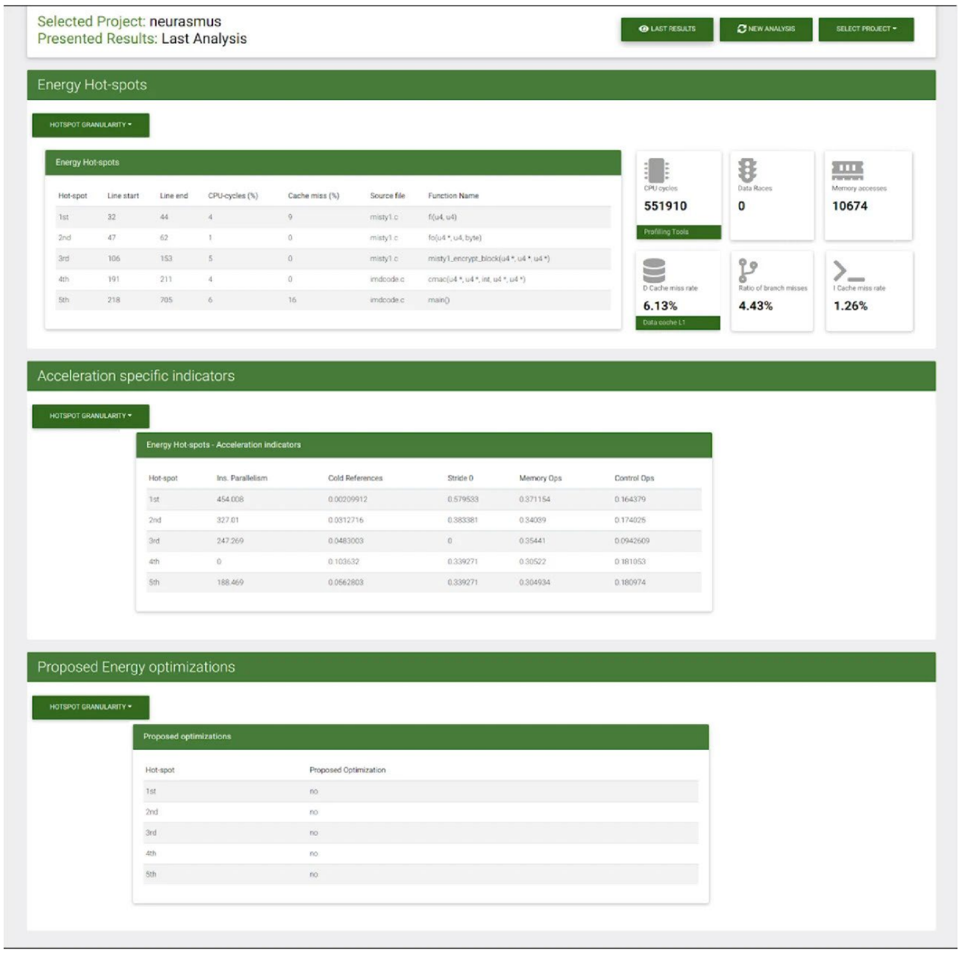


Figure 33. Performance of network resources (from [99]).

The SDK4ED Platform [16] enables the monitoring and optimization of important quality attributes, such as Energy Consumption, of software applications, with emphasis on embedded software. The

Energy Optimization toolbox provides two main functionalities, Consumption Analysis and Estimation & Optimization, which have been implemented in the form of individual web services.



Selected Project: neurasmus
Presented Results: Last Analysis

Energy Hot-spots

Energy Hot-spots

Hot-spot	Line start	Line end	CPU-cycles (%)	Cache miss (%)	Source file	Function Name
1st	32	44	4	9	misty1.c	f(u4, u4)
2nd	47	62	1	0	misty1.c	fo(u4 *, u4, byte)
3rd	106	153	5	0	misty1.c	misty1_encrypt_block(u4 *, u4 *, u4 *)
4th	191	211	4	0	imdcode.c	cmac(u4 *, u4 *, int, u4 *, u4 *)
5th	218	705	6	16	imdcode.c	main()

CPU cycles	551910
Data Races	0
Memory accesses	10674
D Cache miss rate	6.13%
Ratio of branch misses	4.43%
I Cache miss rate	1.26%

Acceleration specific indicators

Energy Hot-spots - Acceleration indicators

Hot-spot	Ins. Parallelism	Cold References	Stride 0	Memory Ops	Control Ops
1st	454.008	0.00209912	0.579533	0.371154	0.164379
2nd	327.01	0.0312716	0.383381	0.34039	0.174025
3rd	247.269	0.0483003	0	0.35441	0.0942609
4th	0	0.103632	0.339271	0.30522	0.181053
5th	188.469	0.0562803	0.339271	0.304934	0.180974

Proposed Energy optimizations

Proposed optimizations

Hot-spot	Proposed Optimization
1st	no
2nd	no
3rd	no
4th	no
5th	no

Figure 34. The front-end page of the Energy Optimization toolbox of the SDK4ED Platform [16].

2.4.2. Security & Privacy

During the development process, various vulnerabilities and weaknesses may be introduced in a software system. Oversights in cryptography, use of insecure components or protocols, insufficient logging, and monitoring, and so forth could enable attackers to steal user data or cause system instability [39]. Security and privacy visualization tools seek to provide an efficient and effective way of exploring and analysing these issues. This section presents an overview of existing works in the field of visualization of security and privacy information.

Since security and privacy are two different properties of a system, this section is subdivided into two subsections: security visualization and privacy visualization. Security is a property of a system that protects the system from outside attacks, while privacy refers to the state of having control over own data [89]. In this view, confidentiality of data or privacy protection comes under security; but the transparency regarding data collection and the ability of the data providers to control their personal data is privacy. Realizing this distinction, these properties are addressed separately.

Security visualization

To ensure security vulnerabilities are caught and handled, it is common for development teams to use vulnerability scanners within the toolchain. While these tools help address security concerns within the system, they are known to have usability issues [18, 19, 22]. For instance, most of these tools lack a proper reporting system and often produce output in a textual format (e.g., OWASP ZAP [37], Burp Suite [40]). Advanced tools like the SonarQube [38] provide a graphical interface, but the features are not adequate and lack functionalities like comparison of records across multiple scans, and search and filter capabilities, among others [19]. Moreover, a single toolchain can use multiple static and dynamic code analysers [21]. Thus, eliminating false-positive and irrelevant results from a large volume of scan results can become challenging and can overwhelm developers or security analysts [22]. A visual analytics tool that aggregates results from these tools to provide an interface that facilitates effective navigation and examination of vulnerabilities is a well-studied problem.

Goodall et al. [18] present a system that integrates outputs from multiple security analysis tools into a single interactive visual environment. The system provides an overview of detected vulnerabilities in terms of individual files in which these vulnerabilities were present. This aggregation of vulnerabilities into files allows users to triage many vulnerabilities detected by multiple tools. Authors argue that the file-based view is more effective because developers tend to be more familiar with the code files, they maintain rather than the vulnerability detected within the codes. To demonstrate the concept, authors developed a prototype and used it to visualize vulnerabilities in the Apache Tomcat repository [41]. Three software analysis tools were used to find around 34,000 security vulnerabilities in the repo. The prototype system then provided an overview of these vulnerabilities organized by a total of about 15,000 files (Figure 35).

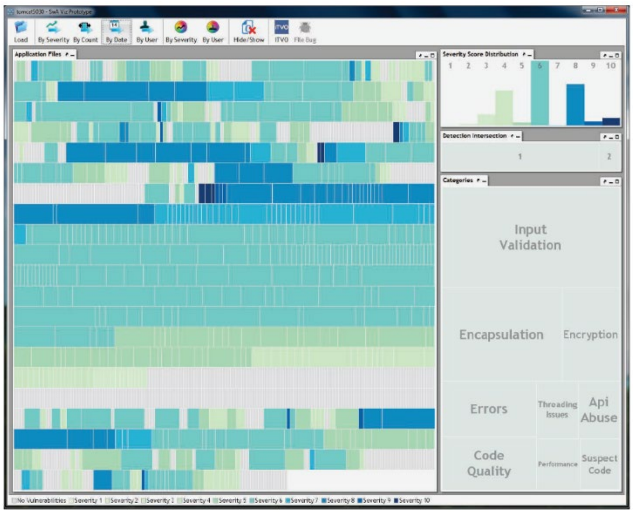
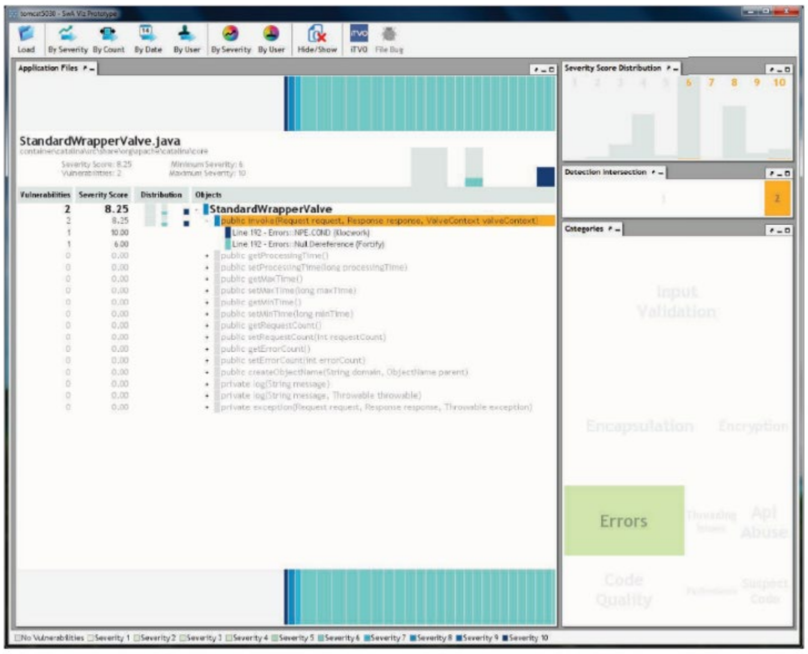


Figure 35. Security vulnerabilities in Apache Tomcat source visualized [18].

In the figure, on the left is the main display in which each block represents an individual file. The colour of a block signifies the average severity while the width of the block represents the number of vulnerabilities detected within the file. The bars can be sorted in accordance with severity, vulnerability count, creation date, or by users (who last committed the code). These sorting options help to customize the view as per priority. The right side of the UI contains three widgets: a severity distribution histogram that depicts vulnerability count per severity level; a detection intersection that shows the number of common vulnerabilities detected by multiple tools; and a Treemap diagram that categorizes vulnerabilities based on their type. These widgets can be used to filter the data on the main display on the left. Moreover, each bar in the main display can be selected to view a particular file in detail. The detail view (Figure 36) provides a tree-table of classes and methods in the source file. The vulnerabilities along with the severity colour coding are depicted with the corresponding line numbers. This enables a source code file to be reviewed separately to conclude if a vulnerability is true-positive.



StandardWrapperValve.java
container\catalina\src\share\org\apache\catalina\core

Severity Score: 8.25 Minimum Severity: 6
Vulnerabilities: 2 Maximum Severity: 10

Vulnerabilities	Severity Score	Distribution	Objects
2	8.25		StandardWrapperValve
2	8.25		public invoke(Request request, Response response, ValveContext valveContext)
1	10.00		Line 192 - Errors::NPE.COND (Klocwork)
1	6.00		Line 192 - Errors::Null Dereference (Fortify)
0	0.00		public getProcessingTime()
0	0.00		public setProcessingTime(long processingTime)
0	0.00		public getMaxTime()
0	0.00		public setMaxTime(long maxTime)
0	0.00		public getMinTime()
0	0.00		public setMinTime(long minTime)
0	0.00		public getRequestCount()
0	0.00		public setRequestCount(int requestCount)
0	0.00		public getErrorCount()
0	0.00		public setErrorCount(int errorCount)
0	0.00		public createObjectName(String domain, ObjectName parent)
0	0.00		private log(String message)
0	0.00		private log(String message, Throwable throwable)
0	0.00		private exception(Request request, Response response, Throwable exception)

Figure 36. Detailed view showing methods and detected vulnerability within a single file [18].

The system has a few shortcomings, for instance, the detail view shows only line number of the possible vulnerable code but does not show the code itself; and the vulnerabilities do not have CEW or CVE categorization.

[19] presents a system to visualize web application vulnerabilities. In the paper, authors propose HWAS-V, a dashboard tool to visualize output from DAST tools. The dashboard incorporates about 50 different metrics to visualize the security status of a web application. These include:

- Base metrics which include the data available from DAST tools like the scan rules, number of vulnerabilities/alerts generated, number of URLs scanned, number of URLs processed, number of URLs with an alert, etc.
- Metrics based on maintenance activities. For instance, set of alerts/vulnerabilities fixed, bug fixing related tasks, the set of alerts/vulnerabilities fixed due to new developments, and the total time to fix vulnerabilities.
- Metrics that reflect application properties like application size (number of lines of codes, modules, and web pages at a given time), deployment structure (geographical server location and base URL), and associated vulnerabilities like vulnerabilities per LOC, per module, and per web page).
- Metrics based on classification of vulnerabilities according to the severity of the alerts, sensitive information risk, and business impact.
- Metrics based on vulnerabilities detected or prevented by the external protection system implemented by the web application.

Furthermore, the dashboard also associates scan rules with security standards like OWASP, WASC, and CWE standards.

The data structure used to compute these metrics is based on the data attributes commonly present in the result of web security analysis tools. Thus, the tool is generic and can be applied for wide variety of DAST tools available. However, the data structure only supports data collected in fixed intervals and does not support continuous data. The visualization mechanism in Figure 37 illustrates different data processing stages.

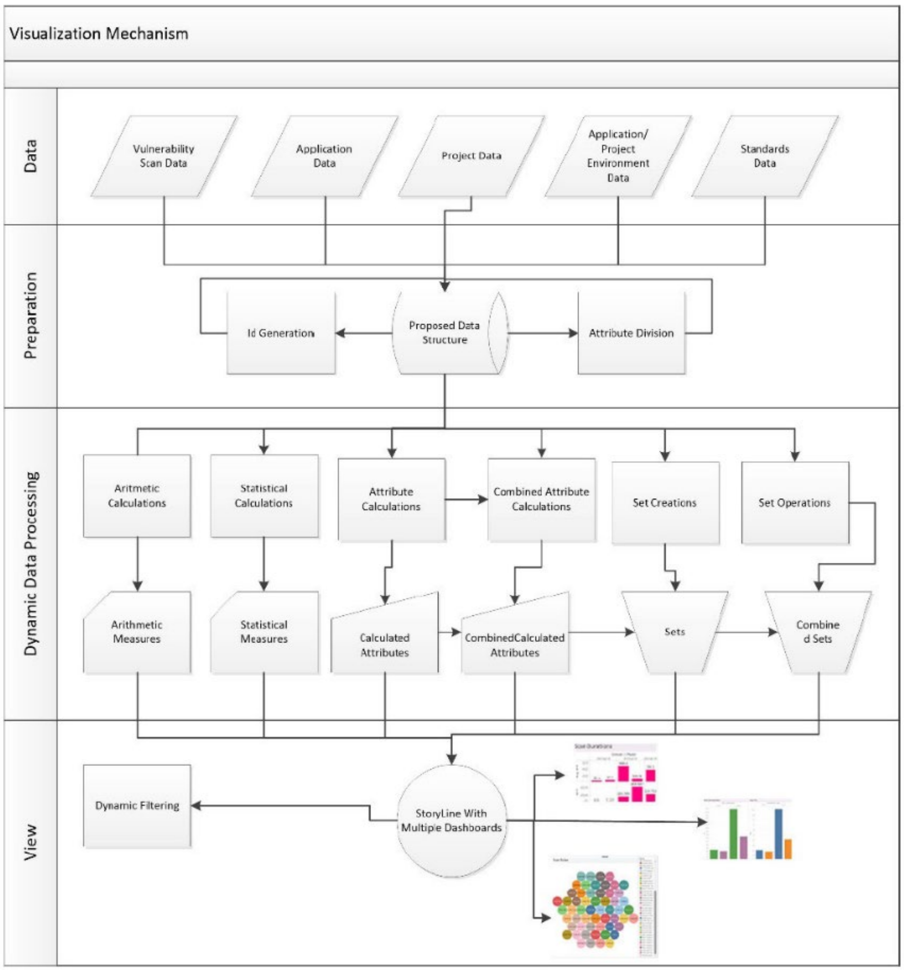
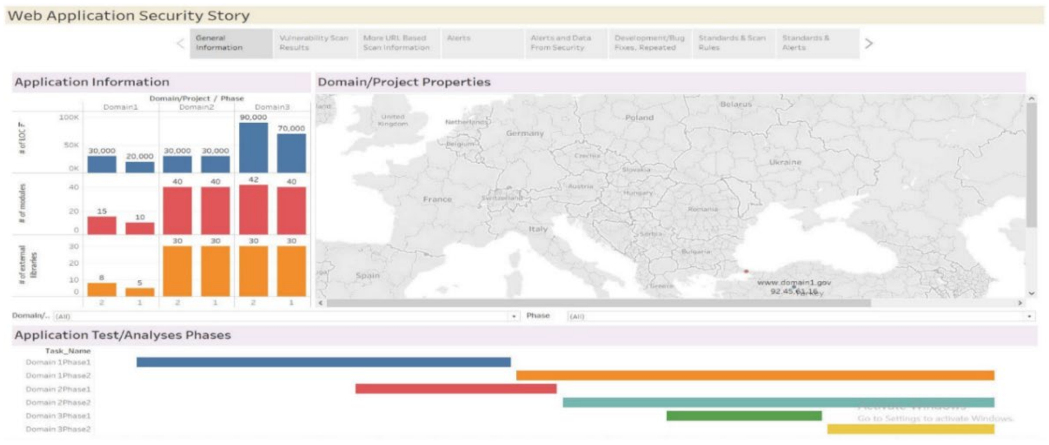


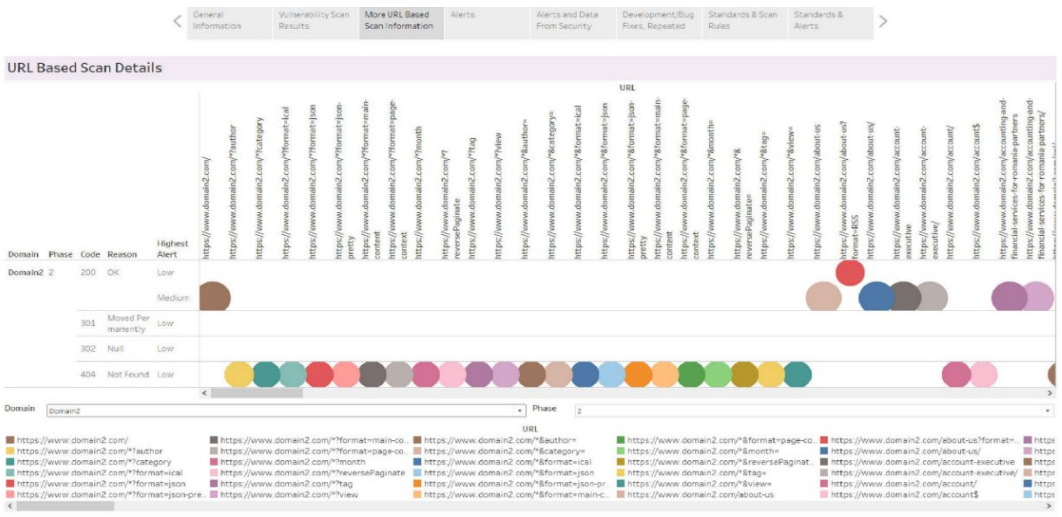
Figure 37. HWAS-V visualization mechanism [19].

