A Prototype for Comparing Customized Software Projects

MASTER THESIS

Manik Malik

Submitted on 28 May 2025

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

Supervisor:
Thomas Woltner
Prof. Dr. Dirk Riehle, M.B.A.

Declaration of Originality

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

Erlangen, 28 May 2025

License

Erlangen, 28 May 2025

Abstract

In software engineering, issue management systems serve as an important indicator for tracking the progress and evolution of a project. The metadata generated by such tools can be used to create quantitative metrics that can help developers and managers guiding the project successfully. Furthermore, these metrics can be used as indicators to compare software engineering projects. However, even though development metrics exist, software engineering projects are highly heterogeneous in their use of development tools. Consequently, the same metrics cannot reliably be defined for different projects, making comparisons challenging or impossible. In this thesis, using a design science approach, we present a novel approach for comparing software engineering projects using two commonly used issue tracking systems GitHub Issues and Jira. Lastly, we use a dataset of different open source projects to demonstrate and evaluate our solution.

Contents

List of Figures

List of Tables

Acronyms

VCS Version Control System

ITS Issue Tracking Softwares

1 Introduction

"Software metrics play a key role in good software engineering. Measurement is used to assess situations, track progress, evaluate effectiveness, etc." (Fenton & Bieman, 2014). The ultimate goal of software metrics is to provide continuous insight into the state of software projects and steer them accordingly. The development metrics and their extraction processes have been a key research topic for many decades.

Version Control System (VCS) and Issue Tracking Softwares (ITS) like Git and Jira represent a foundational aspect of software development. These systems offer developers a platform for collaboration, codebase integrity, and change tracking. This software holds a lot of metadata that can be used to calculate software metrics and facilitate further comparisons.

Many research papers and articles like (Kupiainen et al., 2015) explain the uses of such software development metrics and outline some of the most important metrics and how they contribute towards practical applications like productivity measurements. However, every software development project is unique, and no single approach to tooling and choice of metrics exists. Each software project shares challenges related to technology, people, and timelines. (Bhardwaj & Rana, 2016), in practice, even similar software projects within the same organisation use different development approaches. For instance, the choice between Github and Gitlab as VCS or Jira Issues and Github issues as the ITS can result in different metrics.

Recent studies in software engineering have emphasized the increasing demand for cross-project metric harmonization and metadata standardization, especially within organizations managing multiple repositories and tools. Maalej et al. (2015) demonstrated how inconsistencies in issue tracking terminology can hinder bug report classification across tools. Meanwhile, Herzig et al. (2013) discussed the challenges of mapping developer activities across multiple data sources, highlighting the difficulty of unifying ITS and VCS metadata. Other works like Munaiah et al. (2017) have also underlined the importance of metadata curation for effective project comparison, especially in empirical software engineering. This

thesis contributes to that line of research by providing a unified framework to extract, standardize, and compare software development metrics across diverse development ecosystems, with an emphasis on flexibility and user control.

Consequently, comparing the development metrics of different software projects can be a challenge. This topic has yet to be outlined in previous research, which is a threat to analyzing topics such as productivity and efficiency. This thesis builds on the existing research by focusing on common metrics used between similar types of software development projects and gives a novel solution to compare those metrics and, as an extension, those development processes.

All of these points align well with the context of this work, which addresses the organizational need to analyze and visualize software project issue data in a flexible manner. In detail, this thesis follows the framework proposed by Peffers et al., 2007. This is reflected in the structure of the thesis as such:

- Chapter 2 reviews relevant literature in the areas of issue tracking, metric analysis, and custom query execution frameworks, and outlines the problem this work aims to address.
- Chapter 3 defines the design objectives for the system, based on the identified problem and the gaps observed in existing approaches.
- Chapter 4 presents the proposed solution design, including the architecture of the data ingestion pipeline extension, the structure of the query language, and the planned frontend functionality.
- Chapter 5 describes the technical implementation of the solution, detailing the backend components built, the REST API developed, and the frontend implementation.
- Chapter 6 demonstrates the practical application of the system, showcasing how users can define metric queries and explore the results across multiple software projects.
- Chapter 7 evaluates the system in terms of its usability, flexibility, and alignment with the initial design objectives.
- Chapter 8 concludes the thesis by summarizing the contributions, limitations, and potential directions for future research and development.

2 Problem Identification

As presented in the previous section, a thorough comparison between software project metrics can lead to a one-on-one comparison of productivity and efficiency between software development teams. But doing so is not an easy task. This section will define key terminology, describe potential problems in detail, and give background information by analyzing previous literature.

2.1 Background

Comparing software projects using their metrics, even for similar types of projects, is a challenging task. The difference comes based on the choice of version control software or the issue tracking software that the team decided to use. It is further complicated by the type of software development methodology being used (Agile, Waterfall, etc), which introduces different metadata into the development process.

In addition to tool and process heterogeneity, project context, such as team size, domain complexity, and stakeholder involvement, plays a vital role in shaping development dynamics. Two teams using identical tools may produce completely different metric profiles due to contextual factors. Consequently, establishing equivalence between projects demands more than technical alignment—it requires an understanding of the surrounding human and organizational elements.

2.2 Software Development Approaches

Numerous articles and papers describe the different Software Development Approaches and give detailed comparisons. Scrum, waterfall, Spiral, V-Model, and the list goes on; in the paper, Saeed et al., 2019 21 different methodologies are compared, Chandra, 2015, giving a comparison of 7 of them. "Water-Scrum-Fall" is also a well-known buzzword that has been declared as a reality of agile in organizations by West et al., 2011. Theocharis et al., 2015 supports the claim and explains how Scrum has become the most popular agile method in organizations,

and that agile methods are adapted and combined with other processes.

Furthermore, the trend toward customized or hybrid development methodologies has become more pronounced in industry. Organizations often do not follow pure Agile or Waterfall but instead implement tailored processes that borrow elements from multiple paradigms. This has led to a proliferation of highly context-specific development models. The academic term for this practice is “method tailoring” or “situational method engineering” Henderson-Sellers and Ralyté, 2003, which complicates the reuse and comparison of project data across teams.

2.3 Version Control and Issue Tracking

Despite there being several different Software Development methodologies, we can be certain that almost every software development project uses some sort of version control software for maintaining their code base and an Issue Tracking software ITS to track progress, bugs, features, and other development-related tasks.
″Version control, which is considered to be a very important component of the software development life cycle, which can also be called revision control or source code control, is the management of changes to documents, computer programs, large websites, and other collections of information. Changes can be identified in some ways, such as "revision number", "revision level" and "revision" etc (Deepa et al., 2020)

In software engineering, revision control’s primary aim is to provide control over changes made to source code. Its functionalities can also be enhanced, and it can be used to make electronic documentation, too. (Deepa et al., 2020)

Issue tracking software is a tool used to report, manage, and resolve issues or tasks within a project or organization. It helps teams monitor progress, prioritize tasks, and streamline collaboration.

Jira is an ITS by Atlassian that many big organizations use in their day-to-day. Github and Gitlab are VCS that offer issue-tracking functionality as well.

Emerging tools are beginning to offer integrated development environments that combine version control, issue tracking, and continuous integration pipelines. For example, platforms like GitHub and GitLab blur the traditional boundaries between VCS and ITS by offering native issue management, CI/CD, and even project boards. This convergence raises questions about where to draw analytical boundaries and how to prevent metric duplication across overlapping tool functions.

