An Automated, Multi-Dimensional Evaluation Toolkit for GitHub Repositories

Course Project 10

Chen Ting 59305822, Li Muyu 57126873, Liang Dasi 59739292, Wang Jingchu 59841864, Wen Peixuan 59475386, Zhang Lin 59068050

***Abstract*—With the widespread use of GitHub by developers, many GitHub repository management tools have emerged, and issue-metrics is one of them. issue-metrics is a software engineering metrics tool used to search for issues, pull requests, and discussions in code repositories, measure multiple metrics, and generate a report in the form of a GitHub issue. Using issue-metrics allows teams to quickly grasp project progress and bottlenecks, saving more time compared to manually importing data. However, while issue-metrics is convenient to use, it still has some problems. For example, it only provides command-line scripts, requiring users to manually change configurations, run scripts, and manage output files; the displayed metrics are limited, and the Markdown results have poor readability. To address the shortcomings of issue-metrics, this project improves it, implementing features such as automated pipelines, multi-dimensional metrics, and a visual frontend for reports. The project also provides sample datasets and reproducible workflows for evaluation.**

I. INTRODUCTION

GitHub is the world's largest Git-based code hosting and collaboration platform. Developers can store, manage, and share code there, and utilize features such as Branches, Pull Requests, and Issues to achieve collaborative development and project version control. As GitHub projects grow in size and the number of contributors increases, accurately assessing project quality, activity, and health has become a main challenge for GitHub project management and maintenance. More comprehensive and reasonable assessment methods are therefore becoming necessary.[1]

A. The drawbacks of traditional methods

Manually collecting key data and conducting evaluation is a common and simple method, but its drawbacks are also obvious. On the one hand, manual data collection is inefficient and makes it difficult to achieve real-time data updates. On the other hand, traditional evaluation methods are often limited to indicators such as the number of stars and reposts, lacking analysis of more dimensions of the project, such as collaboration processes or the quality of community interaction. Traditional methods make it difficult to build a comprehensive and accurate project status profile, hindering effective assistance for project management and decision-making.

B. Limitations of Existing Tools

To improve evaluation efficiency, some platforms and tools have emerged that support the scraping and analysis of GitHub project data. For example, Codacy can identify errors, security vulnerabilities, and style inconsistencies in code through static analysis and provide quality scores and trend charts for projects, but its focus remains limited to the code itself. Another example is GrimoireLab, an analysis platform focusing on the health of the open-source ecosystem. It supports data collection from various data sources, such as code repositories and issue tracking systems, and provides rich visualization dashboards, offering more comprehensive functionality. However, studies on replicating GrimoireLab's data pipelines have shown that it requires users to have proficiency in Docker orchestration and ElasticSearch query syntax, which leads to a steep learning curve and high deployment thresholds.[2] It contradicts the view that open-source project evaluation tools should lower the barrier to entry to help non-data analysis experts understand various metrics.[3]

C. Derivative Development Based on Existing Tools

To provide a feasible solution to the above problems to some extent, this project extended a mature open-source project framework to develop an automated, multi-dimensional evaluation tool for GitHub projects. The decision not to develop from scratch was made to effectively reduce development complexity, allowing the project to focus more on addressing the shortcomings of existing tools and building a more complete system within a limited timeframe. Furthermore, derivative development based on an existing project also aligns with the course requirements.

D. Advantages of Our Solution

The tool we developed represents a significant advancement in automation, system integration, and evaluation dimensions. In terms of system architecture, it adopts a design that separates the Flask backend from a modern frontend based on Vue3 and Vite, optimizing the original project's single-format Markdown presentation of analysis results. This architecture has been verified to enhance platform usability, as demonstrated by the open-source framework OpenVNA, which also uses Flask backend and Vue3 frontend separation.[4] Users only need to trigger the analysis process through the frontend interface; the system automatically

updates environment configurations, executes analysis scripts, and synchronizes the generated results report to the frontend, presenting the evaluation results intuitively in a visual format. Regarding evaluation metrics, it implements a multi-dimensional indicator system including contributor activity, team response time, PR merging efficiency, community discussion health, potential developer burnout risk, and burndown charts, enabling users to have a more comprehensive understanding of the project's status. [5]

This paper is structured as follows: Part II discusses some existing project evaluation tools or platforms. Part III presents requirement engineering and compares the technology stacks. Part IV provides detailed technical information about this project. Part V describes the development process of the project tools. Part VI describes the testing and evaluation of the project tools. Part VII presents the conclusions of this work. Part VIII presents the contribution of each team member.

II. Related Work

At the outset of developing this tool, we conducted a comprehensive evaluation and research of several existing related solutions.

A. Codacy

Codacy is a well-known automated code quality assurance platform. Through integration with GitHub projects, it can fetch the latest source code for static code analysis when code changes occur, identifying programming errors, security vulnerabilities, inconsistent coding styles, and providing an intuitive quality score and historical trend graph for projects.

From our project goals, Codacy's limitation lies in its overly simplistic analytical dimensions. It only diagnoses code quality, lacking in-depth analysis of the development collaboration process, which is the focus we wanted the tool to address. We attempted to use its API to obtain basic data, but found that it did not provide API interfaces for collaboration process metrics such as contributor activity patterns and PR review efficiency.

Furthermore, as a pre-packaged SaaS service, Codacy's analytical logic and rule engine are opaque to us. We considered deep customization of its analytical rules, but found this technically difficult, and therefore quickly abandoned the plan to extend it.

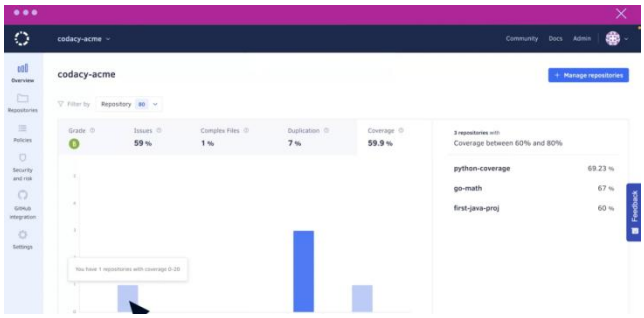

Fig. 1. Codacy Platform

B. GrimoireLab

GrimoireLab is an open-source toolset developed by the CHAOSS project for software development data analysis. It supports data collection from sources such as code repositories and issue tracking systems, and helps analyze project health, community activity, and other metrics through visualization and indicator calculations. Its metric coverage far exceeds that of most tools on the market, including code commit activity, issue lifecycles, and contributor diversity.