Authors present a prototype developed using Tableau [26]. The system constitutes of multiple dashboards organized in tabs. Each dashboard contains metrics that reflect the specific purpose of that dashboard view. For instance, the "General Information" dashboard in Figure 38 (a) provides a top-level view which includes the web server location, application related information, and overview of earlier testing efforts. Each tab is called a "Story", and the naming of each story defines what the corresponding dashboard depicts. The logical grouping of metrics into multiple dashboards not only allows visualization of large number of metrics in an organized manner but also makes the system usable for people in different roles like executives, managers, developers, security analysts, security auditors, and system/ network administrators. For example, for a developer, the "URL based Scan Details" dashboard might be more relevant (Figure 38 (b)), whereas for an executive, "General Information" (Figure 38 (a)) might be more relevant.

Moreover, as can be seen from Figure 38, the system supports visualization of multiple projects and phases in a single view. This enables users to compare analysis results across multiple projects and phases.


Web Application Security Story
General Information
Vulnerability Scan Results
More URL Based Scan Information
Alerts
Alerts and Data From Security
Development/Bug Fixes, Repeated
Standards & Scan Rules
Standards & Alerts
Application Information
Domain/Project Properties
Application Test/Analyses Phases

(a) General information dashboard



(b) URL based scan details dashboard

Figure 38. Various dashboard views from the HWAS-V system. The data visualized are from the scan results and alerts generated by OWASP ZAP when applied on three independent web applications. Each "domain/ project" corresponds to an individual web application subjected to security analysis (three in this case). A "phase" represents the duration between successive scans; each new analysis starts a new phase [19].

In [20], S. Chiasson et al. provide recommendations for improving security interface designs. Although the work mostly focuses on the development of an intuitive UI for security management systems, the concepts are transferable to the design of security visualization tools as well. For instance, the concept of social navigation [27] where the UI provides elements that enables a user to communicate with other users for issue diagnosis and resolution. In this regard, a history of past events integrated in the dashboard can help users compare the current event with previous ones. The system proposed by [19] somewhat integrates these social aspects by presenting the history of actions, usage patterns, and alerts from other projects in a single dashboard view which allows users to compare events across multiple projects.

Similarly, the concept of EID (Ecological Interface Design) [28] is discussed. The design involves presenting as much information as possible to the users, but in different layers of abstraction. The top-most layer provides an overview of the system, and the subsequent layers gradually go into more detail. This type of hierarchical representation has several advantages, for instance, when a problem occurs, users can quickly move up the level to see the overall impact of the issue or move downward to further analyze the cause of the issue. Designs in [18] and [19] follow the EID philosophy as the proposed dashboards contain both a general overview along with an option to view metrics in a more detailed view.

Further, authors also advocate persuasive design technology [29] which incorporates design choices like sending messages when a new vulnerability is detected or reminders for initiating scans or highlighting severe issues, etc. Such feedbacks promote credibility of the users towards the system as the users become more confident that the system is actively working.

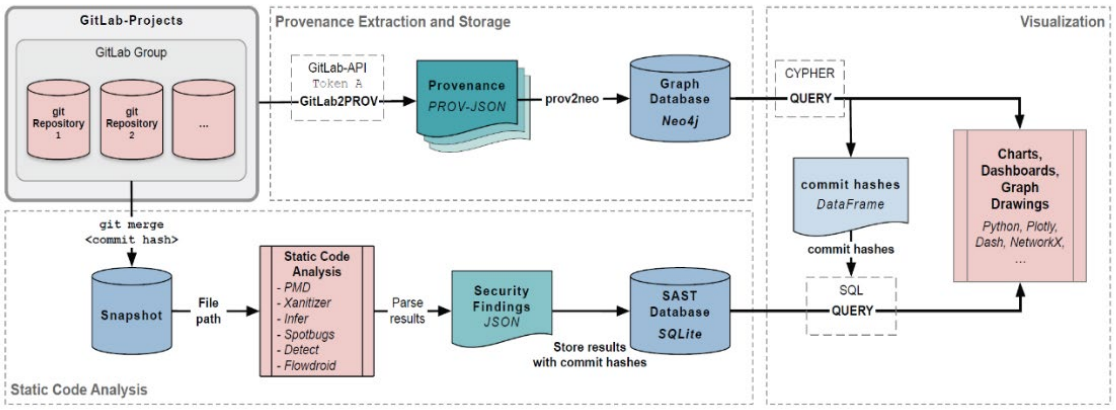


Figure 39. Workflow depicting provenance-based code analysis and visualization [21].

A. Schreiber [21] present a visualization system that combines provenance [30] information captured from the codebase with the results from multiple SAST tools. The integration of provenance information in the workflow enables users to isolate events of interest in the software development process and then analyse the results of static code analysis at the times of those events. The provenance data is collected by mining git repositories of the selected projects and is stored as property graphs in a graph database. The data model constitutes of activities (commits, issue changes, releases), entities (source code files, issues), and agents (developers, tester, and users). Scan results from multiple SAST tools are aggregated and stored in a separate database.

Data visualization involves submitting provenance queries (for instance, getting all commits that change a particular file or commits from a particular developer, or a group of developers) to the graph database. The query results are then used to filter the corresponding static code analysis results from the SAST database. Finally, the results are visualized in the dashboard. Figure 40 illustrates the workflow adopted for code analysis, data storage and data visualization.

As a case study, the authors implemented their approach on a mobile app called Luca [33]. The app code is public and is hosted in Gitlab in 8 repositories. Figure 40 displays the web-based interactive dashboard showing results for the Luca app. The prototype dashboard was developed in Python using the Dash [34] framework and Plotly library [35]. As can be seen from Figure 6, the dashboard comprises of three sections: the "Entities Timeline" shows the Gitlab events (provenance activities) as a timeline with bars; the "Static Analysis Warnings" is a line chart which shows the trend of the number of security-related warnings generated by the SAST tools over time; and the "CWE Breakdown" is an Icicle chart [36] which categorizes the SAST results in accordance to the CWE vulnerabilities. The Icicle chart is hierarchical; the first level is the project, the second level is the CWE class, and the third level represents the affected source code files. The size of the blocks represents the frequency of a warning, i.e., repeated warnings occupy a larger area. The colour, on the other hand, represents the severity of the warnings. Moreover, users can limit the timeline to specific periods in the Entities Timeline, the "Static Analysis Warnings" and "CWE Breakdown" adjust automatically to the selected timeline. The interface also provides functionalities like zoom, data comparison, selection, cross-filtering, and provenance query selection. These interactive capabilities allow for more in-depth analytics.

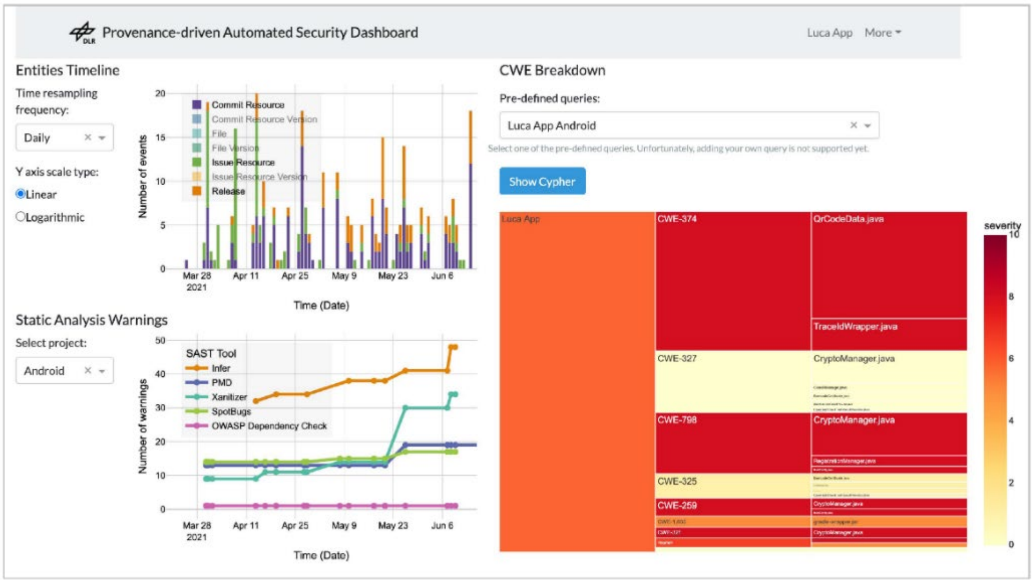


Figure 40. Interactive visualization dashboard which provides an overview of security analysis results from multiple SAST tools based on the provenance information of the code base [21].

In [22], S. L. Reynolds et al. argue that the visualization of code analysis should emphasize a user-centric design that prioritizes tasks and needs of the target users. Based on a preliminary study, authors determine that the visual analysis system should have features such as a dashboard showing overall vulnerabilities, options to search and filter, view details of a vulnerability on demand, and the ability to compare vulnerabilities of two or more versions of an application. Following these requirements, authors present a prototype in the form of a web application that visualizes the scan results from VUSC [31]. VUSC is a static code analyser that processes binary of Android apps.

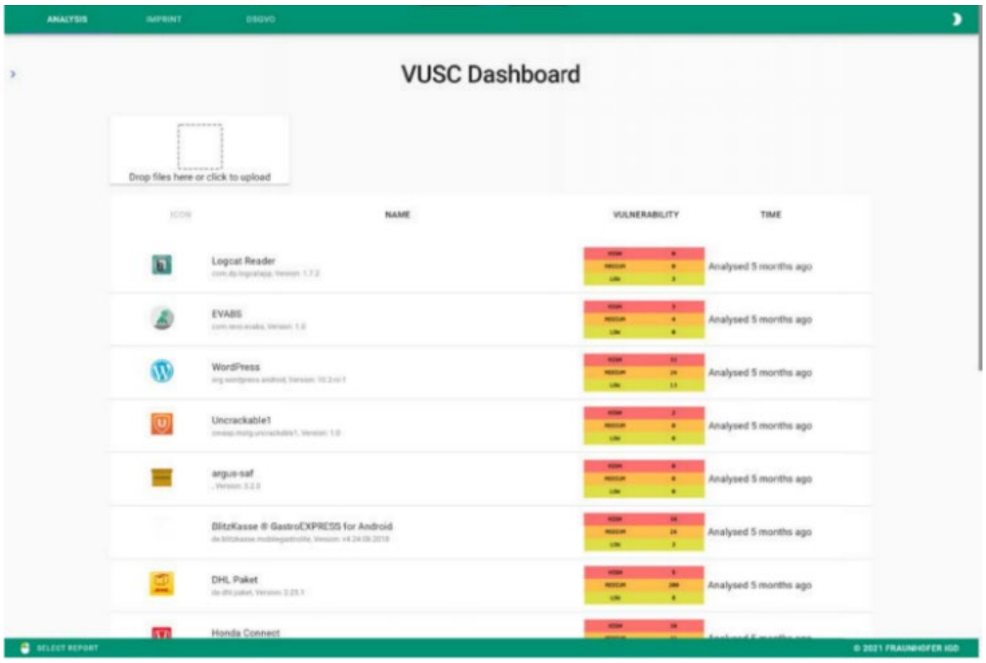
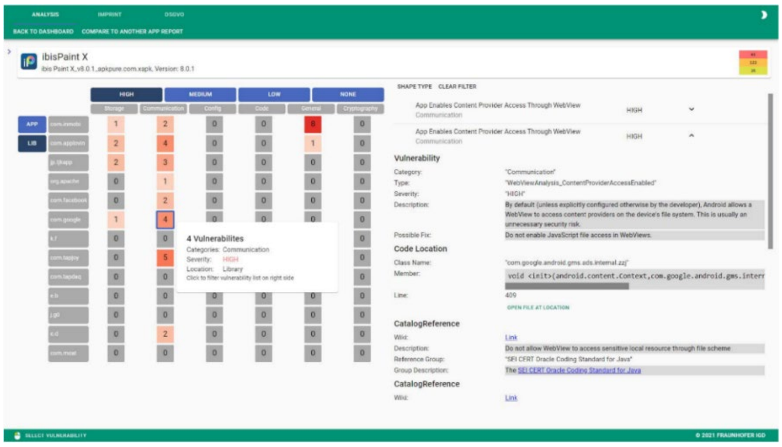


Figure 41. A dashboard view that shows list of Android apps scanned by VUSC analyser [22].

The web application consists of three views, the starting page (Figure 41) displays a list of all scanned applications. Details like the date of last scan and total vulnerability count categorized in accordance with their severity are displayed alongside the name of each listed app. Further, users can drag and drop the binary of an app to queue them for scan and visualize the results.



App Enables Content Provider Access Through WebView — Communication — HIGH

App Enables Content Provider Access Through WebView — Communication — HIGH

Vulnerability

Category:	"Communication"
Type:	"WebViewAnalysis_ContentProviderAccessEnabled"
Severity:	"HIGH"
Description:	By default (unless explicitly configured otherwise by the developer), Android allows a WebView to access content providers on the device's file system. This is usually an unnecessary security risk.
Possible Fix:	Do not enable JavaScript file access in WebViews.

Code Location

Class Name:	"com.google.android.gms.ads.internal.zzj"
Member:	`void <init>(android.content.Context,com.google.android.gms.interr`
Line:	409

CatalogReference

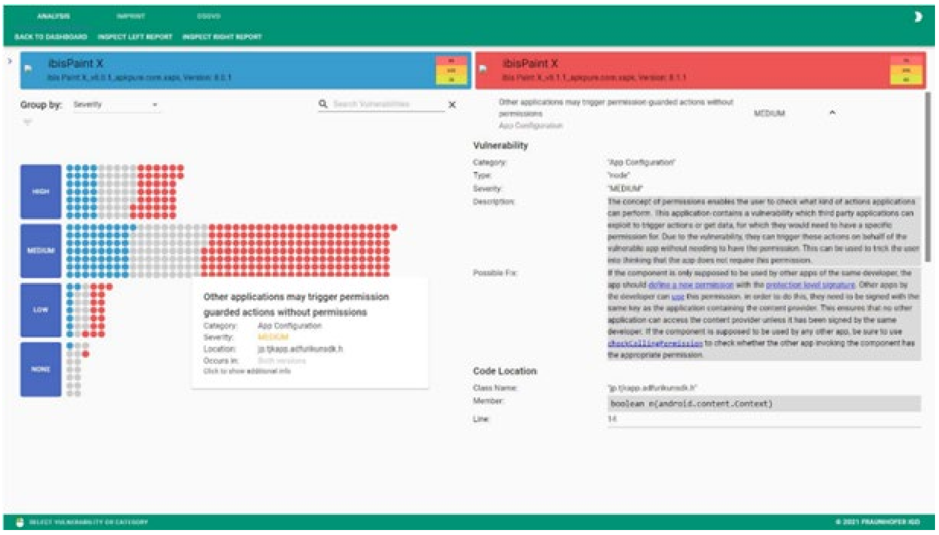
Wiki:	Link
Description:	Do not allow WebView to access sensitive local resource through file scheme
Reference Group:	"SEI CERT Oracle Coding Standard for Java"
Group Description:	The SEI CERT Oracle Coding Standard for Java

CatalogReference

Wiki:	Link

Figure 42. The Report Overview page which presents an overview of detected vulnerabilities in a scanned application [22].

Selection of a scanned app from the dashboard opens a new page (Figure 42) which shows an overview of detected vulnerabilities for that app. As can be seen, the data is presented in a matrix format where the columns represent categories, while the rows represent the origin of vulnerabilities. Rows are grouped as vulnerabilities detected in first party (seen as APP in Figure 42) or third-party (seen as LIB in Figure 42) code. Similarly, columns are further grouped in accordance to the severity (HIGH, LOW, MEDIUM, and NONE). The number of vulnerabilities is colour-coded with relatively higher numbers resulting in shades of red. Selecting individual count from the grid shows further details about the vulnerabilities for deeper investigation.