2.4 Metadata Collection

ITS and VCS software are an integral part of the software development process, and collectively they generate a lot of metadata. VCS tools like Git and SVN contain commit information, file information, branch and merge data, tag, and release data. VCS metadata ensures traceability, enables efficient debugging, and facilitates collaboration by offering a clear history of changes.

ITS tools like Jira and GitHub issues generate metadata that revolves around the life cycle of tasks, bugs, and features. Information like task details, status and workflow, ownership, timestamp, priority, and dependencies supports project tracking, workload distribution, and performance evaluation while ensuring issues are managed systematically.

This metadata is the core of software development analysis. While git metadata is largely similar, different projects use different development methodologies, which will result in different metadata even from similar projects using the same ITS tools. This makes a comparison of projects like solving a puzzle with missing pieces.

A key challenge in metadata-driven comparison is the lack of a shared schema or ontology. Projects often customize fields in ITS tools (e.g., custom issue types or statuses in Jira), making direct mapping across systems impractical. This calls for a normalization layer, an intermediate representation that abstracts and aligns core metadata elements while preserving semantic richness. Research into software process ontologies and schema mapping frameworks is highly relevant in this regard Wermelinger and Yu, 2005.

2.5 Problem Statement

Given the diversity in tools, practices, and data structures, it becomes evident that:

> *There is no standardized framework for comparing software project metrics across heterogeneous development environments.*

This leads to critical research and engineering challenges:

- How can metadata from disparate tools (e.g., Jira, GitHub) be unified for analysis?
- Can we define a flexible schema or abstraction that accommodates diverse workflows?
- What level of detail is required for meaningful comparisons?

- How can visual tools support decision-making based on cross-project metrics?

This gap suggests an urgent need for a middleware or standardized framework that supports automated classification, alignment, and visualization of project metadata from disparate sources. Such a system could enable longitudinal comparisons within organizations or benchmark studies across the software engineering industry. The ability to automatically interpret and normalize metrics could also support real-time dashboards for project health monitoring.

2.6 Research Motivation

Addressing this gap has direct practical and academic value. From an academic perspective, this contributes to the field of empirical software engineering by offering a cross-tool, cross-process comparative framework. From a practical angle, it empowers decision-makers with standardized metrics across teams, enabling better resource allocation, process improvement, and transparency.

From a strategic perspective, resolving this issue aligns with organizational goals such as DevOps maturity, engineering productivity metrics, and data-driven retrospectives. Recent literature has emphasized the growing importance of Developer Experience (DevEx) and Engineering Effectiveness metrics Forsgren et al., 2018, which depend on unified data inputs to be meaningful. This work contributes to that evolving space by proposing a more consistent basis for such measurements.

By developing a tool-supported method for extracting, unifying, and visualizing software metrics, this thesis aims to lay the groundwork for standardized software analytics across diverse ecosystems.

3 Objective Definition

As discovered in the previous chapters, there is no universally accepted solution to compare software projects based on their issue-tracking information. Different development approaches or software choices can change the types of measurements generated. As a result, we aim to design a tool that can isolate relevant metrics from each project and normalize their values to enable systematic comparison.

Firstly, there are observable measurement differences even among projects of a similar nature. To ensure feasibility, the focus is on identifying common metrics in projects that use publicly accessible GitHub[1] repositories. This provides a realistic and replicable foundation for the comparison process.

Secondly, projects are categorized based on the issue tracking systems (ITS) they use, namely Jira[2] Issues or GitHub Issues. This division allows the system to support cross-platform analysis and demonstrates the tool's ability to operate independently of specific ITS formats.

Thirdly, the tool must allow for extensibility. Users should be able to provide contextual information to interpret the extracted metrics accurately and define additional fields where required. This ensures the adaptability of the tool across different datasets and software environments.

Lastly, the tool should offer a user interface that allows users to input metric configurations and interact with the calculated results, including comparative visualizations.

To summarize, the customized software project comparison tool presented in this thesis must fulfill the following criteria:

- Data source used in the tool should be limited to similar open source projects but varied in terms of their ITS to demonstrate generalised comparison.

[1] GitHub - https://github.com

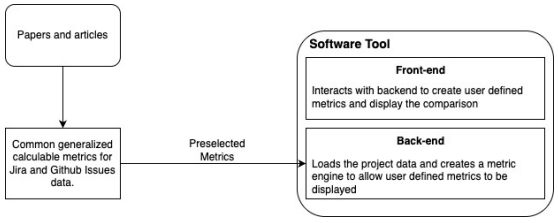
[2] Jira - https://www.atlassian.com/software/jira

- The tool should be able to ingest GitHub and Jira Issues data from any and all projects and extract useful information from the data.
- The tool should allow user to create their metrics.
- The tool should allow users to view and interact with the metric information created.
- Finally, the tool will demonstrate the comparison of some common metrics defined in the literature.

4 Solution Design

As the project comparison tool depends on the data gathered from ITS to create some selected software development metrics and on the user input to handle unknown or custom fields, we will divide the solution process into three parts. Firstly, we will analyze a selection of papers and articles along with the basic data fields available in Jira and GitHub issues to narrow down a list of metrics that can be calculated for Jira and GitHub-based projects. Secondly, we will design an extension of the existing MecoIS system to ingest data and calculate metrics with the added functionality of user input. Lastly, we will design a front-end application to interact with the data ingested into the platform.

Figure 4.1: Overview of the Solution Design Architecture



4.1 Metric Selection

The selection of software development metrics was a critical step to ensure meaningful, actionable, and comparable insights derived from issue tracking systems such as Jira and GitHub. The metric selection process followed a structured, multi-stage approach, focusing on data availability, analytical value, and alignment with project objectives Basili et al., 1994; Dias et al., 2020.

4.1.1 Data Source Analysis

The process began with an analysis of the data structures and fields available in Jira and GitHub. Jira provides structured data through its support for Agile constructs such as story points, epics, sprints, and custom workflows Atlassian,

2024; Gandomani et al., 2018. GitHub, on the other hand, emphasizes collaborative software development through issues, pull requests, reviews, and reactions. A comparative mapping of fields (e.g., issue creation date, resolution date, assignees, labels, status) was created to identify common and platform-specific data points.

4.1.2 Metric Categorization

Based on the field mapping, metrics were grouped into three categories:

- Common Metrics: Applicable to both Jira and GitHub, derived from shared fields such as timestamps and issue statuses. These metrics enable consistent tracking across platforms.
- Jira-Specific Metrics: Leveraging Agile-specific features such as story points, sprints, and epics to evaluate planning accuracy and delivery velocity.
- GitHub-Specific Metrics: Focused on development workflow and code collaboration, utilizing pull request metadata, review timestamps, and contributor statistics.

4.1.3 Normalization Considerations

To support valid comparisons across teams, projects, and time periods, normalization techniques were evaluated and applied where necessary. Metrics like throughput, cycle time, and PR lead time were normalized per developer, per period, or per unit of work (e.g., story points or LOC) to control for team size, workload, and task complexity Mockus et al., 2002; Spinellis, 2016.

4.1.4 Trend Analysis

An additional criterion for metric selection was their suitability for trend analysis and visualization. Metrics were prioritized if they could support longitudinal tracking through line graphs, bar charts, or burndown charts, facilitating continuous improvement and empirical evaluation of process changes.