GrimoireLab is impressive in its comprehensive functionality, but it has a significant deployment and usage threshold. During our testing, we found that deploying the complete toolchain was very time-consuming. GrimoireLab is too complex for small project managers who want quick analysis results. We also evaluated GrimoireLab's interface. While GrimoireLab generates dashboards with a large amount of information, the overall structure is complex. Therefore, although GrimoireLab is an excellent benchmark, it is not suitable as a starting point for our pursuit of lightweight, easy-to-use, and customizable solutions.

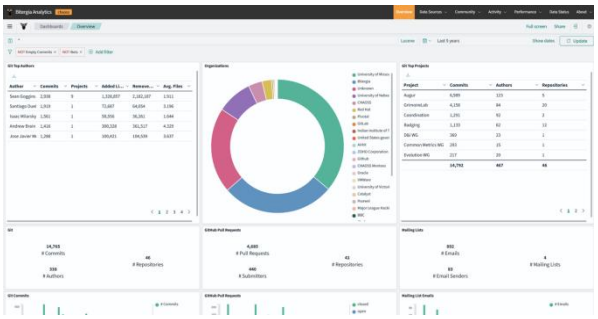

Fig. 2. GrimoireLab

C. Augur

Augur is another open-source software community analysis tool incubated by the CHAOSS community. Similar to GrimourLab, Augur focuses on assessing community health through CHAOSS metrics.

While evaluating Augur, we were initially drawn to its clear RESTful API. However, further investigation revealed the complexity of its data model. Enabling full analytics requires synchronizing Git history and GitHub collaboration data for a specified repository, a time-consuming process for large repositories that may not fully meet rapid response needs.

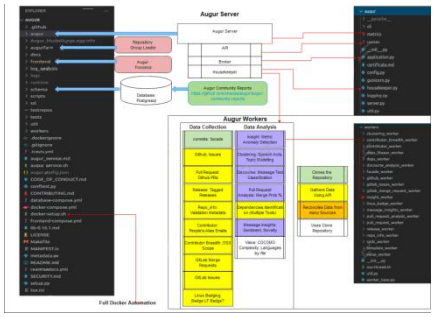

Fig. 3. Augur architecture

D. issue-metrics

issue-metrics is an open-source Python toolchain on GitHub that batch-fetches timelines for issues, pull requests, and discussions, calculates collaboration metrics such as first-response time, closure time, commenters, and roles, and generates Markdown and JSON files. Issue-metrics is a lightweight script that can run as a GitHub Action or be executed locally with a single click, making deployment extremely easy. Furthermore, its metric algorithm definition is completely transparent: every field, tag, and status transition is visible in the source code, allowing for easy modification as needed.

We ultimately chose to continue iterating on this toolchain for the following reasons:

Firstly, issue-metrics has already proven that its automated GitHub data scraping functionality can be directly reused, saving rewrite costs. Secondly, the output includes both Markdown and JSON, making it convenient for us to process the data and present it on the front end. Thirdly, the overall architecture is simple. Compared to rebuilding a large metrics platform, we only need to add Flask services and Vue dashboards to the existing scripts.

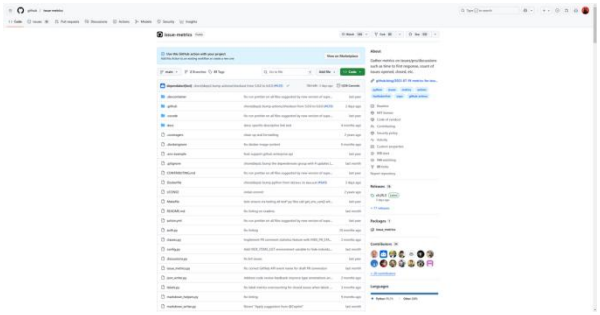

Fig. 4. issue-metrics

Based on the above analysis, we conclude that existing mature platforms either lack analytical dimensions, have excessive complexity leading to high barriers to use and deployment, or struggle to respond quickly to users. Therefore, open-source tools with a simple infrastructure, like issue-metrics, are the most suitable foundation for secondary development of this project.

III. PRELIMINARIES

A. Requirement Engineering

After discussion, we evaluated the existing functionality of issue-metrics and identified the following requirements for iteration:

R1: HTTP Token Integration
Original Project: Only supported command-line input of GH_TOKEN.
Iteration Requirement: Support Token-based Authentication; the frontend can directly configure authentication information.

R2: Workflow Automation
Original Project: Required manual script execution or GitHub Action configuration.
Iteration Requirement: Implement automatic analysis triggered upon request to ensure data real-time performance.

R3: Result Synchronization to Frontend
Original Project: Generated Markdown/JSON files were saved in a specified directory.
Iteration Requirement: Establish an automated synchronization mechanism to synchronize the latest results to the frontend directory.

R4: Upgrade the interaction method
Original project: Text-based report output.
Iteration Requirement: Develop a complete graphical interface, providing credential configuration, online browsing, chart display, and export functions.

R5: Provide a demo dataset
Original project: Supports local generation, but no pre-built data.
Iteration Requirement: Built-in offline demo dataset to support functional demonstrations in offline environments.

R6: Standardize error handling
Original project: Primarily outputs error information through logs.
Iteration Requirement: Establish a structured error handling mechanism to return standardized error responses.

R7: Expand visualization dimensions
Original project: Core metrics only include the burn-down chart and first-response time.
Iteration Requirement: Add more visualization data, including contributor activity, PR efficiency, etc.

R8: Improve architectural maintainability
Original project: Clear logic based on scripts.
Iteration Requirement: While maintaining core simplicity, centralize new features to the service layer and frontend to improve scalability.