Other applications may trigger permission guarded actions without permissions — App Configuration — MEDIUM

Vulnerability

Category:	"App Configuration"
Type:	"node"
Severity:	"MEDIUM"
Description:	The concept of permissions enables the user to check what kind of actions applications can perform. This application contains a vulnerability which third party applications can exploit to trigger actions or get data, for which they would need to have a specific permission for. Due to the vulnerability, they can trigger these actions on behalf of the vulnerable app without needing to have the permission. This can be used to trick the user into thinking that the app does not require this permission.
Possible Fix:	If the component is only supposed to be used by other apps of the same developer, the app should define a new permission with the protection level signature. Other apps by the developer can use this permission. In order to do this, they need to be signed with the same key as the application containing the content provider. This ensures that no other application can access the content provider unless it has been signed by the same developer. If the component is supposed to be used by any other app, be sure to use checkCallingPermission to check whether the other app invoking the component has the appropriate permission.

Code Location

Class Name:	"jp.tjkapp.adfurikunsdk.h"
Member:	`boolean e(android.content.Context)`
Line:	14

Figure 43. The Report Comparison page which shows comparison between detected vulnerabilities in two versions of the same application [22].

The third view enables the comparison of analysis reports from two versions of the same app (Figure 43). As can be seen, the UI presents the two versions as two separate titles in separate colours. The most recent version is coloured red while the older one is coloured blue. This theme is followed throughout the UI; vulnerabilities detected in the new version are coloured red, while those in the older version are coloured blue, and those that are present in both are coloured grey. The intention of this colour scheme is to bring more attention to the newly introduced vulnerabilities. The detected vulnerabilities are displayed as a unit bar chart and are sorted in accordance to their severity and then according to their delta (present in only old version, both versions, or only new version). However, this sort order can be changed from a drop-down menu. Further, hovering the pointer on a vulnerability displays a tooltip with various attributes of that vulnerability, while clicking on the individual vulnerability shows its details on the right.

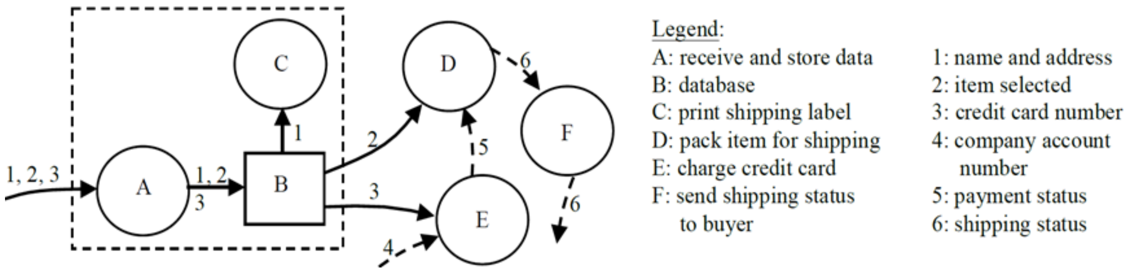


Figure 44. A PIM diagram showing information flow within an online shopping platform. The continuous and dotted arrows represent personal and non-personal information, respectively. The squares represent data storage and circles represent data processing step. The dotted box provides separation between distinct entities [23].

Privacy visualization

In [23], G. Yee proposes a workflow for visualizing privacy risks in a software system. The approach involves representing information source and destination with geometric figures like circles and squares. The data source or destination could be a storage or a processing step. Arrows are then drawn from source to destination to represent the transmission of information. The result is a diagram called Personal Information Map (PIM) that depicts private information flow within the software system. The PIM can then be analysed to find possible privacy risks at each step. Figure 10 shows an example of this approach applied on an online shopping portal. Visualization of privacy risks and their locations can help developers to implement measures to prevent them during development. However, the method proposed is manual and may not be suitable for complex systems without a proper automation engine. A similar work to this approach is presented in [24] where both security and privacy risks are evaluated using Extended Misuse Case (EMC) diagrams. In [24], the author integrates risk factors with security and privacy goals such as availability and confidentiality in an EMC diagram. The EMC diagram is then modelled as a Coloured Petri Nets (CPN) [32] for automatic evaluation of security and privacy risks.

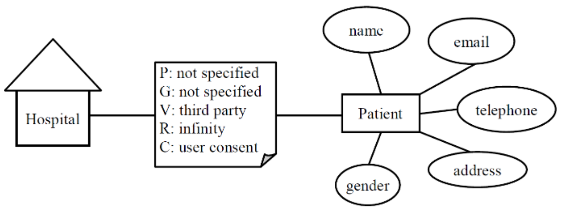


Figure 45. PPVM diagram of a health information system.

K. Ghazinour et al. [25] propose a visualization model to analyse the privacy policies of an application. Authors define privacy policy as a tuple of six elements: purpose, granularity, retention, constraint, and visibility. Purpose defines the motivation of the data collection, granularity defines the depth of personal information (for instance, exact age or range like child, adult, or teenager), retention defines the condition of expiry (pre-specified date, duration, number of accesses, voluntary termination, etc.), visibility defines who can access the data, and finally, and constraint defines any special condition under which data collection can occur (for instance, permission via specific emails to the customers). The values of these elements are set by the data collectors. The visualization model illustrates the logical relation between data collectors and providers along with the tuple elements in a representation called the Privacy Policy Visualization Modelling (PPVM) diagram. Figure 45 shows an example of a PPVM of a health information app which collects relevant personal information from the patients. Such visualization is helpful for the policy officers or security analysts to improve, debug or optimize the designed privacy policies. Visualizations of Software Project Characteristics

2.5. Visualizations of Software Release Level Statistics

Software development is usually done incrementally, defining continuous releases that provide value to the customer. To make an adequate delivery plan, it is necessary to consider the resources, the features to be developed, among other factors. In addition, it is necessary to ensure that the software version to be delivered is completely ready. Therefore, it is important to have tools that allow the visualization of the data produced in the integration and continuous release process.

Aseniero et al. [149], propose STRATOS, a tool that allows to visualize the factors that impact the release planning (e.g., resource allocation and stakeholder preferences) in a single layout to make the decision of the release plan much easier. The visualization is a hybrid visualization combining Sankey diagrams and parallel coordinates in a tree view. As future work, the tool will have tools for the user to manipulate the graphs presented.

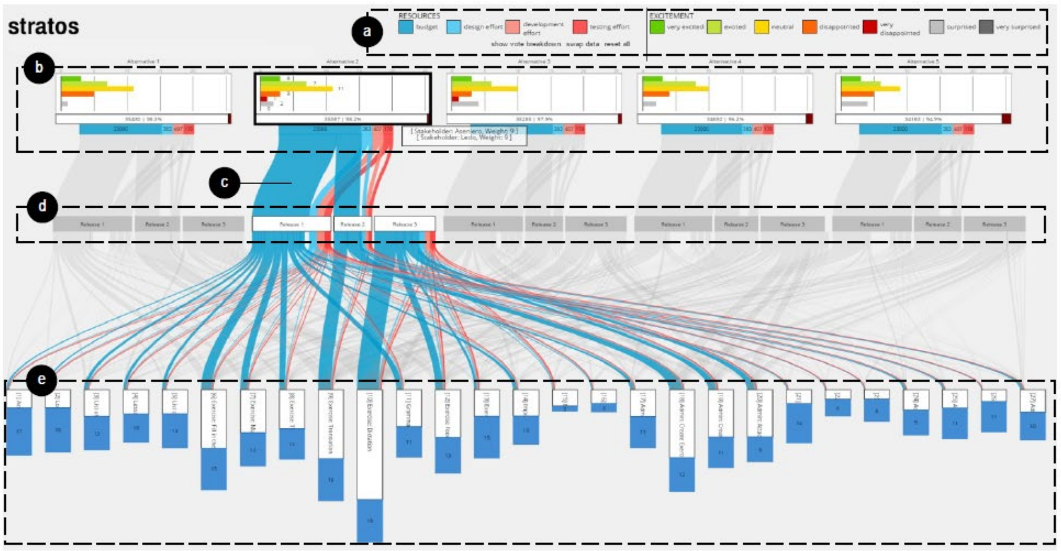


Figure 46. STRATOS' view of a release planning solution [149].

[60] presents softChange, a tool that takes information from sources such as emails, software releases, among others, finding new information (facts about the history of the project, its growth, the important milestones in its development, etc) and allowing the user to visualize it.

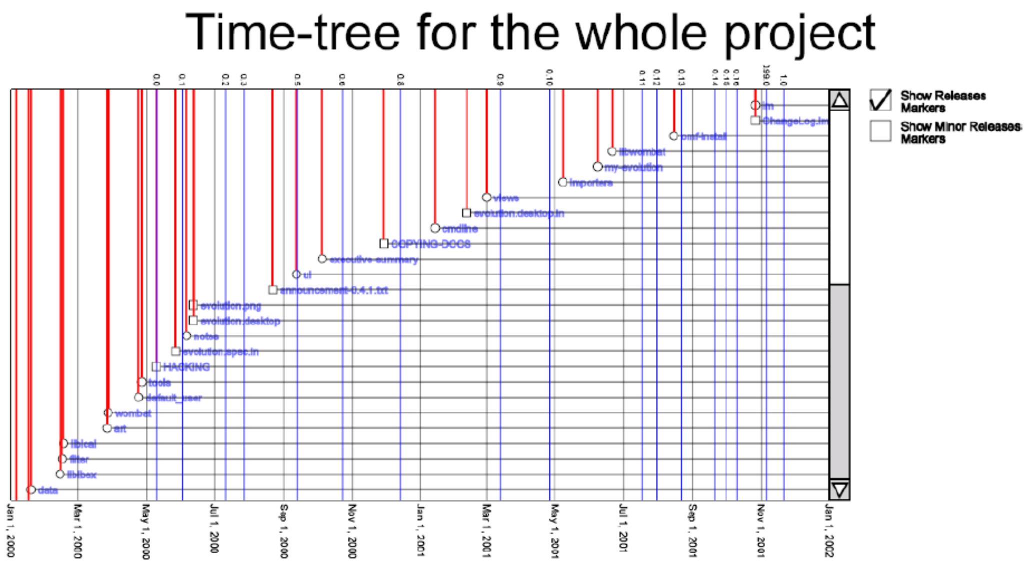


Figure 47. Time-tree in SoftChange [60].

It is important for companies to have information on software evolution through continuous releases, and for software engineers and managers to be able to use it for decision making. In this sense, [97] propose to use graphical technologies, specifically colour and third dimension to visualize entities such as time, system structure and attribute measures.

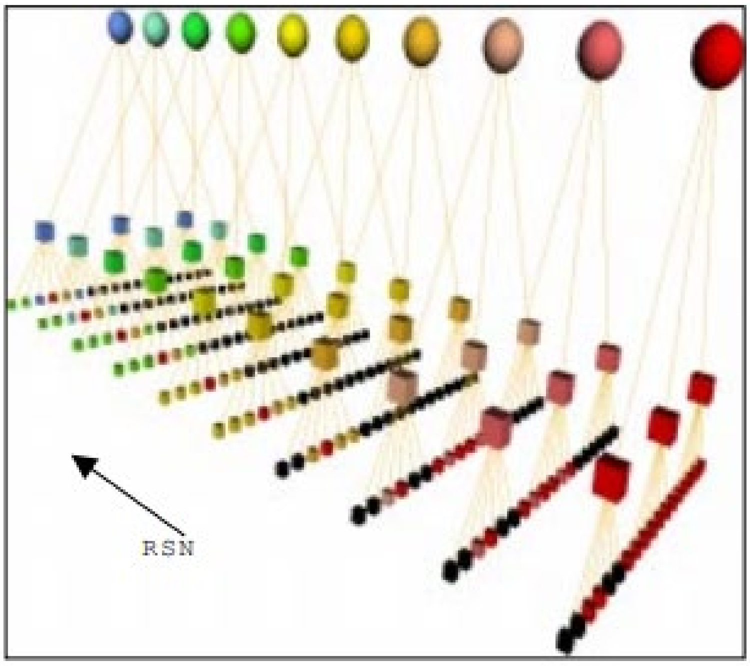


Figure 48. Visualizing the history [97].

According to the results obtained, three-dimensional graphics allow more information to be presented in an understandable way than a 2D graphic. However, one may argue that two-dimensional visualization contains more *actionable* information than 3D. For example, if one needs to show a pull request to a code reviewer in Gerrit, 2D is more effective to focus on what matters. Using different colour scales makes the information easier to use and more intuitive than textual tables, however this remains a challenge for visually impaired people.

2.6. Faults-Related Visualizations

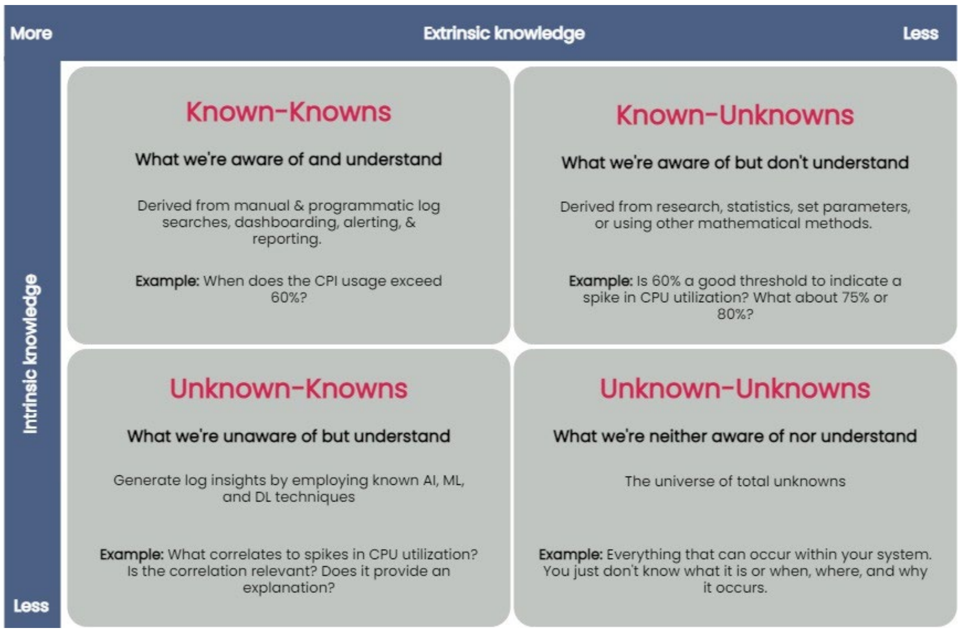
This section covers faults-related visualizations e.g., supporting test results over time and test execution times. As a note to the reader, this section is obviously closely related to “quality of testing” section in this deliverable.

During program debugging, fault (bug or defect) localization is the activity of determining the exact locations of program faults. It is a very expensive and time-consuming process. Its effectiveness depends on the developers' understanding of the debugged program, their ability to make logical decisions, their experience in debugging programs, and how suspicious code is identified and prioritized in terms of the probability of containing faults for inspection of potential fault locations. It is very common for programmers to have at their disposal huge amounts of data collected from program tests when debugging programs. The challenge is how to use such data to help them find program errors effectively [57].

Fault localization techniques can mainly be divided into statistics-based fault localization and machine learning-based fault localization. These fault localization approaches are applied to sequential programs, concurrent programs, distributed systems, and microservices. Generally, these approaches start with logging execution information at the service or thread or node level and then find faults using existing techniques [58].

What are the faults and when did they occur? The logs are examined for answers to the question. DevOps engineers must rely on a combination of logs from various system components and data sources to understand why an event occurred and to predict future events. As seen in Figure 49, faults can be categorized as follows for view and solution approaches [95]. These are;

1. *Known-knowns (What we're aware of and understand)*: Manual and scheduled log searches, dashboard creation, alerting, and reporting log fields are common ways to extract and analyze faults in this category. The DevOps engineer is familiar with data and knows where to look for answers in this type of faults.
2. *Known-unknowns (What we're aware of but don't understand)*: For faults in this category, The DevOps engineer can use statistics or other mathematical tools such as search terms and thresholds.
3. *Unknown-knowns (What we're unaware of but understand)*: Using AI/ML/DL techniques, DevOps engineers identify faults that fall into this category with approaches such as event correlations, log pattern clustering, and known anomaly detection.
4. *Unknown-unknowns (What we're neither aware of nor don't understand):* The most effective way to detect these errors is to use a good visualization tool. Using a log visualization tool (visualizations based on logs are covered in a separate section of this deliverable), the DevOps engineer can quickly isolate minor, anomalous changes in a system by visualizing patterns created from log data and discovering unknowns before an event occurs.



Known-Knowns
What we're aware of and understand
Derived from manual & programmatic log searches, dashboarding, alerting, & reporting.
Example: When does the CPI usage exceed 60%?

Known-Unknowns
What we're aware of but don't understand
Derived from research, statistics, set parameters, or using other mathematical methods.
Example: Is 60% a good threshold to indicate a spike in CPU utilization? What about 75% or 80%?

Unknown-Knowns
What we're unaware of but understand
Generate log insights by employing known AI, ML, and DL techniques
Example: What correlates to spikes in CPU utilization? Is the correlation relevant? Does it provide an explanation?

