CS5351-Group 10-Final Report

Group Members：

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 upon the original project, implementing features such as automated pipelines, multi-dimensional metrics, and visual reports. The project also provides sample datasets and reproducible workflows for evaluation.

1. 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 (PRs, code merge requests), and Issues to achieve collaborative development and project version control. With the growth in the size of GitHub projects and the increase in the number of contributors, accurately assessing the quality, activity, and health of projects has become a major challenge in GitHub project management and maintenance.

The drawbacks of traditional methods

Manually collecting key data and conducting manual 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 and the quality of community interaction. These problems make it difficult to build a comprehensive and accurate project status profile using this method, hindering effective assistance for project management and decision-making. Promoting the automation of data collection and the multidimensionality of evaluation systems has become an inevitable choice for improving project management efficiency.

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 geared towards 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, its complex architecture and high deployment and configuration barriers make it difficult for project teams to directly use for simple daily iteration management.

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.

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. 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.

2. Related Work

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

A. Codacy

@codacy/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 repositories with an intuitive quality score and historical trend graph.

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.

B. GrimoireLab

GrimoireLab is one of the most mature and powerful open-source project health analysis platforms under the CHAOSS community. It can automatically collect data from multiple data sources such as Git, GitHub, and GitLab, and provides a feature-rich, Kibana-based visualization dashboard. 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 barrier. 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.

C. Augur

@chaoss/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 attracted by its clear RESTful API. However, further investigation revealed its data model's complexity. Enabling Augur's full analytics requires synchronizing historical Git data and GitHub collaboration data from a specified repository for

metric calculation. This process is time-consuming for large repositories and may not fully meet the need for rapid response.

D. issue-metrics

`issue-metrics` is an open-source Python toolchain from the GitHub team. Its core capability is batch fetching timelines of issues, pull requests, and discussions, calculating collaboration metrics such as first-response time, closure time, commenters, and roles, and simultaneously generating Markdown reports and JSON structured files. It defaults to a lightweight script format of `.env` + `data/`, and can run as a GitHub Action or be executed locally with a single click, making it virtually untapped for course projects or small teams. More importantly, its metric algorithm definition is completely transparent: every field, tag, and status transition is provided in the source code, allowing for rapid customization as needed, without the limitations of a SaaS black box.

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

First, `issue-metrics` has already proven its automated GitHub data fetching functionality can be directly reused, saving rewrite costs;

Second, the output includes both Markdown and JSON, making it convenient for us to process the data and present it on the front end;

Third, the overall architecture is simple. Compared to rebuilding a massive metrics platform, we can achieve a seamless experience of "front-end token submission, one-click analysis triggering, and synchronized chart rendering" simply by wrapping existing scripts with Flask services and Vue dashboards.

Therefore, `issue-metrics` became the most reliable foundation for secondary development in this project.

E. Solution Based on the above analysis, we conclude that existing mature platforms either lack analytical dimensions, have excessive complexity leading to usage and deployment barriers, or struggle to provide rapid user responses. Therefore, open-source tools with simple infrastructures like `issue-metrics` are perfectly suited for our secondary development goals.

3. Preliminaries

Requirement Elicitation

- R1: Backend must accept authenticated GitHub tokens and repository scopes through an HTTP endpoint.
- R2: The system shall execute issue-metrics analytics and regenerate Markdown/JSON artifacts on every request.
- R3: Outputs must be synchronized to the Vue frontend so dashboards and Markdown share the same snapshot.
- R4: Users need a web UI to submit credentials, inspect charts, and download Markdown without touching the CLI.
- R5: The tool shall provide demo datasets for offline grading and classroom rehearsals.
- R6: Failures (invalid token, API quota, file I/O) must be logged and returned as structured HTTP errors.
- R7: Frontend charts must cover burnout, contributor activity, response time, and PR efficiency.
- R8: Configuration (.env paths, data directories) should remain compatible with the upstream repository to ease upgrades.
- R9: The data format exposed to the frontend must remain stable for future visualization additions.
- R10: Non-functional—solution should be easy to maintain and extensible for new metrics or views.