The importance of all iterative requirements is assessed as follows:

TABLE I
Priority of all iterative requirements

Priority	Requirements
Must Have	R1, R2, R3, R4
Should Have	R5, R6, R7
Could Have	R8

B. Attribute-Driven Design

The iterative goal of this project is to upgrade the original issue-metrics system into an automated metrics system. Therefore, during the architecture design phase, it is essential to simultaneously evaluate three major components: backend performance, frontend interaction, and data storage. Based on the project team's own technical skills and commonly used technology stacks, we evaluated and scored several key quality attributes to ultimately select the most suitable technology stack for this project. These key quality attributes include: inheritance, maintainability, deployability, and performance.

I. Backend Options

We investigated several web framework combinations that work with Python scripts:

TABLE II
Backend Options Comparison

Candidate	Pros/Attributes	Fit	Remarks
Node.js + Express	Non-blocking I/O, large ecosystem	☆☆	Requires rewriting the interaction layer with issue-metrics
Python + FastAPI	Excellent async support, type-friendly	☆☆☆	Advantages in high-concurrency APIs, but requires refactoring
Python + Flask	Lightweight, easily embedded in existing scripts	★★★★	No need for extensive rewriting of original scripts, can be directly encapsulated

Decision: Flask was chosen as the backend technology. Compared to other solutions, Flask aligns with the team members' existing technology stack. Furthermore, Flask can largely reuse the Python ecosystem in issue-metrics without introducing additional threads or ORM burden, meeting the goals of loose coupling and easy maintenance driven by attributes.

II. Frontend Options

We researched common frontend technology stack combinations：

TABLE III
Frontend Options Comparison

Candidate	Pros/Attributes	Fit	Remarks
Plain HTML/CSS + jQuery	Low barrier	☆	Original
React + TypeScript	Mature ecosystem	☆☆☆	Requires additional routing/state libraries, complex bundling and configuration
Vue 3 + Vite	Composable API, lightweight scaffolding	★★★★	Can directly use dynamic components, custom themes, Ant Design Vue

Decision: React + TypeScript and Vue 3 + Vite are both common frontend technology stack combinations, and theoretically, both can be used as the frontend technology stack for this project. Comparatively, Vue 3 + Vite is more suitable for small to medium-sized web applications, enabling faster prototyping, while React + TypeScript is more suitable for large, complex enterprise-level applications. We ultimately decided to choose Vue 3 + Vite as the frontend technology stack, which is suitable for the small size of the project.

III. Data Storage Options

The issue-metrics output is already in Markdown/JSON format. We need to evaluate whether to introduce a database by comparing the following options:

TABLE IV
Data Storage Options Comparison

Candidate	Pros/Attributes	Fit	Remarks
PostgreSQL / MySQL	Structured queries	☆☆	High maintenance costs
MongoDB	Flexible schema	☆☆	Requires instance setup
JSON + Markdown	Zero setup, can be directly reused	★★★★	Consistent with the original script

Decision: Considering the nature of this project, not introducing a database has the following two advantages: First, the issue-metrics output (Markdown/JSON) can directly drive the frontend, saving the process of table creation, migration, and ORM maintenance; second, file read/write is more lightweight when not involving complex queries or multi-user concurrency. After discussion, we finally decided to continue with the original data storage solution. In the future, if it is necessary to accumulate results from multiple runs or support cross-team collaborative analysis, introducing a database would be a more appropriate approach.

IV. Attribute-Driven Score Snapshot

TABLE V
Backend Options Score

Backend	Inheritability	Maintainability	Deployability	Performance	Total
Flask	5	5	5	4	**19**
FastAPI	4	4	4	4	16
Express	3	3	3	5	14

TABLE VI
Frontend Options Score

Frontend	Data Binding	Ecosystem	Build Speed	Learning Curve	Total
Vue 3 + Vite	5	4	5	4	**18**
React + Vite	4	5	4	3	16
HTML + jQuery	2	2	3	5	12

TABLE VII
Data Storage Options Score

Storage	Compatibility	Maintenance Cost	Portability	Extensibility	Total
JSON/Markdown	5	5	5	3	**18**
MongoDB	3	3	2	5	13
PostgreSQL	3	3	2	4	12

V. Final Decisions

- Backend: Python + Flask
- Frontend: Vue 3 + Vite
- Storage: JSON/Markdown

IV. SOLUTION

A. Design Framework

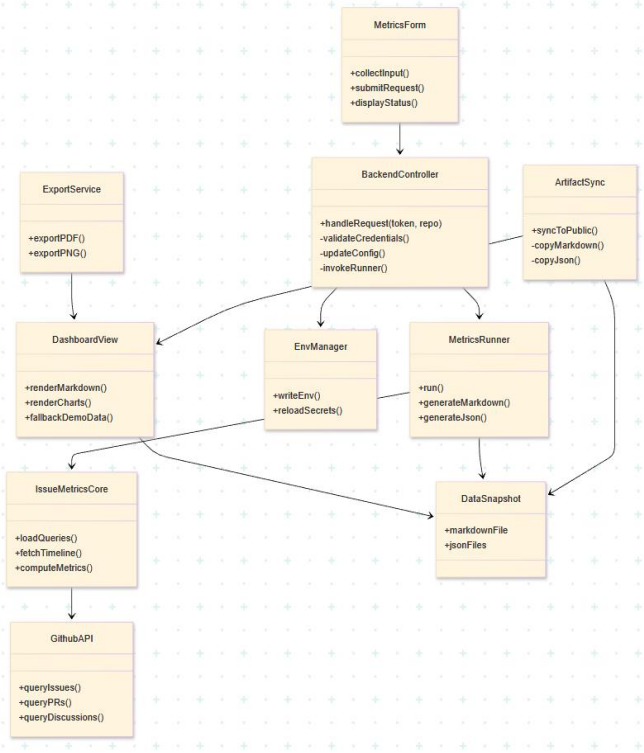

Fig. 5. Class diagram

The class diagram above is an abstract mapping of the system structure, illustrating the division of responsibilities among components during actual system operation.