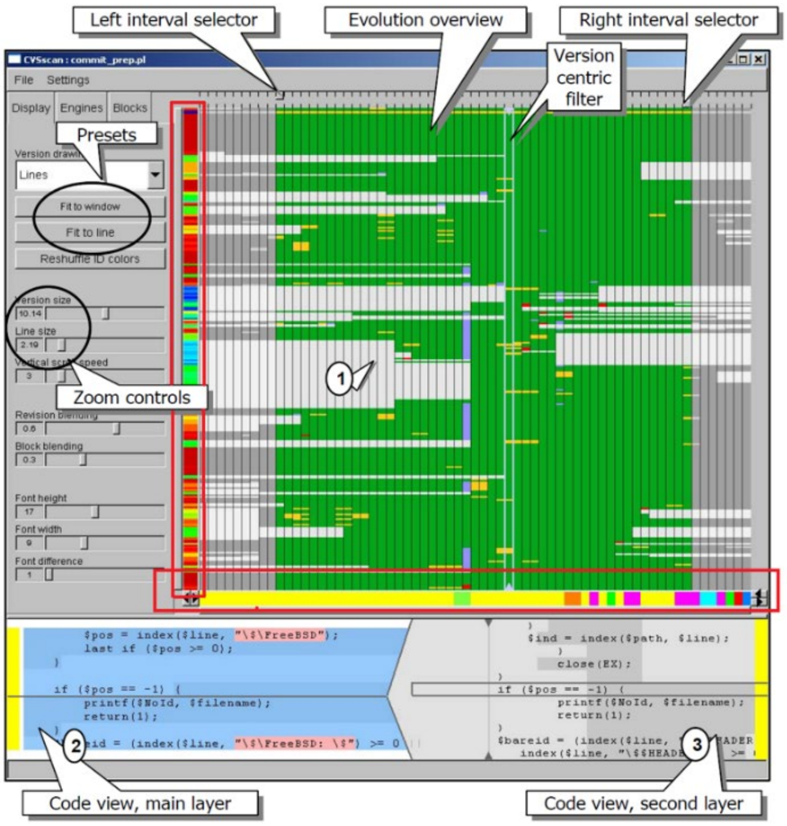
Unknown-Unknowns
What we're neither aware of nor understand
The universe of total unknowns
Example: Everything that can occur within your system. You just don't know what it is or when, where, and why it occurs.

Figure 49. Known and Unknown Fault Categories [95].

2.7. Visualization of Software Evolution

Throughout the development lifecycle, software systems undergo continuous changes in the form of bug fixes, feature updates, and optimizations. As the software evolves over time, it becomes increasingly difficult to track the changes in its quality characteristics. This is especially true in an environment where multiple developers work on various features simultaneously. With many commits every day, a product may have hundreds of development and release versions with their own quality characteristics. Thus, it becomes important to have an overview of the entire system and its development history such that any degradation or improvement in quality can be traced back to the exact change that caused it. This section provides a discussion on the visualization systems that intend to provide such functionality.

One of the effective methods to visualize code changes across multiple versions is through line-based displays. In a line-based view, each line within a code file is represented as a pixel line. This representation allows visualization of all the lines in a single file over multiple versions in a single view. In [54], L. Voinea et al. present a line-oriented display prototype called CVSscan that visualizes data from CVS Version Control System [55]. The visualization produced by a prototype is shown the figure below. The organization of lines in a file across multiple versions forms the evolution view. The horizontal axis represents time while the vertical axis shows line positions. The vertical stripes depict file versions and the horizontal compartments within the stripes represent individual lines in the file.



```
        $pos = index($line, "\$\FreeBSD");
        last if ($pos >= 0);
    }

    if ($pos == -1) {
        printf($NoId, $filename);
        return(1);
    }
    $bareid = (index($line, "\$\FreeBSD: \$") >= 0
```

```
    }
    $ind = index($path, $line);
        }
        close(EX);
    }
    if ($pos == -1) {
        printf($NoId, $filename);
        return(1);
    }
    $bareid = (index($line, "\$$HEADER
      index($line, "\$$HEADER >=
```

Figure 50. Code change overview visualized by the CVSscan tool.

The dashboard allows users to visualize changes according to three different attributes: line status, author, and construct. A line status could be constant, modified, deleted, or inserted. Each status is represented by a customizable colour map as shown in Figure 51 (a). Similarly, a customizable colour scheme is used to differentiate different constructs like comments, code block, and macros. Finally, each author is assigned a fixed, easily distinguishable colour (Figure 51 (b)). These attributes allow developers to view who authored a particular change, which version a particular line was changed, and whether the change was an edit, delete, or insert operation within a particular file.

In Figure 50, the changes are visualized according to the line status attribute in the evolution view. Additionally, the dashboard also provides two metric views (marked in red) and a code view. The horizontal and vertical metric views complement the information provided by the evolution view. The application allows users to select what information they want to display in the horizontal and vertical views. For instance, in the figure, the horizontal view depicts colour coding according to the version author while the vertical view displays the lifespan of the corresponding line position. The code view at the bottom of the screen is divided into two columns. The first column or the main view displays code around the current cursor position. A vertical movement of the cursor scrolls through the code in that version, while horizontal movement shows how the line changes across multiple versions. However, when the cursor encounters a blank line (light grey) in the figure, the main view freezes to the content it was displaying before and the second column or the second layer now displays the code that was deleted in the prior version or will be inserted in next versions.

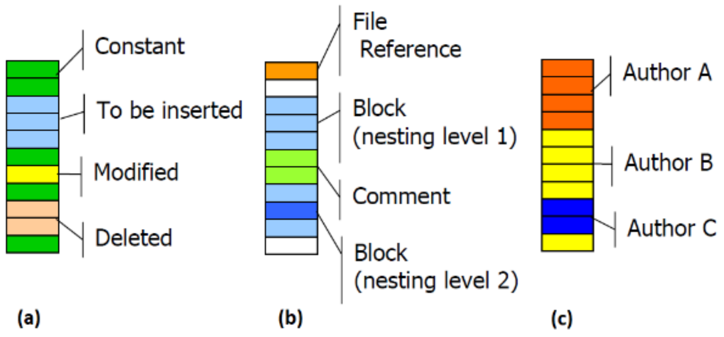


Figure 51. Line colour scheme based on: (a) line status (b) construct (c) author.

Line-oriented visualizations enable comparison of changes only across a single file at a time and thus do not visualize structural (e.g., package and class nesting) and relationship (e.g. class inheritance, and access and call relations) changes.

In [76], F. Steinbrückner et al. use the city metaphor to visualize development history. Static information of a software which includes information like package and class nesting, class type and size, call and access relations, type inheritance relations, and module creation and modification time are used to create a software cities representation. Figure 52 (a) shows the layout for a single version. In the diagram, the streets represent Java packages (subsystems), while the buildings represent Java classes (modules). The size of the individual buildings represents the coupling of the corresponding class with the system i.e., the number of incoming and outgoing function calls, inheritance relationships, and attribute and type accesses. Thus, the size of a module indicates its potential impact on the system. Newly added elements are attached at the end of the street and thus the representation grows as the versions progress. Elements removed from the system are not removed from the layout but instead are represented with a different colour. This preservation of removed elements prevents the layout from changing disruptively from version to version.

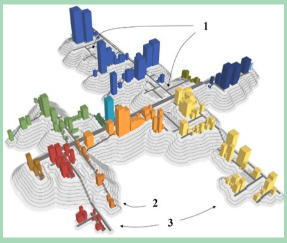
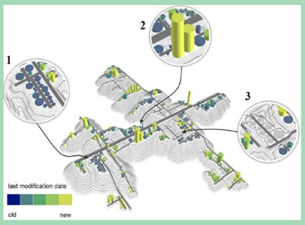
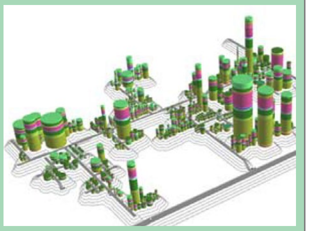
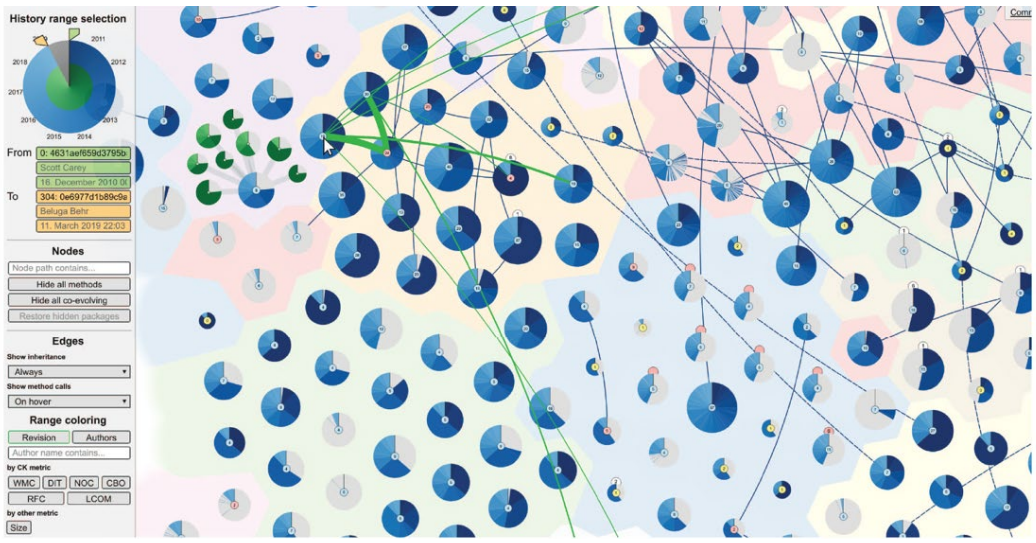
			
(a)	(b)	(c)	(d)

Figure 52. Using city metaphor to visualize software evolution [76]. (a) A simple layout showing decomposition into subsystems and modules. (b) Evolution map of the system. (c) Evolution map showing modification history. (d) Evolution map showing authorship history.

To separate changes in different versions, the concept of elevation is introduced. An older version is elevated to a higher level than the newer version. These elevation levels are represented by contour lines as shown in Figure 52 (b). Thus, a smaller number of contour lines beneath subsystems indicate that the elements are added in the recent version. The resulting layout is an evolution map that incorporates the evolution of different elements of the system throughout multiple versions in a single representation. The property towers can be changed to visualize different aspects of the system. For instance, in Figure 52 (b), the property towers represent individual modules. The height of individual towers is proportional to the module size and the colour indicates the top-level containing subsystem. There are several visual patterns in the representation, (1) the grey spots indicate moved or deleted modules, (2) modules growing along the hillsides represent continuously growing subsystems, and (3) newer subsystems are always on the extremities. The cylindrical towers with different colour gradients in Figure 52(c) are used to indicate the modification time of each class. Similarly, cylinders segmented with multiple colours as shown in Figure 52 (d) are used to represent developers responsible for individual changes in a particular class file.

F. Rabbi et al. [86] present a similar system called SysMap which uses the city metaphor for visualization. The system uses a 3D visualization that tracks various metrics like Lines of Code (LOC), number of packages, number of classes, and Weighted Method per Class (WMC) over multiple versions. However, it does not provide an option to view the evolution of the software in a single view. Users have an option to compare the software evolution in terms of the specified metrics in a separate bar chart, but the 3D visualization has to be viewed in separate windows for manual comparison between different versions. Further, a common drawback of using metaphors for visualization is that the display can easily get complex for large number of components and versions.

In [87], C. V. Alexandru et al. present Evo-Clocks, a unique form of software evolution visualization. The system visualizes both structural and fine-grained changes in a node-link representation. The core idea behind the approach is to use nodes to represent individual components, for instance, a Java class and its metrics over ranges that comprise several versions. Each node is visualized as a pie chart where the segments show the state of the metric being visualized at a certain point in time, thus each node represents the history of the corresponding component. Relationships like inheritance and method calls form the links between the nodes. Figure 53 shows a sample Evo-Clocks visualization.



From 0: 4631aef659d3795b
Scott Carey
16. December 2010 0(
To 304: 0e6977d1b89c9e
Beluga Behr
11. March 2019 22:03

Show inheritance: Always
Show method calls: On hover

Figure 53. Evo-Clocks representation of development history [87].

The application clusters all nodes or "clocks" of the same package together. Each cluster is then represented with a distinct background colour and hovering the pointer over any cluster spawns a tooltip showing the group's name. Similarly, hovering on a sector displays commit metadata while a counter at the centre of a node represents the number of methods contained in that primary node. The application also provides an option to select clusters and hide the rest to prevent the diagram becoming exceedingly complex. Moreover, to reduce the visual clutter, co-evolving nodes or the nodes that have the exact same revision range are grouped together and only one of the nodes is shown in the UI while others are shown as small semi-circles along the periphery of the displayed node. Clicking on the selected semi-circle then reveals the full node. By default, the entire history of the project is visualized; however, the diagram is interactive and allows users to select a time frame to filter changes using the history range selection widget (as seen on the top left corner in Figure 53). Further, clicking on a particular node reveals second-level nodes which represent methods within the selected class. The colour gradients for first level and second-level nodes are different (blue for the first-level and green for the second level in Figure 53) so that they are easily distinguishable. The links between second level and first-level nodes are represented with thick, semi-transparent bands, while the inheritance and outgoing method calls are depicted with thin blue and green lines, respectively.

2.8. Visualization of Microservice-Based Software Project

The complexity and dynamism of microservices systems poses great and unique challenges for debugging, as DevOps engineers must reason about the concurrent behavior of different microservices and understand the interaction topology of the entire system. A fundamental and effective way to understand and debug distributed systems is to monitor and visualize system executions [56].

The microservice architecture has led to an increase in mean resolution time (MTTR) in DevOps environments. The modularity of each service, the ability to be designed and deployed independently of the other services it interacts with, the increasing complexity of the software, and the distributed architecture make it difficult to understand performance and troubleshooting issues. In these respects, it has become important to monitor microservices both to respond to problems and to predict system behavior [46,47]. Figure 54 shows view of the dashboard for microservice-based software system [47]. We can group the points to be considered while monitoring microservice-based software products in 2 ways as follows.

- Monitoring containers and their components for both system performance and service performance
- Monitoring and mapping microservice interactions and service calls

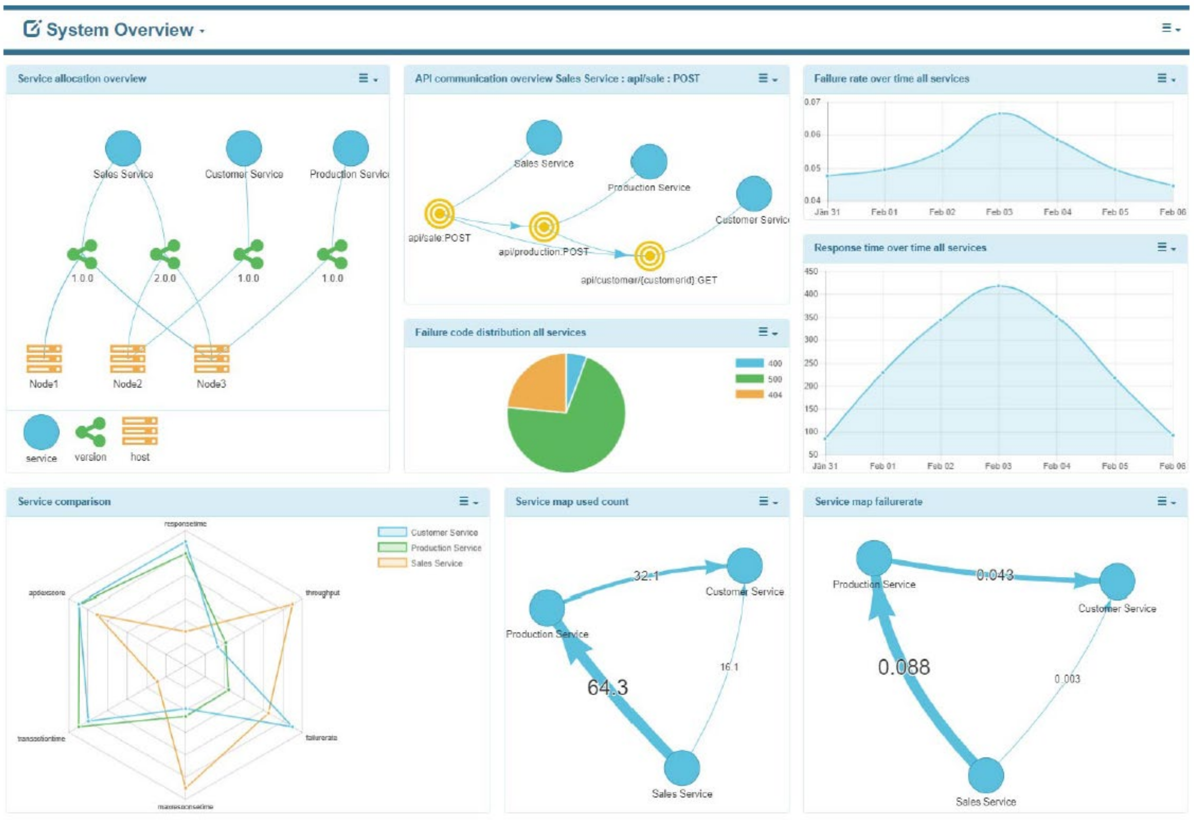


Figure 54. Microservice based system dashboard [47].

Additionally, services can have great differences in design and deployment. As a result, it will be difficult to get the big picture. By monitoring interactions between services, if something breaks, teams can quickly isolate the problem and avoid application errors.

Monitoring microservices, especially in a pre-production environment, can complement test environments by providing the following benefits [48].

1. Performance monitoring: Provides insight into how applications are performing when deploying new versions of code across various environments. In this way, rapid problem resolution and rapid testing of changes are obtained. Secure deployment takes place with real-time verification that new releases are performing as expected.
2. Monitoring Continuity: Monitoring continuity is important for DevOps teams. New updates and changes should support tracking capabilities. Consistency of development, test and operations teams on what and how they monitor is vital to maintaining visibility.