1. Requirement Specification

- R1 is foundational because all downstream analytics depend on authenticated access.
- R2 and R3 build upon R1 by transforming raw GitHub data into consumable artifacts and ensuring cross-layer consistency.
- R4–R7 depend on R1–R3: they describe how users interact with the regenerated artifacts (UI submission, visualization coverage, offline demos).
- R8 and R9 govern compatibility and data contracts, constraining how future changes integrate with the original codebase.
- R10 captures maintainability/extensibility expectations affecting every layer. No conflicts were found; dependencies guided our implementation order (R1 → R2/R3 → R4–R7 → R8/R9 → R10).

Attribute-Driven Design

We extend the CLI-first `issue-metrics` scripts into a browser-facing analytics service, so every architectural decision is guided by four attributes: inheritance (reuse as much of the upstream Python code as possible), maintainability (small student teams can iterate), deployability (classroom demos run on laptops), and predictable performance (still constrained by GitHub API quotas). These attributes drive choices for the backend orchestration layer, frontend visualization stack, and data persistence strategy.

Backend Options

Candidate	Pros/Attributes	Fit	Remarks
Node.js + Express	Non-blocking I/O, large ecosystem	☆☆	Requires rewriting the Python orchestration or building IPC glue
Python + Django	Batteries included, ORM, auth	☆☆	Overkill for a thin wrapper, introduces unnecessary migrations
Python + FastAPI	Async, type hints, modern tooling	☆☆☆	Would force us to refactor synchronous `run.main()`
Python + Flask	Minimal, embeds existing scripts easily	★★★★	Can update `.env`, call `run.main()`, and copy files in the same process

Decision: Adopt Flask as the orchestration shell. It keeps `issue-metrics` as the analytical core, satisfies the inheritability attribute, and minimizes new dependencies.

Frontend Options

Candidate	Pros/Attributes	Fit	Remarks
Plain HTML/CSS + jQuery	Low barrier	☆	Hard to maintain stateful dashboards
React + TypeScript	Mature ecosystem, strong typing	☆☆☆	Heavier tooling; needs routing/state packages
Svelte + Vite	Fast builds, tiny bundles	☆☆	Team has limited experience; smaller community
Vue 3 + Vite	Composition API, single-file components, fast dev server	★★★★	Reads static JSON from `public/`, integrates with Ant Design Vue/Chart.js

Decision: Use Vue 3 + Vite. This satisfies usability and maintainability attributes while enabling static hosting and component reuse (e.g., `BurnoutChart`, `Activityrate`).

Data Storage Choices

Candidate	Pros/Attributes	Fit	Remarks
PostgreSQL / MySQL	Rich queries, transactions	☆☆	Requires DB provisioning; overkill for report snapshots
MongoDB	Flexible schema, horizontal scaling	☆☆	Adds operational overhead, not needed for read-mostly metrics
Redis	In-memory speed	☆	Does not provide durable historical snapshots
Flat JSON + Markdown	Zero setup, matches upstream outputs	★★★★	Works offline, versionable via Git,

			readable by SPA directly

Decision: Continue writing `issue_metrics.md` + JSON to `data/`, then mirror to `frontend/public/` with `copy_files.py`. This honors the compatibility attribute and keeps demos database-free.

Attribute-Driven Score Snapshot (Illustrative)

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

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

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

Final Decisions Aligned with Attributes

- **Backend**: Flask orchestrator encapsulating `issue-metrics`, preserving the proven Python logic (inheritability).
- **Frontend**: Vue 3 + Vite SPA for token input, Markdown preview, and chart rendering (usability & maintainability).
- **Storage**: Plain files synchronized to `frontend/public/`, ensuring compatibility with the upstream repo and effortless offline demos.

This attribute-driven approach lets us ship a service-oriented experience without re-implementing the analytics core, balancing extensibility, low operational cost, and classroom readiness.

4. Solution

A. Solution Walkthrough

Users submit token + repo via the Vue dashboard. Flask `/update_env` validates input, rewrites `.env`, runs `run.main()` inside `data/`, copies Markdown/JSON outputs to `frontend/public`, and returns `issue_metrics.md`. The SPA reloads the synchronized JSON to update burnout, activity, response-time, and PR-efficiency charts; failures surface as structured HTTP errors.