The frontend includes three view classes: MetricsForm, DashboardView, and ExportService. The first two are responsible for collecting the token and GitHub repository path, rendering Markdown and multi-dimensional charts based on the latest snapshot, and automatically using demo data to display output when it is missing. The export module reuses the same view state to generate PDF/PNG files.

All requests are handled by BackendController, which calls EnvManager to complete credential verification and configuration writing. Then, MetricsRunner drives IssueMetricsCore to fetch the timeline of issues, PRs, and discussions via the GitHub API and generate Markdown and JSON files. The results are then encapsulated into a DataSnapshot, and ArtifactSync is responsible for synchronizing the snapshot to the frontend, ensuring that the interface always displays the latest data.

B. MVC architecture

In this project, we adopted the classic MVC model:

The Model is handled by the native issue-metrics script, which interacts with the GitHub REST API to generate Markdown and JSON files.

The Controller is handled by Flask, coordinating the entire process through a unified backend interface.

The View layer is built with Vue 3, using Ant Design Vue, Chart.js, and Tailwind to present text reports and multi-dimensional charts to users, and provides export options.

This architecture allows us to inherit the existing algorithms of issue-metrics, and through lightweight services and frontend components, users only need to enter their token and GitHub repository path to receive the latest data presentation, without needing to redeploy scripts or manually move files. The actual iteration focus of this project has primarily been on adding new metrics and improving the frontend presentation.

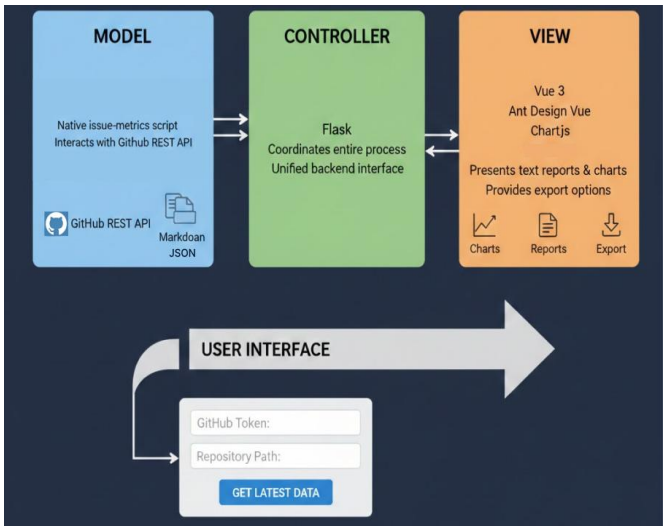

Fig. 6. MVC

C. Workflow

The overall workflow for using this project is as follows: After entering the GitHub Token and GitHub repository path on the frontend page and clicking the Analyze button, the frontend generates an HTTP request and sends it to the backend interface. The backend verifies the credentials based on the request information and modifies the environment configuration. Then, it directly calls the issue-metrics analysis script to batch-fetch data such as issues, PRs, and discussions, and outputs Markdown and multiple JSON files.

Next, the synchronization component copies the output Markdown and JSON files to the frontend static directory. The frontend immediately fetches the updated files and automatically refreshes the component status. On the frontend page, the left-hand Markdown reading area displays the text report, and the right-hand side displays visualized data, including contributor activity, response time, PR efficiency, and burnout warnings. If the user does not have a usable GitHub Token, the frontend will automatically read the project's built-in demo data.

Furthermore, if invalid parameters are used or a GitHub API exception is triggered, the interface will return a structured error response and prompt for a retry.

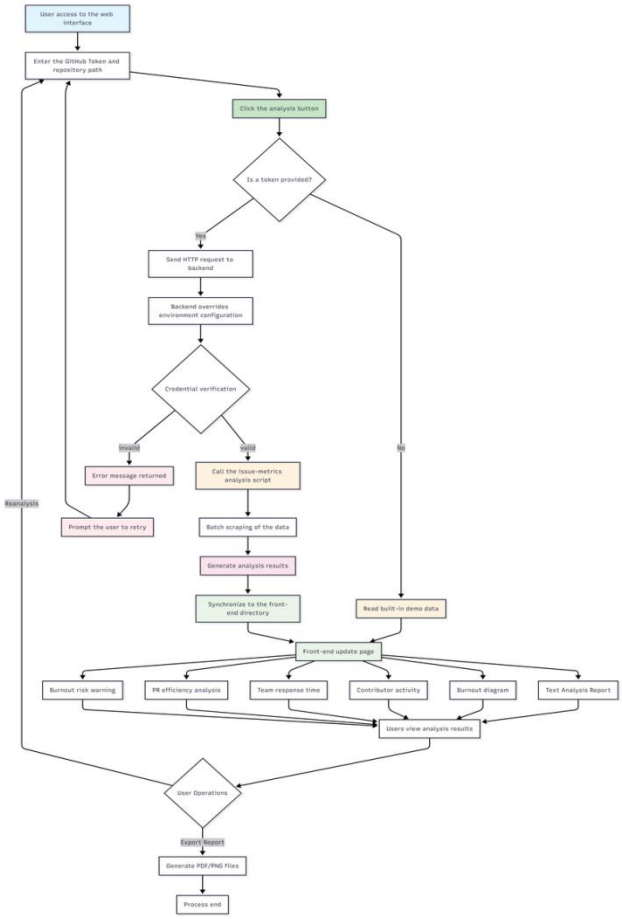

Fig. 7. Flowchart

V. Software Process

Our development cycle is built around three Sprints. At the beginning of each Sprint, the team sets the goals for that Sprint, assigns tasks using story points, and reviews the work at the end of the Sprint to guide the development of the next cycle. All our work can be presented from the dashboard below.