There are some solutions such as Dynatrace [49], NewRelic [50], Prometheus [51], Zipkin [52], Elastic Stack, Logstash, Kibana [53] which are used to monitor microservices based systems. These solutions provide full stack trace by capturing infrastructure and service information.

3. Technologies for Software Visualization

The focus of this section is to summarize tools and technology stacks that have been used for visualizing artefacts related to software development.

3.1. Software Visualization Tools: A Comparison

Many software visualizations and tools have recently been accessible to aid processes of analysis, modelling, testing, debugging, and maintenance. The software tools that users get the most benefit from offers time saving, better understanding, increased productivity, complexity management, quality improvement and assistance in locating code errors. The users also put emphasis on tool reliability, quality of the user interface and tool documentation [44]. Beside development purposes, software visualization tools can also improve users' business intelligence by bringing a better understanding of the product to partners and potential customers.

According to above mentioned needs and trends, we have analysed and established differences, similarities, strengths, and weaknesses of major visualization tools in the market such as Grafana, Kibana and Chronograf.

Grafana

Grafana open source is a visualization and analytics tool that analyses metrics, logs, and traces from any location, transforms data from a time-series database (TSDB) into graphs and visualizations. Primarily, Grafana allows users to quickly create dashboards, as well as edit metrics and functions. Grafana's rapid, client-side rendering allows users to generate extensive charts with smart axis formats.

Grafana offers integration with various data sources, with a core plug-in: time-series databases like Prometheus, Graphite, OpenTSDB, InfluxDB; logging and document databases like Elasticsearch and Loki; distributed tracing systems like Jaeger and Zipkin; SQL databases like MySQL and PostgreSQL; and public cloud monitoring tools like AWS CloudWatch, Google Cloud Monitoring, and Azure Monitor [65].

Grafana has also made available different visualization styles such as stat panel, bar gauge, and offers various display settings with bars, lines, and staircase graphs.

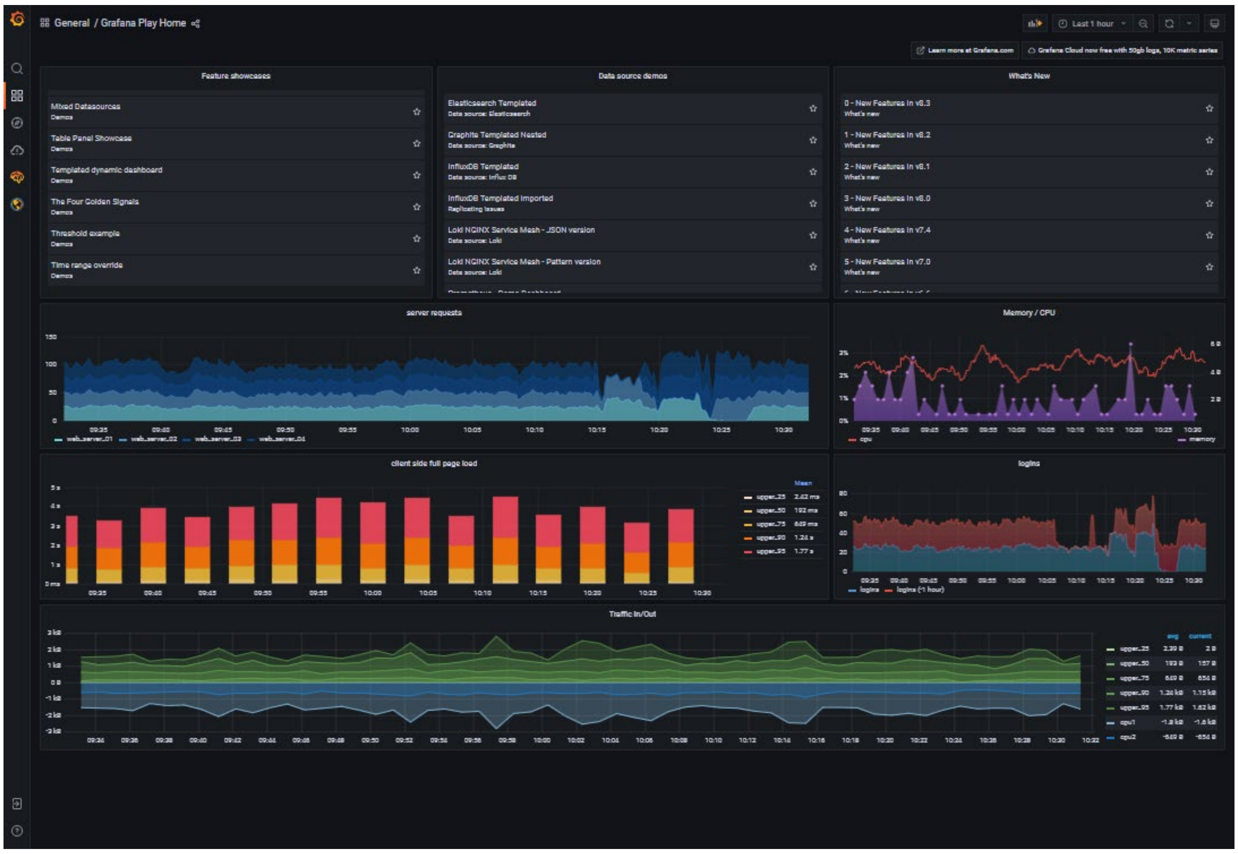


Figure 55. Visualization of Data in Grafana [66].

The figure below presents a dashboard in Grafana that shows the graphics with data gathered from different parameters (air quality, movement, dust, acoustics) via MQTT topics.

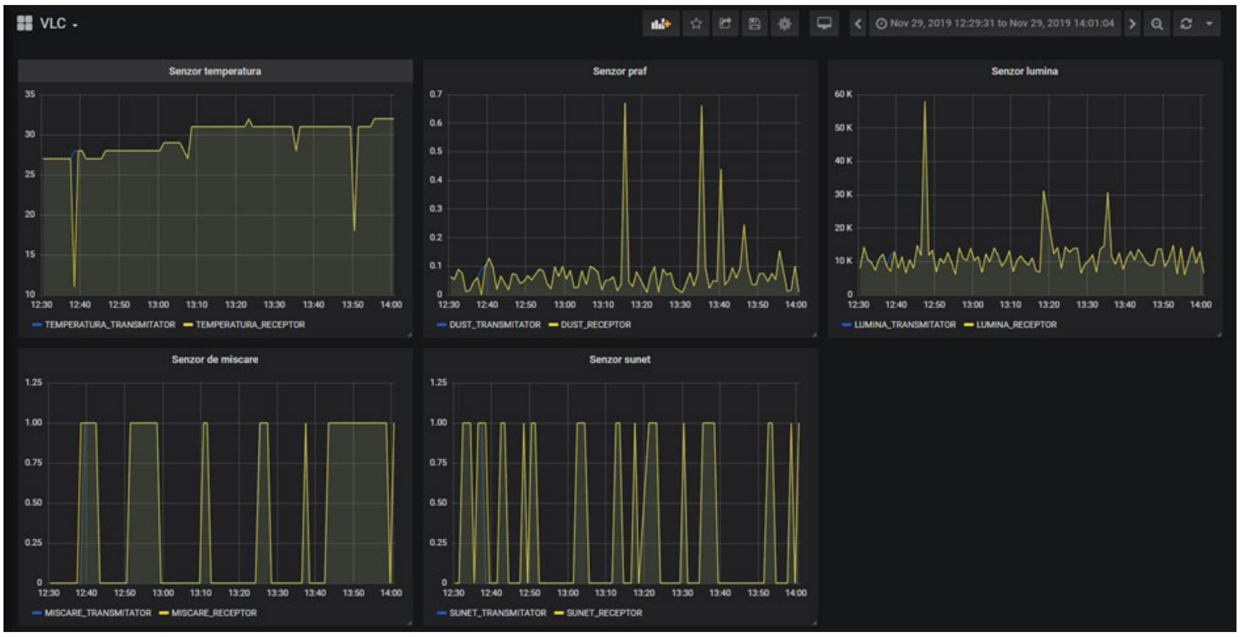
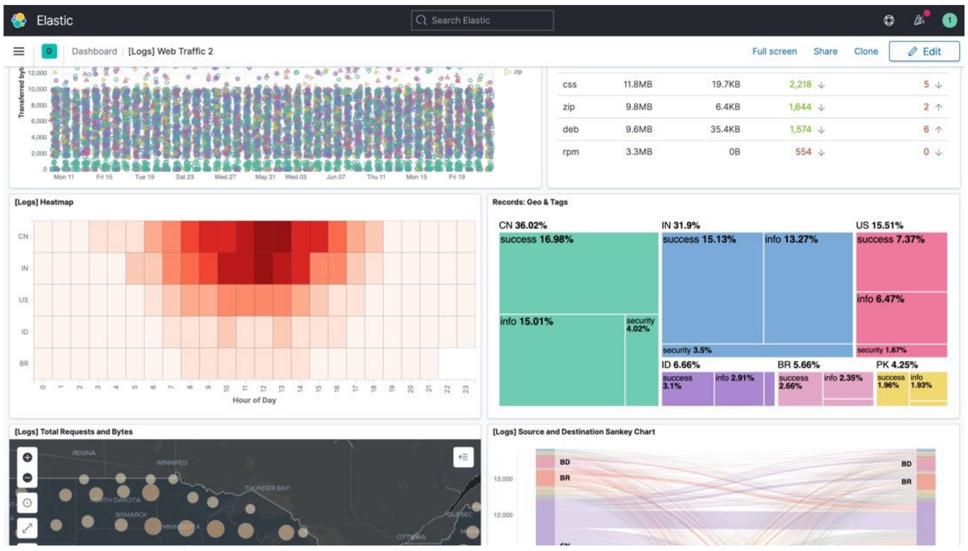


Figure 56. An example of BEIA's Grafana dashboard with data gathered from different types of sensors.

Kibana

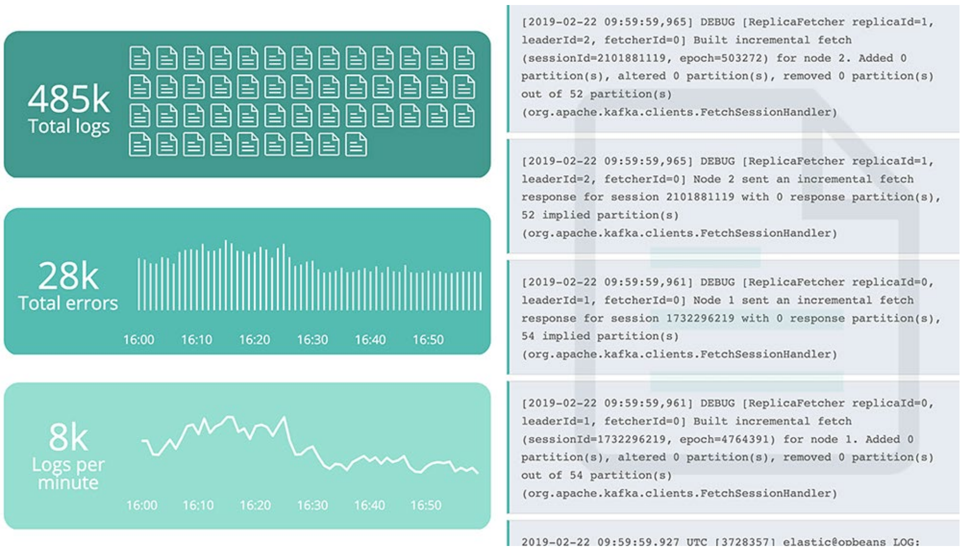
Kibana is a data visualization dashboard software of Elasticsearch used for log and time-series analytics, application monitoring, and operational intelligence use cases.

Kibana's main functionality is data analysis and querying. Users can search the data indexed in Elasticsearch for specific events or strings inside their data using a variety of ways to do root cause analysis and diagnostics. Users may utilize Kibana's visualization tools to show data in several ways, including charts, tables, geographical maps, and other forms of visualizations, based on these queries.



css	11.8MB	19.7KB	2,218 ↓	5 ↓
zip	9.8MB	6.4KB	1,644 ↓	2 ↑
deb	9.6MB	35.4KB	1,574 ↓	6 ↑
rpm	3.3MB	0B	554 ↓	0 ↓

Figure 57. Web Traffic [67].



```
[2019-02-22 09:59:59,965] DEBUG [ReplicaFetcher replicaId=1,
leaderId=2, fetcherId=0] Built incremental fetch
(sessionId=2101881119, epoch=503272) for node 2. Added 0
partition(s), altered 0 partition(s), removed 0 partition(s)
out of 52 partition(s)
(org.apache.kafka.clients.FetchSessionHandler)

[2019-02-22 09:59:59,965] DEBUG [ReplicaFetcher replicaId=1,
leaderId=2, fetcherId=0] Node 2 sent an incremental fetch
response for session 2101881119 with 0 response partition(s),
52 implied partition(s)
(org.apache.kafka.clients.FetchSessionHandler)

[2019-02-22 09:59:59,961] DEBUG [ReplicaFetcher replicaId=0,
leaderId=1, fetcherId=0] Node 1 sent an incremental fetch
response for session 1732296219 with 0 response partition(s),
54 implied partition(s)
(org.apache.kafka.clients.FetchSessionHandler)

[2019-02-22 09:59:59,961] DEBUG [ReplicaFetcher replicaId=0,
leaderId=1, fetcherId=0] Built incremental fetch
(sessionId=1732296219, epoch=4764391) for node 1. Added 0
partition(s), altered 0 partition(s), removed 0 partition(s)
out of 54 partition(s)
(org.apache.kafka.clients.FetchSessionHandler)

2019-02-22 09:59:59.927 UTC [3728357] elastic@opbeans LOG:
```

Figure 58. Log Analysis [67].

Chronograf

Chronograf is an open-source web application and visualization tool developed by InfluxData which helps to visualize system monitoring data and build queries, alerts and automation rules.

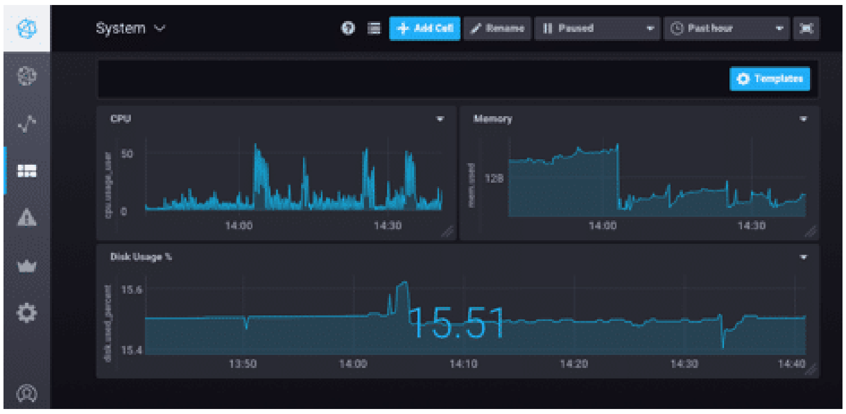


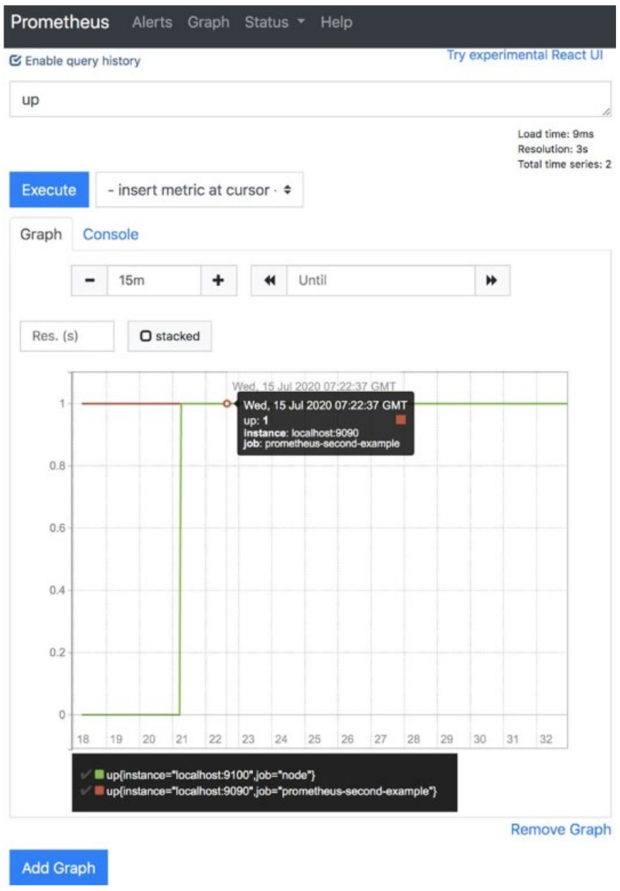
Figure 59. Visualization of Data in Chronograf [68].

Prometheus

Prometheus, suitable for dynamic environments, can collect metrics from a system over HTTP and place it into a time-series database that yields results on how the applications and underlying infrastructure is behaving [69].

Prometheus can scrape metrics in different ways such as using Node exporters, third-party exporters, and client libraries. Prometheus officially supports Go, Java, Python, and Ruby.