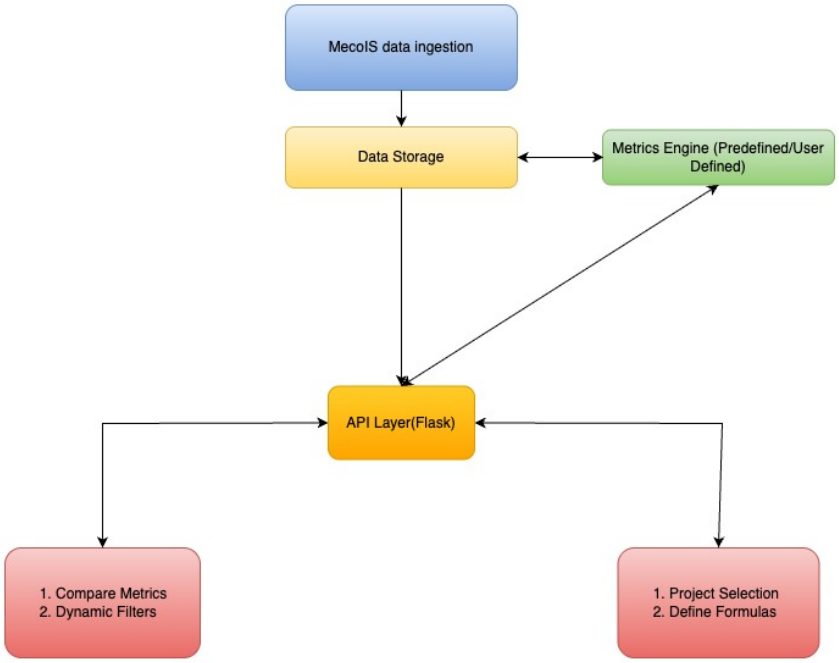
Table 4.1: Final Selected Software Development Metrics

Category	Metric Name	Description
Common	Lead Time	Time from issue creation to resolution
	Cycle Time	Time from work started to completion
	Throughput	Number of issues closed per period
	Issue Age	Age of currently open issues
	Resolution Rate	Ratio of closed to total issues
	Avg Time to Close	Mean duration from issue creation to closure
	Issues by Assignee	Workload distribution by team member
	Open vs Closed Ratio	Ratio of open to closed issues
Jira-Specific	Sprint Velocity	Story points completed per sprint
	Epic Progress	Completion percentage of epics
	Burndown Rate	Rate of work completion within sprint timeline
	Time in Status	Average time issues spent in each status
GitHub-Specific	PR Lead Time	Time from PR creation to merge
	Avg PR Review Time	Average time taken to review pull requests
	Comments per Issue	Average number of comments per issue
	Reopen Rate	Ratio of reopened to closed issues
	Contributor Activity	Number of issues and PRs per contributor

4.2 Software Tool

Having established some predefined metrics that would act as the initial point of comparison for our projects, we now move to describe our software tool. As illustrated in Figure 4.2, the software tool is not one single entity but consists of multiple different modules. The tool is a web application with a frontend and a backend. We will start with the modules present in the backend, we will explain how the data is ingested into the MecoIS system, then proceed to explain the functioning of the metric engine. Lastly, we will focus on the front end and explain the different functionalities that the user will have access to.

Figure 4.2: Overview of the Software Design Flow



4.2.1 Backend

Now we present the design of a backend system that extends the MecoIS data ingestion framework to support the integration and analytical querying of issue-tracking data from GitHub and Jira. The core innovation in the system is the development of a custom query language executor that enables users to construct structured analytical queries through a frontend interface. These queries are expressed in a predefined JSON schema, which the backend interprets and executes against issue data stored in the MecoIS data lake.

Design Goals

The following design goals were established for the backend architecture:

- **Extensibility:** The system must allow for future integration of additional data sources and analytical functions without significant architectural changes.
- **Modularity:** Components should be loosely coupled and independently maintainable.
- **Performance:** The system should efficiently process large-scale issue data and support complex analytical queries Fowler, 2002; Garlan and Shaw,

2000.

Core Components

The backend system is composed of the following core components, each fulfilling a distinct role in the query execution pipeline.

Query Parser and Validator

This component is responsible for parsing the JSON-based user query into an internal representation, suitable for planning and execution. The parser ensures syntactic correctness and performs semantic validation, verifying that fields, metrics, and operators are aligned with the underlying data schema.

Query Planner and Optimizer

Once validated, the query is passed to the planning module, which generates an execution plan. The planner organizes operations such as filtering, aggregation, and grouping in an optimized sequence. Optimization strategies include predicate pushdown, field projection pruning, and efficient ordering of transformation stages to reduce data volume early in the pipeline.

Execution Engine

The execution engine interprets the plan and performs the necessary data transformations. This includes loading the relevant dataset from the data lake, applying the filters, computing aggregations, and assembling the result set according to the query specification. The engine is designed to work with large-scale datasets and can leverage distributed processing frameworks where required.

Data Source Connectors

To support ingestion and querying of heterogeneous sources (e.g., GitHub and Jira), the system includes connectors that abstract the underlying data storage details. These connectors facilitate schema mapping, type conversions, and data access optimization across different formats within the MecoIS data lake.

API Layer

An API interface exposes query submission and result retrieval functionality to the front end. It ensures secure access and input sanitization and provides standardized responses to the frontend form-based query builder.

Data Flow and Processing Pipeline

The overall data flow in the backend system can be summarized in the following stages:

1. **Query Submission:** The frontend sends a structured query in JSON format to the backend via the API.
2. **Parsing and Validation:** The backend parses the query and validates it against the expected schema and business rules.
3. **Planning and Optimization:** An execution plan is generated and optimized based on query complexity and data characteristics.
4. **Execution:** The engine reads data from the data lake and applies transformations according to the plan.
5. **Response Generation:** The results are compiled and returned to the frontend in a format suitable for visualization or export.

Design Principles and Considerations

Several architectural principles guided the design:

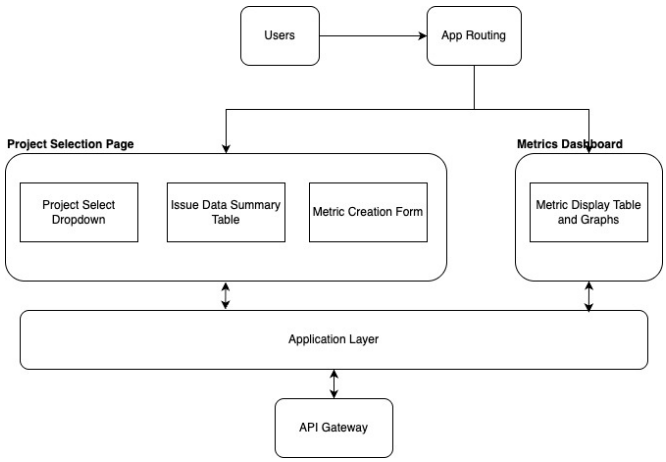
- **Separation of Concerns:** Parsing, planning, and execution are separated into distinct modules to improve clarity and maintainability.
- **Scalability:** The architecture supports horizontal scaling through parallel processing capabilities of the execution engine.
- **Loose Coupling:** Components interact through well-defined interfaces, allowing independent development and testing.
- **Configurability:** The system supports external configuration for data source definitions, query constraints, and resource management.

Extensibility and Maintenance

The backend is designed to be extensible with minimal changes to core logic. New metrics, data sources, or transformation functions can be added as plugins or modules. This modular approach ensures long-term maintainability and supports future research or feature expansion within the MecoIS platform.

4.2.2 Frontend

Figure 4.3: Overview of the Frontend Architecture



The frontend interface plays a critical role in enabling users to construct and explore analytical queries over issue-tracking data from GitHub and Jira. The frontend is not merely a user interface; it is an essential instantiation of the artifact that supports the core objective of enhancing user-driven, flexible, and comparative analysis across projects.