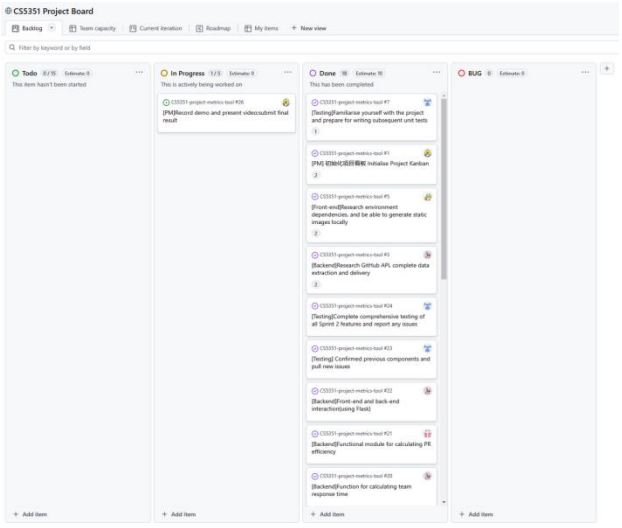

Fig. 8. Github Board

A. Sprint 1- Weeks 2 to 5

The goal of Sprint 1 was to develop user stories, sketch out architecture diagrams to meet non-functional requirements, establish the infrastructure, and deliver the first core feature: a burndown chart.

At the beginning of Sprint 1, we held our first group meeting, assigning responsibilities to team members as frontend developers, backend developers, testers, project managers, and technical writers, as required by the course. In the second group meeting, based on the information gathered after the first meeting, we focused on discussing the project's theme and development approach, and determined the project requirements and constraints for Sprint 1.

Afterwards, we clarified the project's infrastructure, which would be based on issue-metrics for secondary development, and selected a suitable technology stack. By analyzing the team members' existing technology stacks and comparing various mainstream frontend, backend, and database solutions, we ultimately chose Python + Flask as the backend framework, Vue 3 + Vite as the frontend framework, and adopted issue-metrics' mechanism of storing data via JSON/Markdown. We created a GitHub project to track progress and allocated story points through GitHub issues.

The details during the Sprint 1 development process are as follows:

1. Backend Development

Task: Implemented the burndown chart calculation engine.

Process: Developed a function that accepts JSON-formatted sprint data (start/end dates, issues list) and calculates the data points for both the actual and ideal burndown lines.

Challenge & Solution: Encountered inconsistent story point fields in GitHub issues. Addressed this by implementing a robust parsing logic using regular expressions to extract similar fields, with a default fallback of 1 story point per issue.

2. Frontend Development

Task: Constructed the main user interface and integrated the burndown chart visualization.

Process:

1) Built the core application using Vue 3 with the Composition API for reactive state management.

2) Leveraged Ant Design Vue component library to create a structured and visually consistent layout, moving away from a plain Markdown display.

3) Integrated Chart.js to render the burndown chart based on the processed data.

4) Used the Fetch API to retrieve backend-generated JSON data.

5) Enhanced user experience with a dynamic starfield background using the HTML5 Canvas API and styled the application with Tailwind CSS.

Challenges & Solutions:

1) Initially failed to fetch and display Markdown files. Solved by correctly placing static files in the public directory.

2) Ant Design Vue components were initially imported incorrectly due to a version mismatch; this was resolved by correcting the import syntax.

3) Unfamiliar with design layout using Tailwind CSS. By studying the technical documentation and repeatedly adjusting the widths of containers and components, created a centered, consistent, and aesthetically pleasing layout.

Review: We successfully delivered a fully functional web application as expected. This application extended the functionality of issue-metrics, enabling it to retrieve data from GitHub and visualize burn-down charts on the frontend page after data processing. During Sprint 1, we maintained a weekly team meeting schedule. Developers focused on discussing issues encountered and kept the PM and technical writers updated on development progress and technical details. We also communicated progress irregularly via WeChat. Furthermore, we conducted alpha testing at the end of the first iteration. Details of the testing will be provided in the evaluation section.

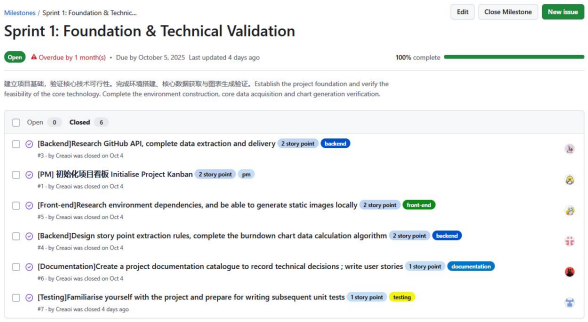

Fig. 9. Sprint 1

B. Sprint 2 - Weeks 6 to 9

Sprint 2 focused on expanding the tool's analytical capabilities by adding three new metrics and establishing a strong connection between the frontend and backend based on Flask. We also improved existing features based on the results of alpha testing.

The details during the Sprint 2 development process are as follows:

1. Backend Development

Tasks: Implemented three new modules:

1) User Activity Rate: Calculates contributor engagement based on closed issues and comments.

2) Team Response Time: Measures the efficiency of team responses to new issues.

3) PR Efficiency: Evaluates the efficiency of pull request reviews and merges.

Challenges & Solutions:

1) Due to permission issues, we could not directly obtain repository member information from GitHub. We introduced an alternative logic to address this situation: any user who has submitted comments on an issue is considered a contributing team member for response time calculation.

2) Since pull requests (PRs) vary in size, we cannot measure PR efficiency by time alone. The solution is to combine code change data from GitHub to calculate an efficiency score, and then use that score as the basis for evaluation.

2. Frontend Development

Tasks:

1) Created three new Vue components (ActivityRate.vue, ResponseTime.vue, Efficiency.vue) to display the new metrics.

2) Developed Analyze.vue as the analysis entry point, allowing users to input repository details and a GitHub Token.

3) Built a Dashboard.vue as the central hub for visualizing all analysis results.

3. Full-Stack Integration

Process: Introduced a Flask backend server to handle dynamic requests.

Challenge & Solution: A Cross-Origin Resource Sharing (CORS) error occurred. This issue was resolved by integrating the Flask-CORS extension.

Review: During the development of Sprint 2, we successfully expanded the project based on Sprint 1, transforming the application from a static data visualization tool into a dynamic full-stack tool. We also added three new metrics to provide users with more dimensions of data presentation.