B. Design Principles & Framework

We keep the original `issue-metrics` scripts as the **Model**, wrap them in a Flask **Controller** that handles HTTP orchestration and filesystem tasks, and present results through a Vue 3 **View** layer with reusable chart components. This separation mirrors an MVC mindset and keeps analytics reproducible while the UI remains reactive.

C. Modern Code Review

Development happens on feature branches, merged through GitHub PRs. Reviewers inspect API contracts (env handling, JSON schemas) and front-end bindings before approving. Example PRs (e.g., dashboard syncing) show comments, requested changes, and final merges. *(Fig. 3 "Git Log & PR" placeholder.)*

D. Testing

Backend Pytest suites validate burndown math, contributor aggregation, and Markdown formatting using bundled demo data. Frontend verification ensures each chart consumes the correct JSON fields. Key checks: `.env` rewrites, directory switching, file copy success, and component rendering with sample datasets.

E. Technical Debt

Current compromises include GitHub rate-limit sensitivity, the reliance on static `public/` syncing for deployments, and pending schema work for future metrics (discussion sentiment, mentor SLA). All items are logged for future iterations to maintain extensibility and operational resilience.

---

5. Software Process

For this project, we adopted Scrum as our primary software development framework. Scrum's agile and iterative nature was instrumental in managing the development of our GitHub project metrics analysis tool. It allowed us to break down the complex system into manageable increments, respond effectively to emerging challenges, and deliver functional value consistently at the end of each Sprint. Our development cycle was structured around three Sprints, each with a defined set of goals, followed by a review and retrospective to inform the next cycle.

Sprint 1: Foundation and Core Feature Delivery

Duration: Weeks 2 to 5

The objective of Sprint 1 was to establish the foundational architecture and deliver the first core feature: the Burndown Chart.

- 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.
  - Technology: Python for data processing logic.
- Frontend Development:
  - Task: Constructed the main user interface and integrated the burndown chart visualization.
  - Process:
    - Built the core application using Vue 3 with the Composition API for reactive state management.
    - Leveraged Ant Design Vue component library to create a structured and visually consistent layout, moving away from a plain Markdown display.
    - Integrated Chart.js to render the burndown chart based on the processed data.
    - Used the Fetch API to retrieve backend-generated JSON data.
    - Enhanced user experience with a dynamic starfield background using the HTML5 Canvas API and styled the application with Tailwind CSS.
  - Challenges & Solutions:
    - Initially failed to fetch and display Markdown files. Solved by correctly placing static files in the `public` directory.
    - Ant Design Vue components were initially imported incorrectly due to version mismatch; resolved by correcting the import syntax.
    - Adjusted container and component widths using Tailwind CSS to create a centered, uniform, and aesthetically pleasing layout.

Sprint 1 Outcome: Successfully delivered a functional web application that could fetch data, process it, and visually present a burndown chart, setting a solid foundation for future enhancements.

Sprint 2: Metrics Expansion and Full-Stack Integration

Duration: Weeks 6 to 9

Sprint 2 focused on expanding the analytical capabilities of the tool by adding three new metrics and creating a robust connection between the frontend and backend.

- Backend Development:
  - Tasks: Implemented three new calculation modules:
    i. User Activity Rate: Calculates contributor engagement based on closed issues and comments.
    ii. Team Response Time: Measures the efficiency of team responses to new issues.
    iii. PR Efficiency: Evaluates the efficiency of pull request reviews and merges.
  - Challenges & Solutions:
    - Could not directly fetch repository members from GitHub due to permission issues. Implemented a fallback logic: any user who has submitted a comment on an issue is considered a contributing team member for response time calculation.
    - PR efficiency could not be measured by time alone due to varying PR sizes. Solved by incorporating code change data (additions/deletions) from GitHub to calculate an efficiency score (changes per hour).
- Frontend Development:
  - Tasks:
    - Created three new Vue components (`ActivityRate.vue` , `ResponseTime.vue` , `Efficiency.vue`) to display the new metrics.
    - Developed `Analyze.vue` as the analysis entry point, allowing users to input repository details and a GitHub Token.
    - Built `Dashboard.vue` as the central hub for visualizing all analysis results.
  - Technology: Continued use of Vue 3, Ant Design Vue, and Tailwind CSS.
- Full-Stack Integration:
  - Process: Introduced a Flask backend server to handle dynamic requests.
  - API Workflow:
    i. Frontend sends a analysis request to the Flask endpoint (`/update_env`).
    ii. Backend uses `python-dotenv` to update environment variables (Token, Repo Name).
    iii. Backend executes the main analysis script (`run.main()`) which fetches data from GitHub and generates the required JSON output.
    iv. A custom script (`copy_files.copy_files_to_frontend()`) copies the results to the frontend's `public` directory.

v. Backend sends a response back to the frontend, which then navigates to the dashboard to display the results.