The design of the frontend adheres to the principles of relevance and usability Nielsen, 1994; Shneiderman et al., 2016, ensuring that stakeholders with varying technical expertise can construct valid queries without writing code.

Page 1: Project Selection and Metric Definition

The first page of the interface addresses the need for query flexibility and data exploration by offering the following features:

- **Project Selection:** Users can choose one or more projects from a dropdown menu. This selection filters the available dataset, ensuring relevance to the user's analysis goals.
- **Column Exploration:** Once a project is selected, the system displays the available data columns along with up to five distinct values for each. This preview provides immediate insight into the data structure and possible grouping or filtering dimensions, supporting informed query construction.
- **Metric Editor:** A dedicated form allows users to define, edit, or remove metrics. Each metric can be configured by specifying:
  - The descriptive `name` of the metric

  - The aggregation `function` (e.g., `count`, `avg`)
  - The corresponding data `field`

  This design decision supports a key requirement identified in the problem space: enabling non-technical users to define domain-relevant performance metrics.

- **Query Enhancements:** Additional inputs allow users to define filters, grouping levels, sorting criteria, and other advanced options. These inputs are structured into a JSON query that conforms to the backend execution format.

By enabling users to iteratively refine their analytical goals through an interactive interface, this page directly addresses the research objective of increasing analytical accessibility and configurability.

Page 2: Metric Visualization and Project Comparison

The second page of the frontend focuses on delivering insights and supporting comparative evaluation. Its design features include:

- **Tabular Output:** Computed metrics are presented in a table, allowing for precise inspection of numerical values across selected dimensions.
- **Graphical Visualization:** One or more interactive charts (e.g., bar charts, line graphs) are rendered based on the grouping and metric dimensions specified in the query. These visualizations enable users to identify trends and patterns in the data.
- **Multi-Project Comparison:** The interface supports the simultaneous display of metrics from two or more projects. This capability was designed to fulfill a key requirement uncovered during problem analysis: the need to benchmark or compare project performance on issues such as resolution time, open issue counts, or issue priority distribution.
- **Interactive Feedback:** Graphs and tables dynamically update based on user-defined inputs, providing a responsive experience that supports exploratory analysis.

5 Implementation

This section goes over the implementation of the solution described in Chapter 4. Firstly, we will cover different tools that were used during the development process. Secondly, we will describe the data sources and the data collection process for GitHub and Jira Issues. Thirdly, we describe the process of the custom query executor. Lastly, we will go through the front-end implementation.

5.1 Used Tools

For the implementation of our solution, we require a variety of software tools. As the backend is basically an extension of the existing MecoIS codebase, we would take a look at all the tools used to implement that without going into much detail.

5.2 Used Datasets

As previously mentioned, the initial objective of the tool is to be able to perform comparative analysis. For that purpose, we looked for open source projects to import their ITS data into the MecoIS system. The initial focus was to get projects that use Jira issues, as Jira is mostly used in closed-source company projects. We tried to find similar projects, with one using Jira and the other GitHub issues to better demonstrate the tool´s abilities.

Table 5.1: Open Source Projects and Their Metadata

Project	Work Domain	Code Repository	Issue Tracking System
Wildfly	Java Application Servers	GitHub	Jira Issues
Payara	Java Application Servers	GitHub	GitHub Issues
Neo4j	NoSQL Database	GitHub	GitHub Issues
Riak	NoSQL Database	GitHub	GitHub Issues
Moodle	Online Learning Platform	GitHub	Jira Issues
Sakai Project	Online Learning Platform	GitHub	Jira Issues

5.3 Data Ingestion

The first step was to ingest the ITS data into the MecoIS system. We extended the existing data pipelines to include GitHub and Jira Issues.

5.3.1 GitHub Issues

The ingestion starts by first getting the issue details from the respective project API endpoint. We need to set up the authorization in the file `api_tokens.py` by giving the JWT, owner, and repository name to the client params. We also add the project path and related information in the `pipeline-config.yml` file.

The API request is sent to the `/repos/{repo_name}/issues?state=all` endpoint, and the data is ingested in blocks of 100 elements per API request. The result is stored in a JSON file in the output folder under the project name folder.

The issues JSON file is then loaded into the bronze and silver tables using PySpark. Before doing that, we need to preprocess the JSON file to separate the label fields. Unlike Jira Issues, GitHub does not have custom fields or a separate status field. The important information is passed through labels. In the JSON response of GitHub Issues, labels come as a list of objects, we need to extract the individual label for each issue and represent it as a stand alone column. Below is the example payload, followed by how the separated labels would look for this example.

Listing 5.1: Example Label Payload

```
{
    "labels": [
      {
        "id": 19053963,
        "node_id": "MDU6TGFiZWwxOTA1Mzk2Mw==",
        "url": "https://api.github.com/repos/neo4j/neo4j/
           labels/bug",
        "name": "bug",
        "color": "fc2929",
        "default": true,
        "description": null
      },
      {
        "id": 647068085,
        "node_id": "MDU6TGFiZWw2NDcwNjgwODU=",
        "url": "https://api.github.com/repos/neo4j/neo4j/
           labels/team-kernel",
        "name": "team-kernel",
        "color": "0052cc",
        "default": false,
        "description": null
      }
    ],
}
```

Listing 5.2: Example Label Seperated

```
{
"label_bug": null,
"label_team-kernel": null
}
```

Now, as not all labels will be present in every issue, while adding the data into tables, the process does a union of all such columns; the issues with certain columns not present would have the data set as NULL for that column. This causes the main difference on how the data is queried for Jira and GitHub. As an example, for aggregating bugs, we would put the filter in Jira as `issuetype = "bug"`, whereas in GitHub the same query would look like `label_bug != null`.

The processed file is saved in the same location with a `_processed` prefix. The

processed file is then added to the bronze and silver tables. The nested structure is flattened, and the union of columns is performed in this step.

5.3.2 Jira Issues

The first step is similar to GitHub, we add details to the token and configuration file. The API call is made to the endpoint `rest/api/2/search`, with a JQL like `project= {project_key}` sent as a parameter.

In Jira, we have a big advantage of receiving the changelog information in the payload itself with a small change of setting `expand= "changelog"`, while sending the request. This gives us the historical information and the changes the issue has gone through. This is particularly important when extracting information related to status changes.