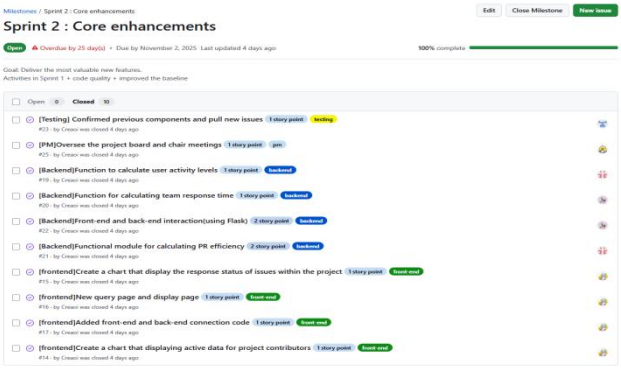

Fig. 10. Sprint 2

C. Sprint 3 - Weeks 10 to 13

The goal of Sprint 3 was to improve usability, refine functionality, and conduct system testing. The main task of the development work was to further optimize the frontend pages. In addition, based on course requirements, we also completed presentation videos, demo videos, and a final report.

Frontend Development & Enhancements

Tasks:

1) Added a sidebar with a guided tutorial on how to obtain a GitHub Personal Access Token.

2) Implemented an "Export to PDF" feature using the html2canvas library, allowing users to save and share analysis results offline.

3) Added dynamic SVG graphics, collapsed chart components into a grid layout with modal pop-ups for detailed views, and introduced a collapsible sidebar.

4) Added a search and filter function to the Issue Metrics comments section, enabling users to quickly find specific discussions.

Review: During Sprint 3 development, the frontend pages became more appealing. The developers also refined the functionality based on system testing results, improving project stability. Ultimately, the team successfully delivered a fully functional application.

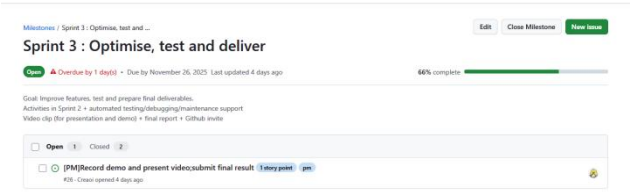

Fig. 11. Sprint 3

VI. Evaluation

Our evaluation of the solution primarily relied on testing. Test results determined whether the project's functionality was implemented as expected and whether it effectively addressed the issues outlined in the report. We conducted two main tests: an alpha test and a system test. At the end of Sprint 1, we performed an alpha test to verify the initial functionality and further refined the project in Sprint 2 based on the results. At the end of Sprint 3, we conducted a more comprehensive system test to verify that all functionality was implemented as expected. The details of both tests and the analysis of the results are described below.

A. Test Environment

We implemented a comprehensive testing suite using pytest with the following environment:

TABLE VIII
Test Environment

Testing Environment	
Language	Python 3.13.5
Platform	Windows-10-10.0.19045-SP0
Testing Framework	pytest 8.4.2

B. Alpha Testing

1. Test Objectives

To verify the accuracy of data preparation and the integrity of calculation logic for the burndown chart module, we set the test goals below:

1) Able to correctly standardize input data into a uniform format;
2) Able to parse story points from multiple sources;
3) Able to generate burndown charts as expected based on input data;
4) Able to throw reasonable errors for abnormal input.

2. Test Design

TABLE IX
Input Normalization Module

Test Dimension	Coverage Points
Input Type Support	1. Python dict 2. SimpleNamespace 3. datetime objects/ISO time strings 4. Mixed inputs of dict and objects
Date Normalization	1. Sprint start/end dates are uniformly converted to YYYY-MM-DD strings 2. created_at/closed_at in issues are normalized to YYYY-MM-DD strings
Boundary Scenario Handling	1. Cross-month dates: 04-30 and 05-02 2. Same day: Sprint start and end dates automatically set to the same day 3. Empty issues list 4. Missing created_at
Exception Handling	1. closed_at is None/empty string/non-existent field 2. Invalid date format: "2025/06/01"

TABLE X
Story Point Parsing Module

Test Dimension	Coverage Points
Direct Field	1. Supports int/float/numeric strings 2. Field is None
Labels	1. Supports multi-format labels 2. No valid labels/parsing failure
Default Behavior	Neither a story_points field nor a resolvable tag

TABLE XI
Burndown Curve Generation Module

Test Dimension	Coverage Points & Expected Results
Scenario	1. Basic scenario: Initial total points are correct, decreasing as issues are closed 2. Missing story points: Each issue is calculated as 1.0, and the curve logic is normal 3. All issues have no closed_at: remaining_points remain unchanged throughout the sprint cycle 4. All issues closed within the sprint: the remaining points were 0.0 on the last day 5. All issues are open: remaining points increase/remain stable with creation time 6. Empty issues: The actual curve remains 0.0 from the second day onward 7. One-day sprint: The curve length is 1, and the logic for same-day creation/closure is correct 8. All issues closed before the sprint: the remaining points are 0.0 throughout the cycle 9. All issues created after the sprint: the remaining points are 0.0 throughout the cycle 10. Illegal date format: Throws ValueError

3. Test Results

- Total Tests: 29
- Passed: 20 (69%) Failed: 9 (31%)
- Duration: 29ms

TABLE XII
Alpha Testing Results

Test Module	Pass Count	Pass Rate
Story Point Parsing	8/8	100%
Burndown Curve Generation	7/10	70%
Input Normalization	5/11	45.45%

4. Analysis

The overall pass rate for the Alpha test of the burndown chart was 69%, which is relatively low. The failures mainly lie in the Input Normalization section. The system performed poorly in handling invalid date formats, missing key fields, and boundary conditions, indicating a flaw in the data processing functionality. This data processing deficiency also led to a chain reaction of failures in the Burndown Curve

Generation module. On the other hand, the Story Point Parsing module passed all test cases, indicating that its core calculation logic was reliable within the scope of the test cases. Our team realized from the Alpha test that in Sprint 2, we should improve the Input Normalization module, strengthening the exception handling in date parsing and the validation of missing fields, in order to ensure stable operation when facing truly complex input data.