Prometheus offers a built-in visualization tool called Expression Browser, also a few console templates that can be used to create different consoles which are then served directly from the file system of the Prometheus server. With these dashboards, users may generate customizable web pages and execute source control. Because console templates are such a strong feature, they are recommended only be used to solve sophisticated use cases [70].



up

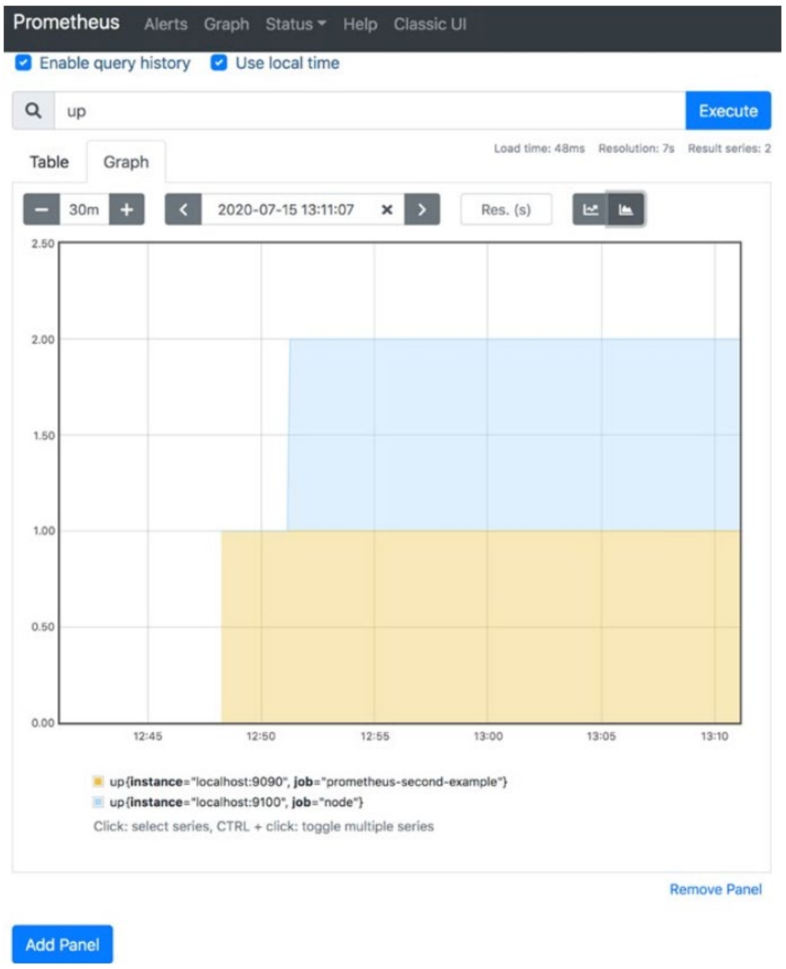
Load time: 9ms
Resolution: 3s
Total time series: 2

15m

Wed, 15 Jul 2020 07:22:37 GMT
up: 1
instance: localhost:9090
job: prometheus-second-example

🟩 up{instance="localhost:9100",job="node"}
🟥 up{instance="localhost:9090",job="prometheus-second-example"}

Figure 60. Graphical Representation [71].



up

Load time: 48ms Resolution: 7s Result series: 2

30m 2020-07-15 13:11:07

🟨 up{**instance**="localhost:9090", **job**="prometheus-second-example"}
🟦 up{**instance**="localhost:9100", **job**="node"}
Click: select series, CTRL + click: toggle multiple series

Figure 61. Stacked Graphs [72].

TOOLS/FEATURES	GRAFANA	KIBANA	CHRONOGRAF
Open-Source	Yes	Yes	Yes
Data Sources	More than 30+ (Built-in integration with Graphite, Prometheus, Elasticsearch, PostgreSQL, MySQL, InfluxDB, and AWS CloudWatch)	Elasticsearch	InfluxDB, Kapacitor, and Telegraf
Visual Types	Both simple and advanced visual types	Both simple and advanced visual types	Basic variations of visuals
Compatible with Difference OS	Cross-Platform	Cross-Platform	Cross-Platform

TOOLS/FEATURES	GRAFANA	KIBANA	CHRONOGRAF
Size of the Community [45]	Largest	Larger	Smaller
Versatile & Flexible Dashboards	Dynamic & Versatile (A single dashboard could reflect information from various sources)	Dynamic & Versatile	Only works with InfluxDB
Template Variables	Allows creation of dynamic variables	Allows creation of dynamic variables	Allows creation of dynamic variables
Alerts	Built-in alerting engine	Requires integration with ELK Stack, ElastAlert or X-Pack	Requires integration with Kapacitor
Free Trial	Offers more than 21 days	14-days	14-days
Pricing	Have flexible plans (free, pro, advanced, enterprise)	Have flexible plans (standard, gold, platinium, enterprise)	Have flexible plans (free, usage based, annual)

Table 1. A comparison of different features for a limited set of three commonly used tools.

3.2. Libraries

Some data visualization apps, such as Tableau, iCharts, Infogram, RAW Graphs, and Visualize Free, do not require programming languages and are not covered by this document. In this section, we inspect some of the most cutting-edge web technologies, third-party libraries and frameworks that can deliver custom visualization applications. These libraries and frameworks are built entirely in JavaScript, allowing users to communicate without the need for roundtrips to servers or external plugins [73].

HTML5: HTML5 is a markup language with a broad set of technologies (which includes CSS3) that enables more diversified and powerful websites and apps. It offers syntactic features and SVG support that eliminates dependency on CSS or JavaScript. In addition, HTLM5 defines JavaScript-compatible APIs and enhances Existing Document Object Model (DOM) interfaces.

jQuery: JQuery is a widespread, lightweight, and fast JavaScript library used by Microsoft, Amazon, Instagram, GoDaddy, Netflix, and ETSY. It is free and open source. With an easy-to-use API that works across a variety of browsers, it simplifies HTML document traversal and manipulation, event handling, CSS animation, and Ajax much simpler. Rather than adding HTML event attributes, to execute JavaScript methods, the jQuery library provides an easy syntax to add event handlers to the DOM using JavaScript. In this way, it encourages developers to keep JavaScript code isolated from HTML markup.

D3.js: D3.js, an open-source JavaScript library, offers interactive, browser-based data visualization by exposing the underlying elements of a web page in the context of a data set.

It is a toolkit for data visualization in web browsers, that uses web standards such as Cascading Style Sheets (CSS) for style, Scalable Vector Graphics (SVG) for vector graphics, HyperText Markup Language (HTML) for content and JavaScript for interactivity. Its compatibility with the browsers eliminates the need of an additional plug-in. D3.js is flexible and expressive, however, it is argued that learning curve can be steep [74].

GoJS: GoJS is another JavaScript library that can be used to create interactive diagrams and visualizations. With adjustable templates and layouts, GoJS makes it simple to create personalized and sophisticated diagrams. GoJS presents features such as drag-and-drop, copy-and-paste, in-place text editing, tooltips, templates, palettes, and an extensible tool system [75].

Highcharts: Highcharts library, built on JavaScript and TypeScript, comes with interactive charting capability, and offers 2D and 3D line charts, spline charts, area charts, bar charts, pie charts etc. It is compatible with many programming languages such as .Net, PHP, Python, R, Java, also iOS and Android. Highchart's customer base includes companies such as BBC, Twitter, SONY, MasterCard, Visa, and HSBC.

Bootstrap: Bootstrap is an open-source, JavaScript-based HTML, CSS & JS library which offers quick and customizable designs with templates and prebuilt components. Bootstrap is also powered by JavaScript plug-ins that provides various interface elements.

Chart.js: Chart.js, an open-source JavaScript graphics library, offers a good rendering performance by using canvas on HTML5 across all modern browsers. Chart.js can visualize data in 8 different ways; each of them animated and customizable. It can be integrated with plain JavaScript or with different module loaders [123].

FusionCharts: FusionCharts, is another commercial JavaScript library that provides 100+ Interactive Charts & 2,000+ Data-driven Maps. FusionCharts has various installation options available (direct JavaScript, CDN, NPM) and is pre-integrated with all popular JavaScript libraries and back-end programming languages [124]. It supports to process .xml and .json files and export the generated charts as .jpg, .png and .pdf files [125]. Also, consistent API across different charts makes it simple to build complex charts or dashboards.

FlotCharts: FlotCharts is a pure JavaScript plotting library for jQuery, with an emphasis on ease of use, aesthetics, and interactive features. FlotCharts isolates the functional code from the HTML structure and completes charting using the DOM (Document Object Model) It includes pre-made components for four fundamental chart types: charts-bar, line, point, and segment [126].

ZingChart: ZingChart library provides 50+ built-in chart types & modules with interactivity. Users can zoom or scroll through a visualized large data set, change plot visibility, overview data with highlight etc. ZingChart is built with performance in mind, it offers Canvas and SVG rendering options depending on the user's speed needs. In ZingChart, the entire state, data, and configuration of any chart is a JSON object, so they are 100% shippable and saveable across platforms. Lastly, ZingChart's robust API enables sending and receiving information to be used in charts easily [127].

3.3. Libraries Based on Other Programming Languages

GEPHI: Gephi is an open-source graph and network analysis program. It uses a 3D render engine to display large networks in real-time and to speed up the exploration [128].

NODEBOX: NodeBox Live is a web-based modular toolkit that allows users to build unique generative designs and captivating visuals. NodeBox Live bundles functionality in small connectable building blocks called nodes. Users can use these networks of nodes to produce many types of visual output and show what's going on at every stage of the project. With Nodebox Live users can work with CSV and JSON data or talk to an external web API [129].

GGPLOT2: ggplot2 is an open-source system for creating graphics written in R., based on The Grammar of Graphics, a data visualization approach that can build every graph from a data set, a coordinate system and gems that represent data points [130]. ggplot2 offers highly customizable, fine-tuneable elements, however, this makes ggplot2 a little slow [131].

PROCESSING: Processing is a free, flexible software sketchbook built for the visual arts. One of Processing's main aims is to provide non-programmers simplifications to grasp the principles of computer programming in a visual context. Processing uses Java and has a graphical user interface to make the compilation and execution process easier. Processing's integrated development environment includes a text editor, a compiler, and a display window to create two- and three-dimensional graphics [132].

JPGRAPH: JpGraph is an Object-Oriented graph and chart library for PHP 5, 7 and 8. It is developed in PHP and includes tools for making line plots, area plots, bar graphs, pie graphs, scatter plots and fields plot and more [133].

3.4. Techniques and Methods

We have also researched some techniques and methods that implements various visualisation techniques. A selection and examination of these can be seen below:

BLOOM allows understanding of software information (e.g., object allocations, optimizations made by the compiler, real-time compiler performance, and calls). This information is represented using the Box Tree, File Maps, Layouts, Spirals and Point Maps techniques [134].

CocoVizz uses 2D and 3D views to represent packages, classes, methods, and values of some software measures. This tool combines visualization with audio capabilities to aid understanding of the software. When interacting with the visualization components, sounds are emitted specifics that correspond to different meanings (e.g., code smells and when the software has undergone many/few changes) [135].

CodeForest allows understanding the structure and hierarchy of software using a forest. This tool uses two types of trees. The first type corresponds to classes and the second type represents detailed information about methods of a class. The measures equivalent to the classes are mapped by the variation of tree attributes (e.g., height, trunk radius and colour, branch radius and colour, leaf quantity and colour) [136].

MOOSE (Montreal Object-Oriented Slicing Environment) provides support for understanding software using various cognitive models. Its architecture is divided into five components:

i) software visualization
ii) algorithmic techniques
iii) combination of algorithmic techniques with software visualization
iv) repository for storing information
v) assistant tasks.

The software view has two views based on static analysis (Call Graph and Class Diagram) [137].

Concept (Comprehension of Net-Centered Programs and Techniques) is an evolution of MOOSE. Its aim is to use reverse engineering-based approaches to create mental abstractions. In addition, it uses the concept of software slicing to reduce the amount of information to be represented in views. This tool allows you to statically visualize hierarchical components of the software and their relationships, using TreeMap and Hiperbolic Tree. The code can be visualized in Class Diagrams and code comments are extracted and visualized decoupled from the code. Dynamic aspects can be visualized with Sequence Diagram and Collaboration Diagram [138] [139].

SHrimp (Simple Hierarchical Multi-Perspective) is a plug-in for the Eclipse IDE that allows the visualization of the hierarchical structure of the software. This tool combines different types of visualization (eg, TreeMap and Nested Graphs). These views are combined for users to switch between visual representations, selecting the most suitable one to represent the desired information.

In the TreeMap representation, the packages are represented by the yellow colour, the classes are represented by the light green colour, the methods are represented by the dark green and blue colours and the attributes by the red colour. These colours are user configurable. The code corresponding to the visualization is shown in the box in the lower right corner [140] [141] [142].

Creole is Shrimp's integration with the Eclipse IDE. With this tool, you can visualize the relationship between packages, which occurs because of method calls and access to attributes. This information can be viewed in more detail, identifying methods and attributes that generate these relationships. For that, this tool has a configuration menu to select which information should be included in the visualization. The visual representations used are Nested View (default layout), Spring, Tree, Radial and TreeMap [143].

SourceMiner is a plug-in for the Eclipse IDE that visually presents the code structure and hierarchy and considers some software measures. This tool uses TreeMap, Polymetric View, Dependency View and Parallel Coordinates to represent the information. TreeMap and Polymetric Vision allow the understanding of the software hierarchy, as well as measure values related to this hierarchy. Dependency View allows understanding the coupling between software entities. Parallel coordinates allow understanding values of some measures throughout the evolution of the software. It has interaction resources that allow information filtering, navigability for the code and selection of specific colours to represent the measures [144] [145].

Softwarenaut visually represents large software, offering three perspectives to visualize them:

i) exploration: graphs are used that represent software modules and their relationships Each module is represented by smaller graphs, where the nodes correspond to packages, classes, and methods. The edges correspond relationships to the between these nodes. Different colours are used to represent each type of relationship.
ii) detail: on-demand details of the modules represented in the exploration perspective are presented; and
iii) map: the positions of each module in the software hierarchy are presented [146].

CRISTA (Code Reading Implemented with Stepwise Abstraction) allows code understanding using TreeMap, with each TreeMap component corresponding to a code block. The code file is represented by the blue border. The class is represented by the red border. The class components (methods, attributes, and loops) are represented hierarchically. The code can be viewed by selecting any TreeMap component. In addition, comments can be inserted in the visually represented code snippets, using the available resources.

This tool allows the generation of reports, useful for software documentation [147].

There are several more visualization techniques that has not been explored here such as: CodeCity, Evolve, GASP, Rigi, SA4J, SeeIT, ThemeScapes etc. In conclusion, we can say that visualization techniques and methods are extremely flexible, providing users to display the desired information, changing the dimensions used, the types of analysis needed.

4. Visualization Styles

This section covers information regarding, e.g., which visualizations are preferred for certain type(s) of data and which visualizations are suited for delta scenarios.

Visualizing data with graphics is an intuitive approach to communicating aggregated information. Proper visualization types help to understand data and reveal new insights. Picking wrong graphs can lead to a false perception of data [11]. Furthermore, those distort the interpretation and can mislead the reader. Thus, auditors have a responsibility to choose suitable graphs [7].

To visualize different Deltas in the model, categorical, ordinal, and numerical variables can be mapped to several *visual attributes* to receive a better representation [8]. Visual attributes are “visual qualities of objects” [9], including different types of charts, position, length, color, size, or shape. Those attributes significantly impact the reader´s understanding [10].

Andrew V. Abela describes a suitable framework to identify relevant visualization types for data, based on Gene Zelazny's classic work *“Saying It With Charts”.*

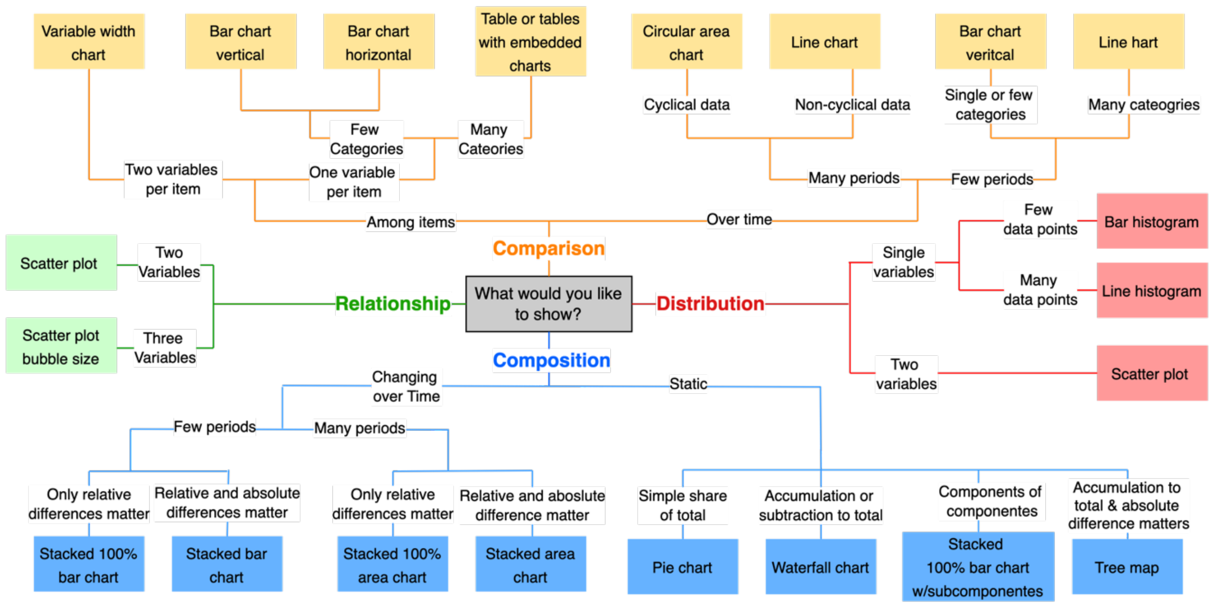


Figure 62. Decision tree to select a suitable chart type [based on [11].