Similar to GitHub, in Jira, after receiving the initial payload and saving it to JSON, we explore the changelog to extract the "time in" information of each status. This opens doors to some very nice metrics calculation that gives an accurate idea of time spent in each stage of the software development process.

```
    def add_time_in_status(self, reduced_data):
        "Adds time stayed in each status"
        time_format = "%Y-%m-%dT%H:%M:%S.%f%z"
        for idx, data in enumerate(reduced_data):
            if 'histories' in data:
                status_change_timeline = {}
                sorted_histories = sorted(
                    data['histories'],
                    key=lambda x: datetime.strptime(x["created"],
                        time_format)
                )
                For history in sorted_histories:
                    if 'items' in history and len(history['items'
                        ]) > 0:
                        for item in history['items']:
                            If item.get('field') == 'status':
                                try:
                                    If not status_change_timeline
                                        :
                                        status_change_timeline[
                                            item.get('fromString')
                                            .lower()] = parser.
                                            parse(data.get('
                                            created'))
                                    status_change_timeline[item.
                                        get('toString').lower()] =
                                        parser.parse(history.get(
                                        'created'))
```

```
19
20                                  field = 'time_in_' + item.get
                                        ('fromString').lower().
                                        replace(" ", "_")
21                                  if item.get('fromString').
                                        lower() in
                                        status_change_timeline and
                                         item.get('toString').
                                        lower() in
                                        status_change_timeline:
22                                       time_from =
                                            status_change_timeline
                                            [item.get('fromString'
                                            ).lower()]
23                                       time_to =
                                            status_change_timeline
                                            [item.get('toString').
                                            lower()]
24                                       time_delta = (time_to -
                                            time_from).days
25                                       data.update({field.lower
                                            (): time_delta})
26                              except Exception as e:
27                                  print(str(e))
28              try:
29                  if status_change_timeline:
30                      events = list(status_change_timeline.
                            values())
31                      total_time_alive = (max(events) - min(
                            events)).days
32                      data.update({'total_time_alive':
                            total_time_alive})
33              except Exception as e:
34                  print(str(e))
35          del data["histories"]
36      self.next(reduced_data)
```

Listing 5.3: Python function to add time in status

After adding the time in the status information, we remove the historical information and continue with the data ingestion process in the same way as described in the section above.

5.4 Data Analytics

Now, after the data is ingested, we move to creating metrics and performing the analysis. As described in the objective and solution design, we need a way for users to define their own metrics as well as to facilitate the generalization of the tool. The custom fields and labels presented in the respective ITS systems cannot

be predicted for and the metrics that we have created are based on the universal fields offered by Jira and GitHub.

As described in the solution design, we have created a Custom Query Executor that parses and processes the user-submitted query object. The executor handles various components of the query, such as:

- **Metrics:** Aggregations such as `count`, `avg`, `sum`, `max`, and `min`.
- **Grouping:** One or more grouping fields with optional granularity (e.g., week, month for dates).
- **Filtering:** Pre-aggregation filters applied using operators like `=`, `>`, `<`, and `in`.
- **Having Clauses:** Post-aggregation filters applied to metric results.
- **Sorting and Limiting:** Ordering of results and limiting output size.

The query executor dynamically constructs a Spark SQL-like operation chain, based on the abstract structure defined in the query JSON. This approach abstracts complexity from the user and provides high flexibility in query specification.

The query format includes specifications for metrics, grouping dimensions, filters, having clauses, sort orders, and limits. For instance:

Listing 5.4: Example JSON Query

```
{
      "metrics": [
          {"name": "open issue count", "function": "count",
             "field": "*"}
          {"name": "avg_resolution", "function": "avg", "
             field": "resolution_time"}
      ],
      "groups": [
        {"field": "project", "alias": "project_name"},
        {"field": "priority"},
        {
          "field": "created_date",
          "granularity": "week",
          "alias": "creation_week"
        }
      ],
      "filters": [
        {"field": "status_name", "operator": "=", "value": "
           Open"}
      ],
      "having": [
```

```
19          {"metric": "issue_count", "operator": ">", "value":
                10}
20        ],
21        "sort": [
22          {"field": "creation_week", "order": "asc"},
23          {"field": "issue_count", "order": "desc"}
24        ],
25        "limit": 100
26  }
```

5.5 REST API

5.5.1 REST API Design and Implementation

The REST API for the system is implemented using the Flask web framework in Python. The API is designed to support data retrieval and configuration operations related to project issue metrics. The backend communicates with a Spark-based data processing layer and reads from and writes to local JSON files used for query storage.

CORS (Cross-Origin Resource Sharing) is enabled for local development to allow interaction between the frontend and backend.

The following endpoints are implemented:

- **GET /api/projects**

  Returns a list of project names by scanning the data lake directory structure.

- **GET /api/project_details?project_name=...**

  Returns available columns and a sample of distinct values from the dataset of the selected project. It uses the appropriate table path depending on the project (e.g., GitHub or Jira issues).

- **GET /api/metrics**

  Retrieves computed metric values for predefined projects. The server reads metric queries from JSON files and executes them using a Spark query engine.

- **GET /api/metric_config?project_name=...**

  Loads metric query configurations for a project from a JSON file.

- **POST /api/metric_config?project_name=...**

  Accepts metric query definitions via JSON payload and stores them in the respective project directory.

  Errors such as file not found, invalid JSON, or processing exceptions are handled with appropriate HTTP status codes (e.g., 400 or 500). The API supports structured communication between the frontend user interface and the backend query execution engine.

  Below is a simplified snippet of the Flask API code:

Listing 5.5: Excerpt from the Flask-based REST API

```
@app.route('/api/metric_config', methods=['POST'])
def post_metric_queries():
    project_name = request.args.get("project_name")
    data = request.get_json()
    path = f"/Users/manikmalik/.../{project_name}"
    try:
        with open(path + "/metrics.json", 'w') as f:
            json.dump(data, f, indent=2)
        return jsonify(data), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

5.6 Frontend

The frontend is implemented using React with TypeScript, enabling the development of a modular, type-safe, and responsive user interface. The frontend is composed of two primary views, corresponding to the query construction and result visualization stages.

5.6.1 Project and Metric Selection Interface

The first page allows users to:

- Select one or more projects from a dropdown populated via the `/projects` API.
- View a list of available columns for the selected projects, with five distinct sample values per column.
- Define metrics by specifying the name, aggregation function, and field.

- Configure grouping, filtering, and sorting options through dedicated forms.

  All user inputs are compiled into a structured JSON query that conforms to the expected schema of the backend query executor.

5.6.2 Results Visualization Interface

The second page is dedicated to displaying the output of the query execution. Key elements include:

- **Data Table:** A tabular representation of the returned metrics, supporting scrolling, sorting, and filtering.
- **Graphical Charts:** Visualization components (e.g., bar charts, line graphs) built using charting libraries such as Recharts or Chart.js, dynamically rendered based on the grouping and metric fields in the result.
- **Multi-Project Comparison:** The interface allows visual comparisons of metrics across two or more projects side-by-side.

  The interface ensures real-time feedback by updating charts and tables in response to user-modified query parameters.

6 Demonstration

Following the design and implementation of the tool, this chapter demonstrates the practical use of the developed system. The goal is to show how the proposed solution supports metric computation and visualization for issue tracking data sourced from open-source repositories. The demonstration is conducted using real data from GitHub and Jira issue trackers for selected software projects.

6.1 Project Selection Interface

The entry point of the system is a user interface that allows users to select one or more projects from a drop-down list. These projects are identified based on the folder structure within the data lake.

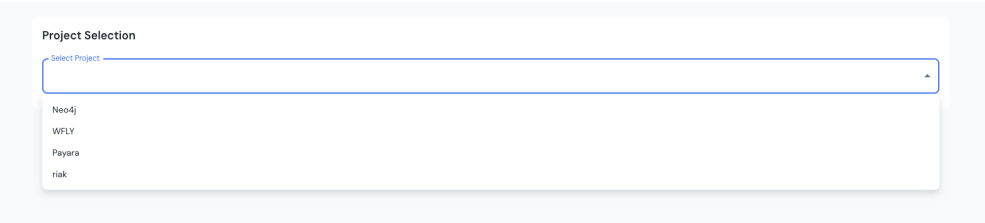


Figure 6.1: Project Selection Drop-down

Once a project is selected, the system loads the available columns from the dataset and displays up to five distinct values per column. This helps the user understand the data distribution and form relevant queries accordingly.

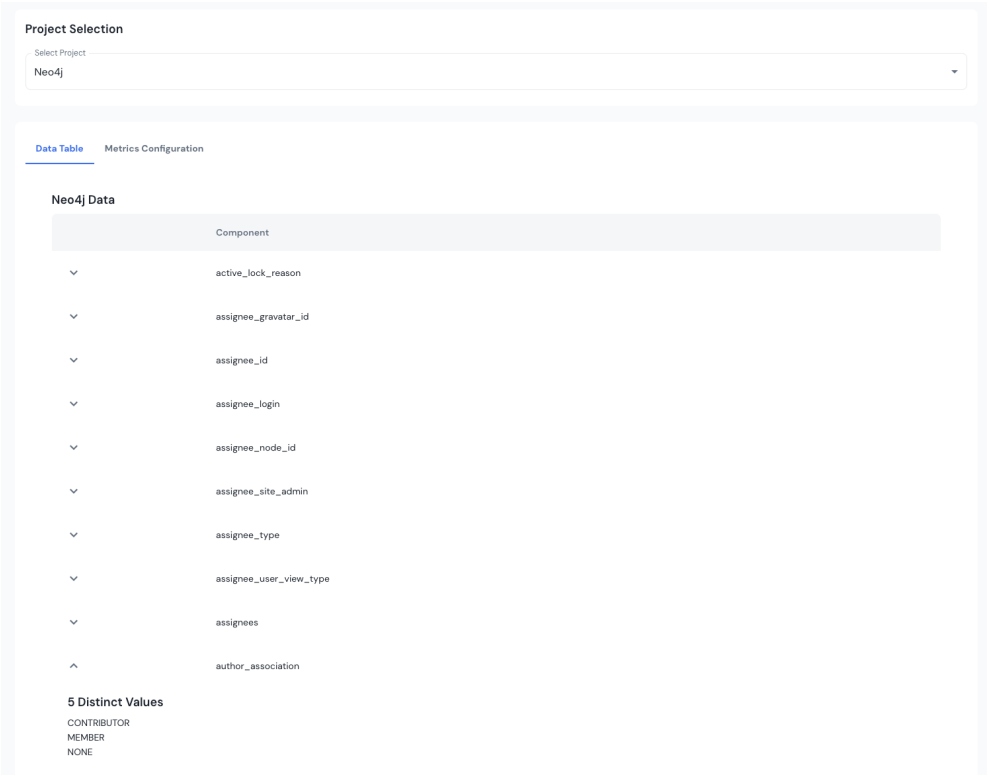


Figure 6.2: Project Data Table

This feature also supports scalability for environments where dozens of projects are stored and tracked. By relying on the underlying folder structure and automatic metadata indexing, the system ensures that project discovery does not require manual configuration. This makes it easier for organizations to onboard new projects without technical intervention. In future versions, this interface can be extended with filtering options based on organization, creation date, or technology stack.

6.2 Metric Configuration Interface

The metric configuration interface allows users to define and edit metrics using a structured form. Users can select aggregation functions (e.g., count, avg), specify the field of interest, group the data by specific dimensions, and apply filters. The user can also add new configurations for metric calculations based on the information provided in the project data table.

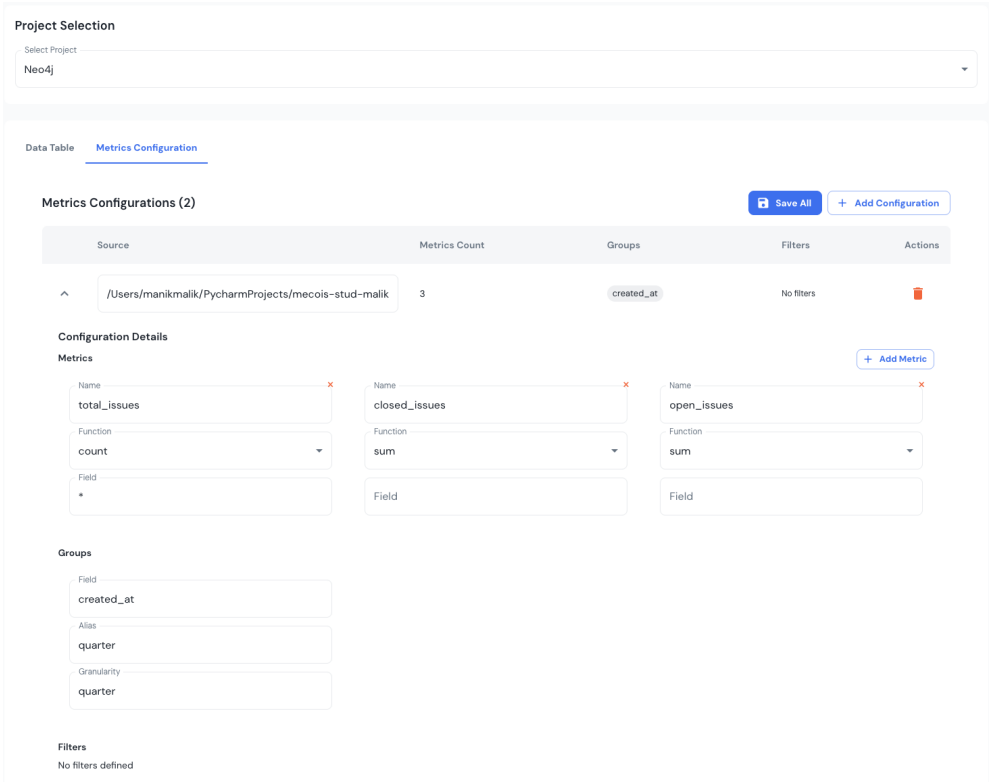


Figure 6.3: Metric Configuration Form

This form is internally converted to a custom query format, which is then passed to the backend for execution, where the custom query is parsed and applied to the DataFrame.

This abstraction enables non-technical users to interact with complex analytical backends. The form validation ensures semantic correctness (e.g., applying average only to numerical columns), thereby reducing user error. The design choice of using structured forms over free-text queries is aligned with usability studies, such as those by Ko et al. (2011), which found that constraint-based UIs improve accuracy in novice users of analytical systems.

6.3 Metric Visualization Interface

The final interface displays the computed metrics and visualizations. This view supports multi-project comparison by rendering side-by-side metric summaries and time-series charts.

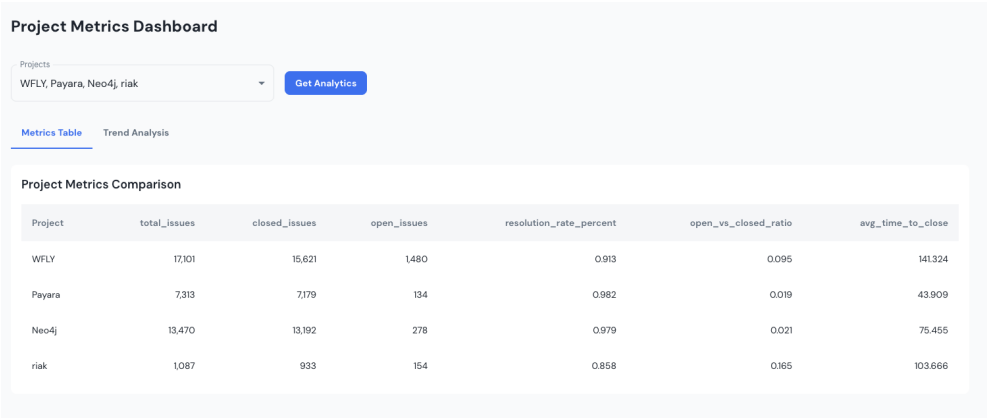


Figure 6.4: Metric Table Page

Users can assess metrics such as resolution rate percent, average time to close, and open/closed ratios across different projects.

The user can also view quarterly trends for each metric across all projects. This visual feedback aids in performance comparison and helps identify areas needing attention.

Additional interactivity is also built into the visual layer. Each metric tile can be expanded into a detailed view, which includes temporal trends and project-level breakdowns. Future enhancements may include export options to PDF/CSV and embedding widgets for dashboards. The visualization strategy draws on established principles of software analytics dashboards, as proposed in Buse and Zimmermann (2010).

6.4 Example Use Case: Jira vs GitHub