Fig. 12. Successful cases in the Alpha test

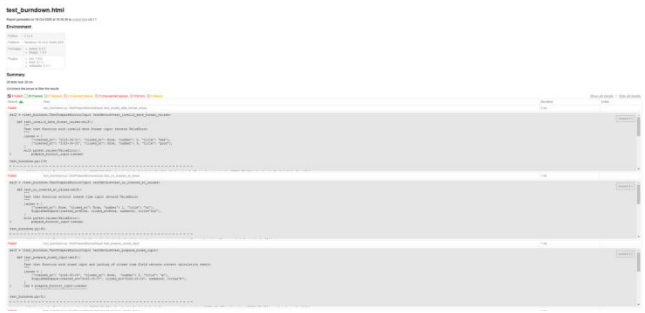

Fig. 13. Failed cases in Alpha testing

C. System Testing

1. Test Objectives

1) Able to standardize issue/PR/comment data from multiple sources into an internally unified format.

2) Consistency in various scores and time calculations across multiple scenarios.

3) The generated overview results conform to the definition of issue-metrics.

4) Promptly throw expected exceptions or safely skip over invalid input.

Fig. 14. Test approach example

2. Test Design

TABLE XIII
Team Response Time

Test Case / Scenario	Description	Result
`test_team_member_response`	Team member replies; average response time computed	✓
`test_multiple_team_replies`	Multiple team replies; earliest one used	✓
`test_comments_sorted_by_time`	Comments sorted chronologically; earliest team reply selected	✓
`test_no_comments`	Issue has no comments; response count stays 0	✓
`test_no_team_replies`	Only external comments; response count remains 0	✓
`test_mixed_issues`	Some issues responded, some not; stats correct	✓
`test_empty_issue_list`	Empty input list	✓
`test_invalid_date_format`	`created_at` invalid ISO string	✗ (ValueError)
`test_reply_before_issue_created`	Reply timestamp before issue creation	✗ (assertion)
`test_no_team_members`	No team members configured	✓
`test_fetch_all_contributors_no_data`	Repo with no contributors	✓
`test_fetch_all_contributors_pagination`	>100 contributors, pagination handled	✓
`test_fetch_all_contributors_api_error`	API failure while fetching	✓
`test_fetch_all_contributors_missing_login`	Contributor missing `login`	✓
`test_build_issues_issue_error`	Issue fetch raises error	✓
`test_build_issues_comment_error`	Comment fetch raises error	✓
`test_build_issues_comment_none`	Comment entry is `None` / missing `user`	✗ (AttributeError)
`test_build_issues_comments_not_list`	`comments()` returns non-list	✗ (TypeError)

TABLE XIV
Contributor Activity

Test Case / Scenario	Description	Result
`test_calculate_contributor_activity`	Basic scoring (closed vs comments)	✓
`test_multiple_users_sorted`	Multi-user aggregation & sort	✓
`test_duplicate_comments`	Duplicate comments counted correctly	✓
`test_open_vs_closed_states`	Distinguish open/closed issues	✓
`test_empty_issue_list`	Returns empty list	✓
`test_missing_closed_by`	Missing `closed_by` field	✓
`test_missing_comments`	Missing or empty `comments`	✓
`test_collect_issue_data_basic`	Baseline issue extraction	✓
`test_collect_issue_data_no_issues`	No issues provided	✓
`test_collect_issue_data_multiple`	Multiple issues supported	✓
`test_closed_by_no_login`	`closed_by` without login	✓
`test_comments_raise_exception`	`comments()` raises error	✓
`test_collect_issue_data_none_in_comments`	Comment list contains `None`	✗ (AttributeError)
`test_comments_not_list`	Non-list returned from `comments()`	✗ (TypeError)

TABLE XV
PR File Stats & Data Collection

Test Case / Scenario	Description	Result
`test_sum_pr_file_stats_normal`	Sum adds/deletes across files	✓
`test_sum_pr_file_stats_missing_values`	Handles `None` additions/deletions	✓
`test_sum_pr_file_stats_no_files`	No files -> (0, 0) totals	✓
`test_sum_pr_file_stats_handles_exceptions`	`files()` throws exception	✓
`test_collect_pr_data_basic`	Happy-path repository with PRs	✓
`test_collect_pr_data_invalid_repo`	Nonexistent repo -> empty	✓
`test_collect_pr_data_partial_failure`	Mixed valid/invalid repos	✓
`test_collect_pr_data_pr_details_exception`	PR detail fetch raises	✓
`test_collect_pr_data_no_prs`	Repo with zero PRs	✓
`test_collect_pr_data_zero_changes`	`changed_files=0` still recorded	✓
`test_collect_pr_data_merge_before_create`	`merged_at` earlier than `created_at`	✓

TABLE XVI
PR Review Efficiency

Test Case / Scenario	Description	Result
`test_single_pr`	Single merged PR	✓
`test_multiple_prs`	Mixed merged/unmerged/zero-comment PRs	✓
`test_zero_review_time`	Merge and review at same timestamp	✓
`test_zero_comment_pr`	Merged PR with zero comments	✓
`test_no_prs`	Empty input list	✓
`test_non_list_input`	Non-list input should raise	✓
`test_invalid_date_format`	Invalid `created_at` / `merged_at`	✗ (ValueError)
`test_missing_fields`	Missing required fields (e.g., `created_at`)	✗ (KeyError)

3. Test Results
- Total Tests: 49
- Passed: 42 (86%) Failed: 7 (14%)
- Duration: 66ms

TABLE XVII
System Testing Results

Test Module	Pass Count	Pass Rate
Team Response Time	16/19	84.21%
Contributor Activity	10/12	83.33%
PR File Stats & Data Collection	10/10	100%
PR Review Efficiency	6/8	75%

4. Analysis

The overall pass rate for this system test was 86%, a significant improvement over the Alpha test (69%). The test results indicate that the system's core functions and data acquisition logic performed well. Similar to the Alpha test, the main problem lies in data processing. Specifically, the PR File Stats & Data Collection module achieved a 100% pass rate, demonstrating its excellent performance in summarizing PR file data and collecting repository data. The errors in the PR Review Efficiency (75%), Contributor Activity (83.33%), and Team Response Time (84.21%) modules were similar, all failing to correctly handle invalid ISO date strings, missing key fields, and unexpected data types returned by the API. The team summarized the reasons for these test results as follows: the team developers focused more on stable implementation during development. The project's data source relies on an external GitHub API, and the returned data format and integrity cannot be guaranteed, which led the developers to fail to establish a sound data validation and exception handling mechanism when writing the program. This introduced some technical debt into our system.