Figure 62 illustrates the procedure using a decision tree. First, begin in the center and answer the question: "*What would you like to show?*". Finding a solution to this question gives the knowledge to follow the path of the shared tree to the appropriate chart type [11] [12]. The concept distinguishes between four terms:

1. **relationship**: shows a correlation between two or more variables
2. **comparison**: a set of variables is compared against one another
3. **composition**: collect different types of data and visualize them together to get a total overview
4. **distribution**: tries to understand the correlation – if possible - between unrelated and related variables [13]

Sample usage

Step 1. What would you like to show?
The goal is to find a suitable visualization to illustrate **changes indexed by time**.

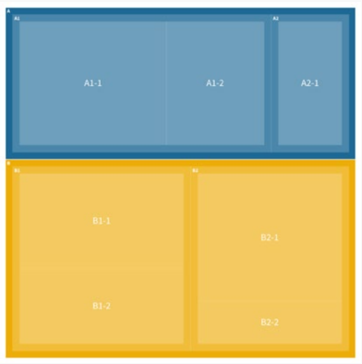
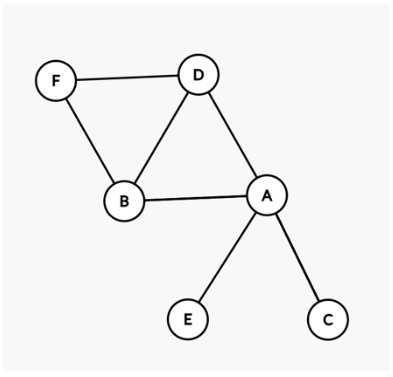
Step 2. Select between: *Comparison, Distribution, Composition,* and *Relationship*?
We try to identify the *differences* between the number of changes, so we go with *Comparison.*

Step 3. Choose between *Among items* and *Over time*?
We obviously follow the *Over time* branch.

Step 4. Next, we must differentiate between *Few* and *Many periods*.
This decision should be based on your data. We assume 48 timeslots, so we choose *Many periods.*

Step 5.* Finally, we must determine whether the data is *cyclic* or *non-cyclic data
In our case, we only have non-cyclic data. In this way, we should have a *line chart* for our case.

Andrew V. Abela introduced a framework that consists of "classical" approaches for data visualization (such as pie charts, bar charts, etc.), which can be used in different cases. As the amount of data becomes more extensive and complex, alternative - more easily understandable - high-level forms of visualizations could be addressed [14]. Some methods are listed in Table 2.

Visualization	**Example**	**Description**
2D/3D-Treemap		A Treemap consists of hierarchical data sets of nested rectangles and cubes. This type of diagram often replaces tree diagrams. Basic code structures and quality aspects are mapped to rectangles or cubes consisting of the different graphical properties: position, size, and colour. If the third dimension is added, the height is considered as well. A Treemap can be derived using different types of tiling algorithms.
Node-link diagrams		A node-link diagram consists of multiple entities (nodes) connected somehow. The relationship between entities is represented by a link (edge) usually containing further information. For instance, a microservice infrastructure could be an example of use. Each service is an entity, and the connection between each node is represented by a weighted edge that holds information about the data traffic [17].

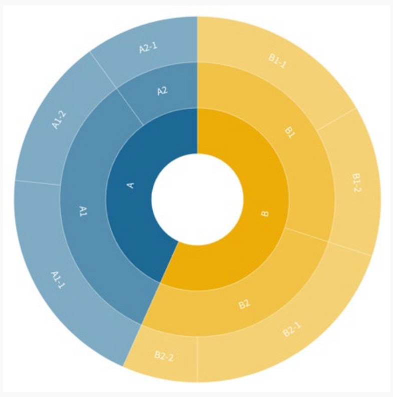
Visualization	Example	Description
2D/3D Sunburst representation		Besides node-link representation or Treemap, the sunburst representation can be used to visualize hierarchies. Arcs around the centre represent the different levels of hierarchy. The circle in the centre represents the root node. Siblings around are shifted radially by angels and related to the inner ring sections (parent nodes).

Table 2. High-level visualization representations [8] [14] [15].

5. Summary

Proper visualization types help to understand data and reveal new insights. There are numerous ways to visualize requirements, like in the form of entities [79]. In addition, there are tools to indicate quality issues with requirements, such as the RQA tool [81]. Model-based design environments provide a rich set of tools to indicate quality issues with requirements, e.g., MES MQC [100] uses different charts to present evolution of certain quality parameters over time. Similarly, MATHWORKS design quality status information is displayed using the metrics dashboard [103].

Software heat maps [105] are one way to visualize code coverage. A graph of connections between modules is proposed in [107] for assessing cyclic dependencies between modules. The X-Ray public domain tool [109] provides a system complexity view and class and package dependency views. Criticality of the test smells can be visualized using VITRum [85], while SonarQube [38] is a popular tool for automated code review.

LinearB [118] collects DORA metrics from different tools and repositories and are presented using graphs. Similarly, Haystack [119] integrates with a Git-based development workflow to collect metrics. Q-Rapids tool [122] also provides for the monitoring of quality metrics, especially in agile software development. Aseniero et al. [149], propose STRATOS, a tool that allows to visualize the factors that impact the release planning (e.g., resource allocation and stakeholder preferences) in a single layout to make the decision of the release plan much easier.

Different solutions exist for visual log and trace analyses such as Logstash for log collection and Elasticsearch for log indexing and retrieval.

The approaches presented in [61] and [64] can be used to compare energy consumption across different deltas of a software product. The SDK4ED Platform [16] enables the monitoring and optimization of important quality attributes, such as Energy Consumption, of software applications, with emphasis on embedded software.

Concerning visualization of privacy and security, various approaches have been proposed to improve security vulnerability triage. A common problem these systems aim to solve is aggregating high volume of information from security analysis tools and presenting them in a form that facilitates quick and error-free interpretation. Goodall et al. [18] present a system that integrates outputs from multiple security analysis tools into a single interactive visual environment, while a provenance-based code analysis and visualization is provided in [21]. Different visualization views in [22] shows e.g., an overview of detected vulnerabilities in scanned applications and a comparison between detected vulnerabilities in two versions of the same application.

Similarly, there are also systems that aim to detect privacy risks by visualizing privacy policies or private data flow within the software system. However, these approaches are mostly manual. Compared to security vulnerabilities, there are very few works on privacy vulnerability visualization [23]. Privacy policies of an application can be visualized using a visualization model proposed in [25] where a privacy policy is defined as a tuple of six elements: purpose, granularity, retention, constraint, and visibility.

The decision tree from Andrew V. Abela will support the creator in identifying appropriate chart types, including representations for relationships, comparisons, compositions, and distributions. In addition, 2D/3D-treemaps, node-link diagrams as well as 2D/3D sunbursts representation are common to visualize extensive and complex data.

Over the years, various techniques have been developed to visualize software evolution. Visualizing the evolution of individual components as well as the overall structure over time is challenging due to high volume of evolution data [88]. Visualization techniques like pixel lines facilitate a fine-grain analysis of changes committed to individual components but they fail to provide an overall context of the change as they do not cover structural or relationship changes. Similarly, techniques that visualize structural changes become cluttered as the versions progress. Recent approaches like the Evo-Clocks [87] try to solve this problem by using node-line display methodology where changes in individual components across multiple versions are compressed into a single node. This together with selective hiding of nodes, interactive filtering, and showing detailed changes only on demand allows the system to visualize both structural as well as fine-grained changes in a single view. However, for a system with many components and version history, this approach still results in an overwhelming representation. Thus, due to the volume of information to be displayed, there seems to always be a trade-off between visualizing fine-grained changes and displaying a global overview of structural evolution.

References

[1] M. Staron, Automotive Software Architectures: An Introduction, 2nd ed. Cham, Switzerland: Springer Nature, 2022.

[2] P. E. Strandberg, W. Afzal, and D. Sundmark, “Software test results exploration and visualization with continuous integration and nightly testing,” Int. J. Softw. Tools Technol. Transf., vol. 24, no. 2, pp. 261–285, 2022.

[3] M. Shahin, M. Ali Babar, and L. Zhu, “Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices,” IEEE Access, vol. 5, pp. 3909–3943, 2017.

[4] A. Ahmad, O. Leifler, και K. Sandahl, ‘Data visualisation in continuous integration and delivery: Information needs, challenges, and recommendations’, IET Software, 2021.

[5] M. O. Ward, G. Grinstein, and D. Keim, Interactive data visualization: Foundations, techniques, and applications, second edition, 2nd ed. London, England: CRC Press, 2021.

[6] “Dashboard in polyspace desktop user interface - MATLAB & Simulink - MathWorks Nordic,” Mathworks.com. [Online]. Available: https://se.mathworks.com/help/bugfinder/ug/dashboard.html. [Accessed: 25-May-2022].

[7] P. J. Steinbart, “The auditor’s responsibility for the accuracy of graphs in,” Accounting Horizons, vol. 3, no. 3, p. 60, 1989.

[8] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster, “Visualizing software changes,” IEEE Transactions on Software Engineering, vol. 28, no. 4, pp. 396–412, 2002.

[9] V. Ferrari and A. Zisserman, “Learning visual attributes,” Advances in neural information processing systems, vol. 20, 2007.

[10] M. Lu, J. Lanir, C. Wang, Y. Yao, W. Zhang, O. Deussen, and H. Huang, “Modeling just noticeable differences in charts,” IEEE Transactions on Visualization and Computer Graphics, vol. 28, no. 1, pp. 718–726, 2021.

[11] C.-h. Chen, W. K. Härdle, and A. Unwin, Handbook of data visualization. Springer Science & Business Media, 2007.

[12] G. Zelazny, Say it with charts workbook. McGraw-Hill, 2005.

[13] J. Rue, “Visualizing Data: A Guide to Chart Types,” 2019. [Online]. Available: https://multimedia.journalism.berkeley.edu/tutorials/visualizing-data-a-guide-to-chart-types/. [Accessed: 25-May-2022].

[14] P. Liggesmeyer, H. Barthel, A. Ebert, J. Heidrich, P. Keller, Y. Yang, and A. Wickenkamp, “Quality improvement through visualization of software and systems,” Quality Assurance and Management, InTech, pp. 315–334, 2012.

[15] G. Langelier, H. Sahraoui, and P. Poulin, “Visualization-based analysis of quality for large-scale software systems,” in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp. 214–223.

[16] M. Siavvas, D. Tsoukalas, C. Marantos, A.-A. Tsintzira, M. Jankovic, D. Soudris, A. Chatzigeorgiou, and D. Kehagias, “The sdk4ed platform for embedded software quality improvement-preliminary overview,” in International Conference on Computational Science and Its Applications. Springer, 2020, pp. 1035–1050.

[17] B. Saket, P. Simonetto, S. Kobourov, and K. B¨orner, “Node, node-link, and node-link-group diagrams: An evaluation,” IEEE Transactions on Visualization and Computer Graphics, vol. 20, no. 12, pp. 2231–2240, 2014.

[18] J. R. Goodall, H. Radwan, and L. Halseth, “Visual analysis of code security,” in Proceedings of the seventh international symposium on visualization for cyber security, 2010, pp. 46–51.

[19] F. ¨ O. S¨onmez and B. G. Kili¸c, “Holistic web application security visualization for multi-project and multi-phase dynamic application security test results,” IEEE Access, vol. 9, pp. 25 858–25 884, 2021.

[20] S. Chiasson, P. van Oorschot, and R. Biddle, “Even experts deserve usable security: Design guidelines for security management systems,” in SOUPS Workshop on Usable IT Security Management (USM). Citeseer, 2007, pp.1–4.

[21] A. Schreiber, T. Sonnekalb, and L. von Kurnatowski, “Towards visual analytics dashboards for provenance-driven static application security testing,” in 2021 IEEE Symposium on Visualization for Cyber Security (VizSec). IEEE, 2021, pp. 42–46.

[22] S. L. Reynolds, T. Mertz, S. Arzt, and J. Kohlhammer, “User-centered design of visualizations for software vulnerability reports,” in 2021 IEEE Symposium on Visualization for Cyber Security (VizSec). IEEE, 2021, pp. 68–78.

[23] G. Yee, “Visualization of privacy risks in software systems,” in Proceedings of the Tenth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2016), 2016, pp. 289–294.

[24] E. Paintsil, “A model for privacy and security risks analysis,” in 2012 5th International Conference on New Technologies, Mobility and Security (NTMS). IEEE, 2012, pp. 1–8.

[25] K. Ghazinour, M. Majedi, and K. Barker, “A model for privacy policy visualization,” in 2009 33rd Annual IEEE International Computer Software and Applications Conference, vol. 2. IEEE, 2009, pp. 335–340.

[26] D. G. Murray, Tableau your data!: fast and easy visual analysis with tableau software. John Wiley & Sons, 2013.

[27] A. Dieberger, P. Dourish, K. H¨o¨ok, P. Resnick, and A. Wexelblat, “Social navigation: Techniques for building more usable systems,” interactions, vol. 7, no. 6, pp. 36–45, 2000.

[28] K. J. Vicente and J. Rasmussen, “Ecological interface design: Theoretical foundations,” IEEE Transactions on systems, man, and cybernetics, vol. 22, no. 4, pp. 589–606, 1992.

[29] B. J. Fogg, “Persuasive technology: using computers to change what we think and do,” Ubiquity, vol. 2002, no. December, p. 2, 2002.

[30] L. Moreau and P. Groth, “Provenance: An introduction to prov (synthesis lectures on the semantic web: Theory and technology),” California, USA: Morgan and Claypool Publishers, 2013.

[31] “VUSC - the Code Scanner.” [Online]. Available: https://www.sit.fraunhofer.de/de/vusc/. [Accessed: 25-May-2022].

[32] Dieberger, A., et al. Social Navigation: Techniques for Building More Usable Systems. ACM Interactions, v.7(6), November-December, 2000.

[33] “luca App for Android.” [Online]. Available: https://www.luca-app.de/. [Accessed: 25-May-2022].

[34] “Dash Documentation & User Guide | Plotly.” [Online]. Available: https://dash.plotly.com/. [Accessed: 25-May-2022].

[35] “Plotly Python Open Source Graphing Library.” [Online]. Available: https://plotly.com/python/. [Accessed: 25-May-2022].

[36] J. B. Kruskal and J. M. Landwehr, “Icicle plots: Better displays for hierarchical clustering,” The American Statistician, vol. 37, no. 2, pp. 162–168, 1983.

[37] “OWASP Zed Attack Proxy.” [Online]. Available: https://www.zaproxy.org/docs/api/. [Accessed: 25-May-2022].

[38] “SonarQube documentation,” Sonarqube.org. [Online]. Available: https://docs.sonarqube.org/latest/. [Accessed: 25-May-2022].

[39] “OWASP Top Ten Web Application Security Risks | OWASP.” [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed: 25-May-2022].

[40] “Burp Suite - Application Security Testing Software.” [Online]. Available: https://portswigger.net/burp/. [Accessed: 25-May-2022].

[41] “Apache Tomcat.” [Online]. Available: https://github.com/apache/tomcat/. [Accessed: 25-May-2022].

[42] J. Bohnet, S. Voigt, and J. D¨ollner, “Projecting code changes onto execution traces to support localization of recently introduced bugs,” in Proceedings

of the 2009 ACM symposium on Applied Computing, 2009, pp.438–442.

[43] S. Diehl, Software visualization: visualizing the structure, behaviour, and evolution of software. Springer Science & Business Media, 2007.

[44] S. Bassil and R. K. Keller, “Software visualization tools: Survey and analysis,” in Proceedings 9th International Workshop on Program Comprehension. IWPC 2001. IEEE, 2001, pp. 8–11.

[45] V. Gupta, “Grafana vs. Kibana | MetricFire Blog.” [Online]. Available: https://www.metricfire.com/blog/grafana-vs-kibana/. [Accessed: 25-May-2022].

[46] M. Nehra, “What are the Top best Tools for Monitoring Microservices?” [Online]. Available: https://www.decipherzone.com/blog-detail/top-best-tools-monitoring-microservices/. [Accessed: 25-May-2022].

[47] B. Mayer and R. Weinreich, “A dashboard for microservice monitoring and management,” in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017, pp. 66–69.

[48] “Monitoring Microservices.” [Online]. Available: https://smartbear.com/learn/performance-monitoring/monitoring-microservices/. [Accessed: 25-May-2022].

[49] “Dynatrace | Automatic and Intelligent Observability.” [Online]. Available: https://www.dynatrace.com/. [Accessed: 25-May-2022].

[50] “New Relic | monitor, Debug and Improve Your Entire Stack.” [Online]. Available: https://newrelic.com/. [Accessed: 25-May-2022].

[51] Prometheus, “Prometheus - Monitoring system & time series database.” [Online]. Available: https://prometheus.io/. [Accessed: 25-May-2022].

[52] “OpenZipkin · A distributed tracing system.” [Online]. Available: https://zipkin.io/. [Accessed: 25-May-2022].

[53] “Elastic Stack, Logstash & Kibana.” [Online]. Available: https://www.elastic.co/. [Accessed: 25-May-2022].

[54] L. Voinea, A. Telea, and J. J. Van Wijk, “Cvsscan: visualization of code evolution,” in Proceedings of the 2005 ACM symposium on Software visualization, 2005, pp. 47–56.

[55] “CVS—Concurrent Versions System v1.11.23.” [Online]. Available: https://www.gnu.org/software/trans-coord/manual/cvs/. [Accessed: 25-May-2022].

[56] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, “Debugging distributed systems,” Communications of the ACM, vol. 59, no. 8, pp. 32–37, 2016.