To validate the tool's applicability, we demonstrate the comparison of two project categories:

- **WildFly (WFLY):** Using Jira issue data
- **Neo4j:** Using GitHub issue data

The configured metrics include average resolution time and issue throughput. Figure 6.6 shows a sample of how metrics for both are displayed and compared.

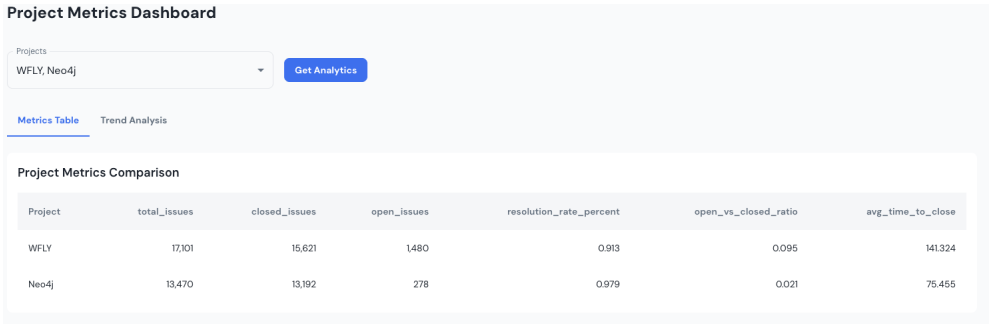


Figure 6.5: Comparison between Jira and GitHub-Based Projects

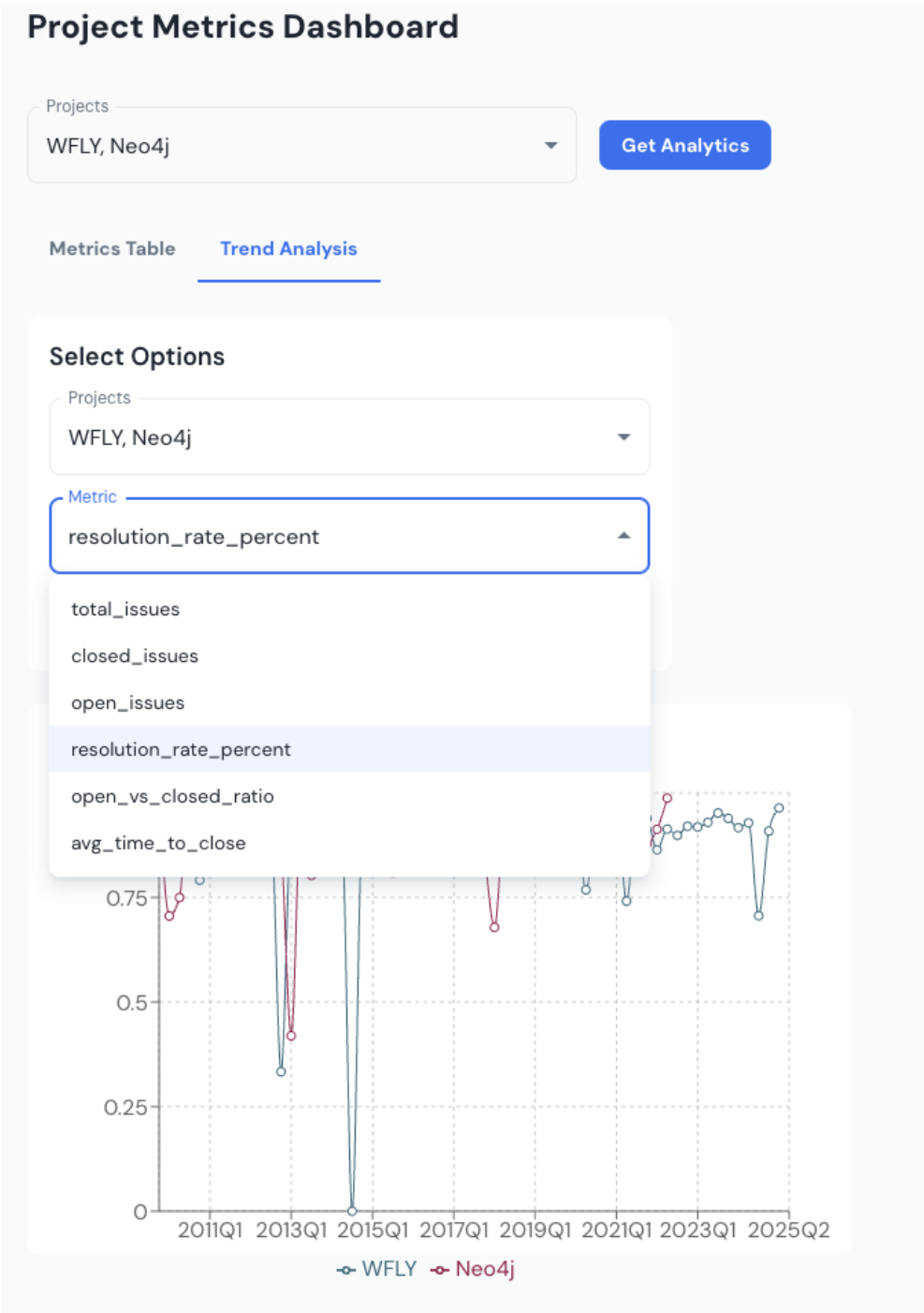


Figure 6.6: Trend Analysis between Jira and GitHub-Based Projects

This comparison illustrates the challenge and value of normalizing disparate data sources. While both projects report similar metadata, the semantic meaning of fields (e.g., "issue state", "closed date") can vary significantly. The system

bridges this semantic gap by applying internal mappings that align field names and types. This capability makes the system highly adaptable for cross-tool analytics, a limitation often cited in existing metric frameworks such as that of Zhang et al. (2013).

6.5 Summary

This demonstration shows how the system allows flexible metric definition, supports user-friendly interaction, and delivers comparative insights using real-world data. The separation of metric configuration and visualization enables clear traceability from input to output, fulfilling both usability and analytical requirements of the design.

Overall, the system demonstrates high usability and versatility in real-world environments. Its modular architecture also opens the door for future extensions, such as machine learning-based prediction models or integration with additional data sources like CI/CD tools and test coverage reports. The successful deployment of this tool lays the groundwork for more comprehensive DevOps analytics platforms.

7 Evaluation

In this chapter, we revisit the objectives defined in Chapter 3 and evaluate whether our solution, implementation, and demonstration fulfill the defined criteria. In addition, we discuss the limitations of our work and identify opportunities for future development.

7.1 Objective Criteria

For the evaluation, we reflect on the five criteria established in Chapter 3. Each criterion is compared with the outcomes described in the solution and implementation chapters.

Firstly, we required that the tool be applied to similar open source projects that differ in the type of issue tracking system (ITS) used. In our demonstration, we selected six open-source projects across three domains. These included projects using GitHub Issues and Jira Issues. By ingesting and analyzing both types of data, our tool demonstrates its ability to work across platforms.

⇒ **Criteria 1 is fulfilled.**

Secondly, we defined that the tool must ingest issue data from GitHub and Jira and extract relevant fields for metric computation. This was achieved by extending the existing ingestion pipeline of the MecoIS system. The updated backend supports reading from multiple structured sources and integrates them into a unified data lake format.

⇒ **Criteria 2 is fulfilled.**

Thirdly, we specified that the tool should allow users to define custom metrics. This functionality is realized through the introduction of a custom query configuration language. The front-end enables users to define metrics by selecting fields, aggregation functions, filters, and grouping parameters, which are parsed and executed by a backend query executor.

⇒ **Criteria 3 is fulfilled.**

Fourthly, we required that users be able to view and interact with the metric data. This is implemented via a two-page React interface: one for configuration and one for displaying metric outputs and visualizations. Users can view metric values, sort them, and explore visual trends across selected projects.

⇒ **Criteria 4 is fulfilled.**

Lastly, we expected the tool to demonstrate the use of common metrics from the literature to compare projects. During the demonstration, metrics such as average resolution time and issue volume over time were applied across projects. These results were visualized to show differences and trends, enabling direct comparison.

⇒ **Criteria 5 is fulfilled.**

7.2 Limitations

Although the tool meets all defined objectives, some limitations exist:

- The ingestion process currently assumes a fixed directory structure and pre-processed data. This may limit scalability unless adapted for more dynamic data sources.
- While the metric query format is flexible, it still requires understanding of the query schema, which may not be intuitive for non-technical users.
- Visualizations are basic and limited to predefined metric outputs. More advanced visual analytics or interactive dashboards could enhance usability.
- The current version does not support automated normalization or alignment of field names between different ITSs, which may limit extensibility to more diverse datasets.

Despite these limitations, the tool effectively demonstrates its ability to fulfill the design goals, providing a foundation for future refinement and broader application.

8 Conclusion

This thesis proposes a novel and generalized solution for comparing software engineering projects using metrics developed from the issue tracking information, irrespective of the underlying ITS.

To determine our approach, we first looked into the common pieces of information that are available through the two most used ITS, GitHub Issues and Jira. We then finalized some metrics that can be calculated from the available information.

We then moved on to ingest the data into the system and create a metric engine that acts as a custom query language executor, to allow users to select and create their metrics based on the information the ITS of their project provides. Finally, we created the user dashboard through which users can interact with the metrics, ITS data, and view the final comparison.