In summary, the system's functionality performed well in the test, but its stability needs further improvement in the future.

VII. Conclusion

This project is a secondary development based on the open-source issue-metrics library. A one-click metric insight pipeline was successfully implemented during the development cycle.

We expanded the original burndown chart, adding dimensions such as contributor activity, team response time, PR efficiency, and discussion health. We also used components like Ant Design Vue, Chart.js, and html2canvas to provide interactive Markdown reading, dark mode, search, and PDF export functionality. Project validation shows that, without introducing a database, relying solely on the GitHub API and local JSON storage, it can effectively reproduce metrics such as burndown charts, response latency, and efficiency distribution.

Next, we aim to enhance the robustness of the existing backend algorithm, optimizing data processing anomalies exposed during system testing to ensure stable parsing of data from any repository. Additionally, we hope to improve the CI/CD and frontend experience, integrating one-click analysis into the automated pipeline and integrating the tool into the GitHub Actions pipeline to automatically analyze and present multiple metrics upon commit. Through future optimizations, we hope this tool will gradually evolve into a long-term, operational repository management assistant.

VIII. Relative Contribution

Name	Role	Sprint 1	Sprint 2	Sprint 3	Total	Contribution (%)
Zhang Lin	Testing Engineer	15	13	7	30	16.67%
Chen Ting	Frontend Developer	12	10	8	30	16.67%
Liang Dasi	Backend Developer	15	10	5	30	16.67%
Wang Jingchu	Backend Developer	12	12	6	30	16.67%
Wen Peixuan	Project Manager	5	10	15	30	16.67%
Li Muyu	Technical Writer	6	6	18	30	16.67%

Fig. 15. Contribution bias diagram

Appendix

Fig. 16. Burndown chart [for this project]

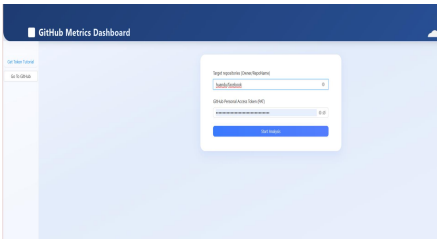

Fig. 17. Repository path and Token input

Fig. 18. Dashboard

Fig. 19. PR efficiency dashboard

Fig. 20. Contribution activity overview

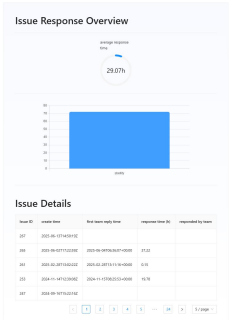

Fig. 21. Issue response overview

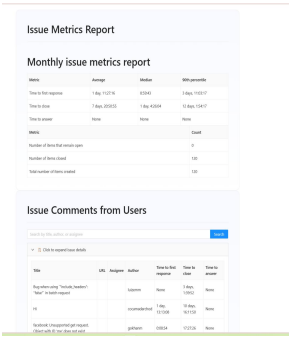

Fig. 22. Issue metrics report

ACKNOWLEDGMENT

The authors acknowledge the use of GPT / Gen AI tools during the development of this project to assist with coding scrapping and summarizations. All AI-generated output was verified, edited, and refined by the human authors, who take full responsibility for the final content and integrity of this work.

REFERENCES

[1] J. Linåker, E. Papatheocharous, and T. Olsson, "How to characterize the health of an Open Source Software project? A snowball literature review of an emerging practice," in Proceedings of the 18th International Symposium on Open Collaboration, J. Arroyo, F. Martín-Rico, I. Steinmacher, A. Decan, G. Catolino, A. K. Verma, A. Balderas, G. Robles, and A. Charleux, Eds., New York, NY, USA: ACM, 2022, pp. 1–12. doi: 10.1145/3555051.3555067

[2] K. Eng and H. Sahar, "Replicating Data Pipelines with GrimoireLab," 2022, doi: 10.48550/arxiv.2205.02727

[3] D. Foster and D. Riehle, "From Data to Action: Building Healthy and Sustainable Open Source Projects," Computer (Long Beach, Calif.), vol. 58, no. 6, pp. 74–78, 2025, doi: 10.1109/MC.2025.3530556

[4] Z. Yuan, B. Zhang, H. Xu, Z. Liang, and K. Gao, "OpenVNA: A Framework for Analyzing the Behavior of Multimodal Language Understanding System under Noisy Scenarios," 2024, doi: 10.48550/arxiv.2407.02773

[5] T. Xia, W. Fu, R. Shu, R. Agrawal, and T. Menzies, "Predicting health indicators for open source projects (using hyperparameter optimization)," Empirical software engineering : an international journal, vol. 27, no. 6, Art. no. 122, 2022, doi: 10.1007/s10664-022-10171-0

Technical Documentation List

[6] Vue.js. Introduction. Vue.js Documentation. Available: https://vuejs.org/guide/introduction.html

[7] Ant Design Vue. Introduction. Ant Design Vue Documentation. Available: https://www.antdv.com/docs/vue/introduce-cn

[8] Marked. Documentation. Marked.js Official Site. Available: https://marked.js.org/

[9] MDN Web Docs. Fetch API. Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[10] Chart.js. Documentation. Chart.js Official Documentation. Available: https://www.chartjs.org/docs/latest/

[11] Tailwind CSS. Documentation. Tailwind CSS Official Site. Available: https://tailwindcss.com/docs

[12] MDN Web Docs. Canvas API. Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

[13] Pallets Projects. Flask Documentation. Available: https://flask.palletsprojects.com/

[14] Flask-CORS. Documentation. Available: https://flask-cors.readthedocs.io/

[15] python-dotenv. PyPI Page. Available: https://pypi.org/project/python-dotenv/

[16] html2canvas. Official Documentation. Available: https://html2canvas.hertzen.com/

[17] Teamwork. Project Management Software. Teamwork.com. Available: https://www.teamwork.com/