[57] W. E. Wong and Y. Qi, “Bp neural network-based effective fault localization,” International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 04, pp. 573–597, 2009.

[58] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” IEEE Transactions on Software Engineering, vol. 47, no. 2, pp. 243–260, 2018.

[59] A. Ahmad, O. Leifler, and K. Sandahl, “Data visualisation in continuous integration and delivery: Information needs, challenges, and recommendations,” IET Software, vol. 16, no. 3, pp. 331–349, Jun. 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1049/sfw2.12030/.

[60] D. M. German and A. Hindley, “Visualizing the evolution of software using softchange,” International Journal of Software Engineering and Knowledge Engineering, vol. 16, no. 01, pp. 5–21, 2006.

[61] E. A. Jagroep et al., “Software energy profiling: Comparing releases of a software product,” in Proceedings of the 38th International Conference on Software Engineering Companion - ICSE ’16, 2016.

[62] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, “What do programmers know about software energy consumption?,” IEEE Softw., vol. 33, no. 3, pp. 83–89, 2016.

[63] S. A. Chowdhury and A. Hindle, “GreenOracle: Estimating software energy consumption with energy measurement corpora,” in Proceedings of the 13th International Conference on Mining Software Repositories, 2016.

[64] K. Aggarwal, A. Hindle, and E. Stroulia, “GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption,” in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015.

[65] M. Chakraborty and A. P. Kundan, “Monitoring cloud-native applications,” p. 195, 2021.

[66] “Grafana.” [Online]. Available: https://play.grafana.org/d/000000012/grafanaplay-home/. [Accessed: 25-May-2022].

[67] “Visualization of data in kibana.” [Online]. Available: https://www.elastic.co/kibana/. [Accessed: 25-May-2022].

[68] “Chronograf: Complete Dashboard Solution for InfluxDB.” [Online]. Available: https://www.influxdata.com/time-series-platform/chronograf/. [Accessed: 25-May-2022].

[69] M. Chakraborty and A. P. Kundan, “Monitoring cloud-native applications,” p. 99, 2021.

[70] M. Chakraborty and A. P. Kundan, “Monitoring cloud-native applications,” p. 121, 2021.

[71] M. Chakraborty and A. P. Kundan, “Monitoring cloud-native applications,” p. 125, 2021.

[72] M. Chakraborty and A. P. Kundan, “Monitoring cloud-native applications,” p. 128, 2021.

[73] F. Shahzad, T. R. Sheltami, E. M. Shakshuki, and O. Shaikh, “A review of latest web tools and libraries for state-of-the-art visualization,” Procedia Computer Science, vol. 98, p. 102, 2016.

[74] M. Bostock, V. Ogievetsky, and J. Heer, “D3 data-driven documents,” IEEE transactions on visualization and computer graphics, vol. 17, no. 12, pp. 2301–2309, 2011.

[75] F. Shahzad, T. R. Sheltami, E. M. Shakshuki, and O. Shaikh, “A review of latest web tools and libraries for state-of-the-art visualization,” Procedia Computer Science, vol. 98, p. 106, 2016.

[76] F. Steinbr¨uckner and C. Lewerentz, “Representing development history in software cities,” in Proceedings of the 5th international symposium on Software visualization, 2010, pp. 193–202.

[77] S. Diehl, Software visualization: Visualizing the structure, behaviour, and evolution of software. Berlin, Germany: Springer, 2010.

[78] M. Kamalrudin, J. Hosking, and J. Grundy, “Improving requirements quality using essential use case interaction patterns,” in Proceeding of the 33rd international conference on Software engineering - ICSE ’11, 2011.

[79] M. Osama, A. Zaki-Ismail, M. Abdelrazek, J. Grundy, and A. Ibrahim, “SRCM: A semi formal requirements representation model enabling system visualisation and quality checking,” in Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, 2021.

[80] G. Génova, J. M. Fuentes, J. Llorens, O. Hurtado, and V. Moreno, “A framework to measure and improve the quality of textual requirements,” Requir. Eng., vol. 18, no. 1, pp. 25–41, 2013.

[81] “SE life cycle Management, requirements quality Management,” Reusecompany.com, 11-Apr-2022. [Online]. Available: http://www.reusecompany.com/index.php?option=com_content&view=category&layout=blog&id=171&Itemid=75&lang=en. [Accessed: 25-May-2022].

[82] M. Rahimi, M. Mirakhorli, and J. Cleland-Huang, “Automated extraction and visualization of quality concerns from requirements specifications,” in 2014 IEEE 22nd International Requirements Engineering Conference (RE), 2014.

[83] B. Cornelissen, A. Zaidman, A. van Deursen, and B. van Rompaey, “Trace visualization for program comprehension: A controlled experiment,” in 2009 IEEE 17th International Conference on Program Comprehension, 2009.

[84] D. Gračanin, K. Matković, και M. Eltoweissy, ‘Software visualization’, Innovations in Systems and Software Engineering, τ. 1, τχ. 2, σσ. 221–230, 2005.

[85] F. Pecorelli, G. Di Lillo, F. Palomba, and A. De Lucia, “VITRuM: A plug-in for the visualization of test-related metrics,” in Proceedings of the International Conference on Advanced Visual Interfaces, 2020.

[86] F. Rabbi, N. T. Niloy, N. Nahar, M. Tawhid, and N. Ahad, “Sysmap: A lightweight software visualization tool to analyze the software evolution of a system,” arXiv preprint arXiv:2108.05989, 2021.

[87] C. V. Alexandru, S. Proksch, P. Behnamghader, and H. C. Gall, “Evoclocks: Software evolution at a glance,” in 2019 Working Conference on Software Visualization (VISSOFT). IEEE, 2019, pp. 12–22.

[88] T. Khan, H. Barthel, A. Ebert, and P. Liggesmeyer, “Visualization and evolution of software architectures,” in Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering-Proceedings of IRTG 1131 Workshop 2011. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[89] A. Nematzadeh and L. J. Camp, “Threat analysis of online health information system,” in Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments, 2010, pp. 1–7.

[90] “SolarWinds | log Visualization Tool - Visualize Log Data & Files.” [Online]. Available: https://www.solarwinds.com/log-analyzer/use-cases/log-visualization/. [Accessed: 25-May-2022].

[91] “EventLog Analyzer| log Visualization and Analysis Tool- Read and Analyze your Logs Online.” [Online]. Available: https://www.manageengine.com/products/eventlog/log-visualization-tool.html/. [Accessed: 25-May-2022].

[92] “Fluentd | Open Source Data Collector | Unified Logging Layer.” [Online]. Available: https://www.fluentd.org/. [Accessed: 25-May-2022].

[93] “Graylog.” [Online]. Available: https://www.graylog.org. [Accessed: 25-May-2022].

[94] “Nagios.” [Online]. Available: https://www.nagios.org/downloads/. [Accessed: 25-May-2022].

[95] T. C. Huang, “Visualize Logs to Get More Value from Data,” Jul. 2021. [Online]. Available: https://devops.com/visualize-logs-to-get-more-value-from-data/. [Accessed: 25-May-2022].

[96] SentinelOne, “Log Visualization: How and Why To Bring Your Logs To Life | Scalyr,” Sep. 2020. [Online]. Available: https://www.sentinelone.com/blog/log-visualization-bring-your-logs-to-life/. [Accessed: 25-May-2022].

[97] H. Gall, M. Jazayeri, and C. Riva, “Visualizing software release histories: the use of color and third dimension,” in Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM’99). ’Software Maintenance for Business Change’ (Cat. No.99CB36360), 1999, pp. 99–108.

[98] T. Johann, M. Dick, S. Naumann, and E. Kern, “How to measure energy-efficiency of software: Metrics and measurement results,” in 2012 First International Workshop on Green and Sustainable Software (GREENS), 2012.

[99] M. Afaq and W.-C. Song, “sFlow-based resource utilization monitoring in clouds,” in 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2016, pp. 1–3.

[100] “MQC,” Model-engineers.com. [Online]. Available: https://model-engineers.com/en/quality-tools/mqc/. [Accessed: 25-May-2022].

[101] “Make requirements fully traceable with a traceability matrix - MATLAB & Simulink,” Mathworks.com. [Online]. Available: https://www.mathworks.com/help/slrequirements/ug/find-and-link-unlinked-requirements-with-a-traceability-matrix.html. [Accessed: 25-May-2022].

[102] “Review requirements verification status - MATLAB & Simulink - MathWorks Nordic,” Mathworks.com. [Online]. Available: https://se.mathworks.com/help/slrequirements/ug/review-requirement-verification-status-metrics-data.html. [Accessed: 25-May-2022].

[103] “Collect and explore metric data by using the metrics dashboard - MATLAB & Simulink - MathWorks Nordic,” Mathworks.com. [Online]. Available: https://se.mathworks.com/help/slcheck/ug/collect-and-explore-metric-data-by-using-metrics-dashboard.html. [Accessed: 25-May-2022].

[104] “Explore status and quality of testing activities using the model testing dashboard - MATLAB & Simulink - MathWorks Nordic,” Mathworks.com. [Online]. Available: https://se.mathworks.com/help/slcheck/ug/model-testing-dashboard.html. [Accessed: 25-May-2022].

[105] L. Soft, “Visualize Code with Software Architecture Diagrams,” Lexington Soft, 10-Dec-2020. [Online]. Available: https://www.lexingtonsoft.com/visualize-code-with-software-architecture-diagrams/. [Accessed: 25-May-2022].

[106] “Hello2morrow - Sonargraph architect,” Hello2morrow.com. [Online]. Available: https://www.hello2morrow.com/products/sonargraph/architect9. [Accessed: 25-May-2022].

[107] B. Merkle, “Stop the software architecture erosion: Building better software systems,” in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH ’10, 2010.

[108] L. Softwarearchitekturen, Fg-arc.gi.de. [Online]. Available: https://fg-arc.gi.de/fileadmin/FG/ARC/Architekturen2015/Langlebige_Softwarearchitekturen.pdf. [Accessed: 25-May-2022].

[109] J. Malnati, “Jacopo Malnati, X-Ray 1.0.4.1,” Usi.ch. [Online]. Available: https://xray.inf.usi.ch/xray.php. [Accessed: 25-May-2022].

[110] L. Soft, “Visualize Code with Software Architecture Diagrams,” Lexington Soft, 10-Dec-2020. [Online]. Available: https://www.lexingtonsoft.com/visualize-code-with-software-architecture-diagrams/. [Accessed: 25-May-2022].

[111] “CodeChecker,” Readthedocs.io. [Online]. Available: https://codechecker.readthedocs.io/en/latest/. [Accessed: 25-May-2022].

[112] “Clang C language family frontend for LLVM,” Llvm.org. [Online]. Available: https://clang.llvm.org/. [Accessed: 25-May-2022].

[113] “Software quality visualization - tech Debt | CodeScene,” Codescene.com. [Online]. Available: https://codescene.com/. [Accessed: 25-May-2022].

[114] “CodeScene, “CodeScene,” Codescene.io. [Online]. Available: https://codescene.io/projects/168/jobs/365254/results. [Accessed: 25-May-2022].

[115] R. Lincke, J. Lundberg, και W. Löwe, ‘Comparing software metrics tools’, στο Proceedings of the 2008 international symposium on Software testing and analysis, 2008, σσ. 131–142.

[116] LeanIX GmbH, “DORA metrics,” Leanix.net. [Online]. Available: https://www.leanix.net/en/wiki/vsm/dora-metrics. [Accessed: 25-May-2022].

[117] S. De Simone, “SPACE, a new framework to understand and measure developer productivity,” InfoQ, 18-Mar-2021. [Online]. Available: https://www.infoq.com/news/2021/03/space-developer-productivity. [Accessed: 25-May-2022].

[118] “Developer workflow optimization,” LinearB, 20-Aug-2019. [Online]. Available: https://linearb.io. [Accessed: 25-May-2022].

[119] “Haystack - Analytics for Engineering Leaders,” Usehaystack.io. [Online]. Available: https://www.usehaystack.io. [Accessed: 25-May-2022].

[120] “Code insights,” Swarmia.com. [Online]. Available: https://www.swarmia.com/product/code/. [Accessed: 25-May-2022].

[121] K. Dreef, V. K. Palepu, and J. A. Jones, “Global overviews of granular test coverage with matrix visualizations,” in 2021 Working Conference on Software Visualization (VISSOFT), 2021

[122] S. Martinez-Fernandez et al., “Continuously assessing and improving software quality with software analytics tools: A case study,” IEEE Access, vol. 7, pp. 68219–68239, 2019.

[123] “Chart.js | Open source HTML5 Charts for your website.” [Online]. Available: https://www.chartjs.org/. [Accessed: 28-May-2022]

[124] “Easiest JavaScript charting library for web & mobile.” [Online]. Available: https://www.fusioncharts.com. [Accessed: 28-May-2022]

[125] S. Nadhani and P. Nadhani, FusionCharts Beginner’s Guide: The Official Guide for FusionCharts Suite. Packt Publishing Ltd, 2012.

[126] P. Pokorn`y and K. Stokl´aska, “Chart visualization of large data amount,” in Computer Science On-line Conference. Springer, 2017, pp. 460–468.

[127] “JavaScript Charts in one powerful declarative library | ZingChart.” [Online]. Available: https://www.zingchart.com/. [Accessed: 28-May-2022]

[128] M. Bastian, S. Heymann, and M. Jacomy, “Gephi: an open source software for exploring and manipulating networks,” in Proceedings of the international AAAI conference on web and social media, vol. 3, no. 1, 2009, pp. 361–362.

[129] “NodeBox Live.” [Online]. Available: https://nodebox.live/. [Accessed: 28-May-2022]

[130] “rstudio/cheatsheets,” May 2022, originaldate: 2017-01-04T18:04:05Z. [Online]. Available: https://github.com/rstudio/cheatsheets/blob/45c1e642468695830fd8b724587ccfe8901e2185/data-visualization-2.1.pdf. [Accessed: 28-May-2022]

[131] “ggplot2 Version of Figures in “Lattice: Multivariate Data Visualization with R” (Final Part) | Learning R.” [Online]. Available: https://learnr.wordpress.com/2009/08/26/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-final-part/. [Accessed: 28-May-2022]

[132] “Welcome to Processing!” [Online]. Available: https://processing.org/. [Accessed: 28-May-2022]

[133] “JpGraph - Most powerful PHP-driven charts.” [Online]. Available: https://jpgraph.net/. [Accessed: 28-May-2022]

[134] S. P. Reiss, “An overview of bloom,” in Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2001, pp. 2–5.

[135] S. Boccuzzo and H. Gall, “Cocoviz: Towards cognitive software visualizations,” in 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis. IEEE, 2007, pp. 72–79.

[136] K. Maruyama, T. Omori, and S. Hayashi, “A visualization tool recording historical data of program comprehension tasks,” in Proceedings of the 22nd International Conference on Program Comprehension, 2014, pp. 207–211.

[137] J. Rilling and A. Seffah, “Moose-a task-driven program comprehension environment,” in 25th Annual International Computer Software and Applications Conference. COMPSAC 2001. IEEE, 2001, pp. 77–84.

[138] J. Rilling, A. Seffah, and C. Bouthlier, “The concept project-applying source code analysis to reduce information complexity of static and dynamic visualization techniques,” in Proceedings First International Workshop on Visualizing Software for Understanding and Analysis. IEEE, 2002, pp. 90–99.

[139] J. Rilling and S. P. Mudur, “3d visualization techniques to support slicing based program comprehension,” Computers & Graphics, vol. 29, no. 3, pp. 311–329, 2005.

[140] P. Caserta and O. Zendra, “Visualization of the static aspects of software: A survey,” IEEE transactions on visualization and computer graphics, vol. 17, no. 7, pp. 913–933, 2010.

[141] H. L¨u and J. Fogarty, “Cascaded treemaps: examining the visibility and stability of structure in treemaps,” in Proceedings of graphics interface 2008, 2008, pp. 259–266.

[142] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, “Shrimp views: an interactive environment for information visualization and navigation,” in CHI’02 Extended Abstracts on Human Factors in Computing Systems, 2002, pp. 520–521.

[143] H. Padda, A. Seffah, and S. Mudur, “Investigating the comprehension support for effective visualization tools–a case study,” in 2009 Second International Conferences on Advances in Computer-Human Interactions. IEEE, 2009, pp. 283–288.

[144] G. d. F. Carneiro, R. C. Magnavita, E. Spinola, F. Spinola, and M. Mendonca, “Evaluating the usefulness of software visualization in supporting software comprehension activities,” in Proceedings of the Second ACMIEEE international symposium on Empirical software engineering and measurement, 2008, pp. 276–278.

[145] G. d. F. Carneiro, M. Mendonca, and R. Magnavita, “An experimental platform to characterize software comprehension activities supported by visualization,” in 2009 31st International Conference on Software Engineering-Companion Volume. IEEE, 2009, pp. 441–442.

[146] M. Lungu and M. Lanza, “Exploring inter-module relationships in evolving software systems,” in 11th European Conference on Software Maintenance and Reengineering (CSMR’07). IEEE, 2007, pp. 91–102.

[147] D. Porto, M. Mendonca, and S. Fabbri, “Crista: A tool to support code comprehension based on visualization and reading technique,” in 2009 IEEE 17th International Conference on Program Comprehension. IEEE, 2009, pp. 285–286.

[148] MES Model Examiner® (MXAM). [Online]. Available: https://model-engineers.com/de/quality-tools/mxam/ [Accessed: 30-May-2022].

[149] B. A. Aseniero, T. Wun, D. Ledo, G. Ruhe, A. Tang, S. Carpendale, "STRATOS: Using Visualization to Support Decisions in Strategic Software Release Planning", The 33rd Annual ACM Conference on Human Factors in Computing Systems, ACM, 2015.