During our demonstration, we found that our approach is suitable for analyzing different open source projects that each use similar or different ITS. We were able to compare heterogeneous software projects using the most popular ITS GitHub Issues and Jira.

This allows us to analyze key metrics of these projects and generate meaningful insights into individual progress and bottlenecks. Thus, our system is flexible enough to be used with different kinds and scales of projects.

For future research, we suggest expanding the scope of our approach to include advanced analytics in the custom query language executor, which will allow for more individual and complicated metric calculations. Furthermore, there is a possibility to use LLMs to explain the differences in the projects to users who are not of a technical background and would not be able to extract meaningful insights from looking at metrics alone.

Appendices

A.1: Repositories used for demonstration

Table 1: Repositories used for demonstration

Repository URL
https://issues.redhat.com/projects/WFLY
https://github.com/payara/issue-tracker
https://github.com/neo4j/neo4j/issues
https://github.com/basho/riak/issues
https://tracker.moodle.org/browse/MDL
https://sakaiproject.atlassian.net/jira/software/c/projects/SAK/issues/

References

Atlassian. (2024). Jira software documentation [https://support.atlassian.com/jira-software-cloud/].

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). The goal question metric approach. In *Encyclopedia of software engineering*. Wiley.

Bhardwaj, M., & Rana, A. (2016). Key software metrics and its impact on each other for software development projects. *SIGSOFT Softw. Eng. Notes*, *41*(1), 1–4. https://doi.org/10.1145/2853073.2853087

Buse, R. P., & Zimmermann, T. (2010). Analytics for software development. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 77–80.

Chandra, V. (2015). Comparison between various software development methodologies. *International Journal of Computer Applications*, *131*(9), 7–10.

Deepa, N., Prabadevi, B., Krithika, L., & Deepa, B. (2020). An analysis on version control systems. *2020 international conference on emerging trends in information technology and engineering (ic-ETITE)*, 1–9.

Dias, D. M., Neto, P. M., & Trindade, F. (2020). Metrics in agile software development: A systematic mapping study. *Journal of Systems and Software*.

Fenton, N., & Bieman, J. (2014). *Software metrics: A rigorous and practical approach*. CRC press.

Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations*. IT Revolution.

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley.

Gandomani, T. J., Nafchi, M. H., & Azadeh, P. (2018). Agile performance metrics: A systematic literature review. *Information and Software Technology*.

Garlan, D., & Shaw, M. (2000). *Software architecture: Perspectives on an emerging discipline*. Prentice Hall.

Henderson-Sellers, B., & Ralyté, J. (2003). Situational method engineering: State-of-the-art review. *Journal of Universal Computer Science*, *9*(6), 452–471.

Herzig, K., Just, R., Rau, F., & Zeller, A. (2013). S or not s—that is the question: How to correctly classify bug reports. *Proceedings of the 2013 international conference on Software engineering*, 201–211.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B. A., et al. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, *43*(3), 1–44.

Kupiainen, E., Mäntylä, M. V., & Itkonen, J. (2015). Using metrics in agile and lean software development – a systematic literature review of industrial studies. *Information and Software Technology*, *62*, 143–163. https://doi.org/https://doi.org/10.1016/j.infsof.2015.02.005

Maalej, W., Tiarks, R., Roehm, T., & Koschke, R. (2015). Bug report classification using natural language processing: A comparison of five classifiers. *Information and Software Technology*, *57*, 1–15.

Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *11*(3), 309–346.

Munaiah, N., Kroh, S., Cabrey, C., & Nagappan, M. (2017). Curating github for engineered software projects. *Proceedings of the 13th International Conference on Mining Software Repositories*, 422–425.

Nielsen, J. (1994). *Usability engineering*. Morgan Kaufmann.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, *24*(3), 45–77.

Saeed, S., Jhanjhi, N., Naqvi, M., & Humayun, M. (2019). Analysis of software development methodologies. *International Journal of Computing and Digital Systems*, *8*(5), 446–460.

Shneiderman, B., Plaisant, C., Cohen, M., & Jacobs, S. (2016). *Designing the user interface: Strategies for effective human-computer interaction*. Pearson.

Spinellis, D. (2016). *Code quality: The open source perspective*. Addison-Wesley.

Theocharis, G., Kuhrmann, M., Münch, J., & Diebold, P. (2015). Is water-scrum-fall reality? on the use of agile and traditional development practices. *Product-Focused Software Process Improvement: 16th International Conference, PROFES 2015, Bolzano, Italy, December 2-4, 2015, Proceedings 16*, 149–166.

Wermelinger, M., & Yu, Y. (2005). Ontology-based modeling of software development processes. *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, 321–322.

West, D., Gilpin, M., Grant, T., & Anderson, A. (2011). Water-scrum-fall is the reality of agile for most organizations today. *Forrester Research*, *26*(2011), 1–17.

Zhang, F., Mockus, A., Zou, Y., Hassan, A. E., & Adams, B. (2013). An empirical study of classification algorithms for bug resolution time prediction. *Empirical Software Engineering*, *20*, 152–190.