- Challenge & Solution: Faced Cross-Origin Resource Sharing (CORS) errors. Resolved by integrating the Flask-CORS extension to enable secure communication between the frontend and backend.

Sprint 2 Outcome: Transformed the application from a static data visualizer to a dynamic, full-stack tool capable of analyzing any GitHub repository and presenting a comprehensive set of project metrics.

Sprint 3: User Experience Refinement and Polishing

Duration: Weeks 10 to 13

The goal of Sprint 3 was to enhance usability, address user feedback, and add final polishing features.

- Frontend Development & Enhancements:
  - Tasks:
    - Improved Usability: Added a sidebar with a guided tutorial on how to obtain a GitHub Personal Access Token, lowering the barrier for entry.
    - Data Persistence: Implemented a "Export to PDF" feature using the html2canvas library, allowing users to save and share analysis results offline.
    - Enhanced Navigation & Visualization: Added dynamic SVG graphics, collapsed chart components into a grid layout with modal pop-ups for detailed views, and introduced a collapsible sidebar.
    - Advanced Data Interaction: Added a search and filter function to the Issue Metrics comments section, enabling users to quickly find specific discussions.
    - Visual Appeal: Prototyped a dark/light mode toggle (partially implemented across components).
- Backend Support:
  - Process: The backend was updated to support the new frontend features, ensuring the generated JSON data structures were compatible with the search, sorting, and enhanced burndown chart (which now included a direct actual vs. ideal line comparison).

Sprint 3 Outcome: Delivered a polished, user-friendly, and feature-complete application that not only provides powerful analytics but also ensures a smooth and accessible user experience.

Project Management and Tracking

Throughout the project, we maintained a product backlog to prioritize tasks. Weekly scrum meetings were held to synchronize the team, discuss progress, and tackle impediments. We utilized Burndown Charts to visually track the completion of tasks within each Sprint, providing transparency and enabling the team to monitor its velocity and adjust plans proactively. This disciplined yet flexible approach ensured the timely and successful delivery of the project across all three Sprints.

6. Evaluation

This test focused on verifying the functionality of core modules related to the burndown chart. The goal was to ensure the correctness of data processing logic for three key modules: prepare_burnout_input (input normalization), parse_story_points (story point parsing), and generate_burnout (burndown curve generation). It also documented known issues during system operation. The test covered normal functions, boundary scenarios, and exception handling, with specific results as follows:

Test Objectives

To verify the accuracy of data preparation and integrity of calculation logic for the burndown chart module, ensuring:

1. **Input data (issues) can be correctly normalized into a unified format;**
2. **Story points can be accurately parsed from multiple sources (fields/labels);**
3. **Ideal/actual burndown curves can be generated as expected based on input data;**
4. **The module can throw reasonable errors for abnormal inputs (e.g., invalid dates, empty data).**

Test Framework Configuration

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

Testing Environment	
Language	**Python 3.13.5**
Platform	**Windows-10-10.0.19045-SP0**

Testing Framework	**pytest 8.4.2**
Reporting	**pytest-html 4.1.1**
Coverage	**pytest-cov 7.0.0**

Detailed Test Results for Each Module

1. prepare_burnout_input (Input Normalization Module)

This module is responsible for converting unstandardized issue data into a unified format and calculating the start and end dates of the sprint. The test covered three types of scenarios: input compatibility, date normalization, and boundary/exception handling:

Test Dimension	Coverage Points & Results
Input Type Support	**Successfully compatible with 4 types of inputs:** **- Python dict (containing created_at/closed_at fields)** **- SimpleNamespace (or similar objects)** **- datetime objects/ISO time strings (e.g., "2025-10-01T09:00:00Z")** **- Mixed inputs of dict and objects**
Date Normalization	**The output format fully meets expectations:** **- Sprint start/end dates (sprint_start/sprint_end) are uniformly converted to YYYY-MM-DD strings** **- created_at/closed_at in issues are normalized to YYYY-MM-DD strings (None if the issue is not closed)**
Boundary Scenario Handling	**1. Cross-month dates (e.g., 04-30 and 05-02): Sprint cycle calculated correctly** **2. Single issue: Sprint start and end dates automatically set to the same day (start == end)** **3. Empty issues list: Throws ValueError (consistent with exception definition)** **4. All issues missing created_at: Throws ValueError (prevents invalid data flow)**
	1. closed_at is None/empty string/non-existent field: Uniformly handled as None

Exception Handling	**2. Invalid date format (e.g., "2025/06/01"): Throws ValueError (blocks illegal data)**

2. parse_story_points (Story Point Parsing Module)

This module extracts story points (for burndown curve calculation) from issues and supports parsing from multiple sources. The test covered direct field parsing, label parsing, and default behavior:

Parsing Source	Coverage Points & Results
Direct Field (story_points)	**1. Supports int/float/numeric strings (e.g., "3"): All correctly parsed to float (e.g., 3 → 3.0, "1.5" → 1.5)** **2. Field is None: Automatically falls back to the default value of 1.0**
Labels	**1. Supports multi-format labels (case-insensitive):** **- Examples: "SP:3" → 3.0, "story_points-2" → 2.0, "sp 1.5" → 1.5** **2. No valid labels/parsing failure: Returns the default value of 1.0**
Default Behavior	**If there is neither a story_points field nor parsable labels: Uniformly returns the default value of 1.0 (ensures calculation is not interrupted)**

3. generate_burnout (Burndown Curve Generation Module)

This module generates ideal and actual burndown curve data based on normalized input and parsed story points. The test covered output structure and 10 typical scenarios:

Test Dimension	Details
Output Structure	**Returns a dictionary containing two arrays: "actual" and "ideal". Each element in the array is {"remaining_points": float value} (meets subsequent visualization requirements)**
Typical	**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**

Scenario Verification	3. All issues have no closed_at: remaining_points remains unchanged throughout the sprint cycle 4. All issues closed within the sprint: remaining_points is 0.0 on the last day 5. All issues are open (not closed): remaining_points increases/remains stable with creation time 6. Empty issues: The actual curve remains 0.0 from the second day onward 7. One-day sprint (start == end): The curve length is 1, and the logic for same-day creation/closure is correct 8. All issues closed before the sprint: remaining_points is 0.0 throughout the cycle 9. All issues created after the sprint: remaining_points is 0.0 throughout the cycle 10. Illegal date format (e.g., "2025/10/01"): Throws ValueError (blocks invalid parameters)

Test Results Analysis

Overall Test Performance

Execution Summary:

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

Successful Test Categories

Test Category	Pass Count	Key Validations
Story Point Parsing	8/8	All met
Burn-down Calculations	7/10	Core logic met
Input Handling	5/11	Basic function met

7. Conclusion

The project upgrades the open-source `issue-metrics` metrics script into a "one-click" metrics insight pipeline: the frontend (Vue3/Vite) receives tokens and repository names, triggers the Flask API, and the backend calls `run.main()` to collect GitHub issue/PR/discussion data and generate Markdown/JSON reports, which are then synchronized to `frontend/public` via `copy_files.py` for visualization.

We've expanded the original burndown chart, adding dimensions such as contributor activity, team response time, PR efficiency, and discussion health. We also use components like Ant Design Vue, Chart.js, and html2canvas to provide interactive Markdown reading, dark mode, search, and PDF export, allowing teams to directly reuse these insights in sprint reviews or course presentations.

Project validation shows that even without a traditional database, relying solely on the GitHub API and local JSON storage, it can reliably reproduce core metrics such as burndown, response latency, and efficiency distribution, providing data support for guiding course team management and open-source community governance. Next, we will follow the CI/CD practices in the literature and integrate `run.main()`, static front-end building, and Playwright end-to-end verification into a pipeline in GitHub Actions to achieve a continuous feedback loop of "analysis upon submission and visibility upon release", so that the extended version of `issue-metrics` can be truly integrated into daily repository operations.

8. Relative Contribution

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

9. References

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

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

[3] Marked. *Documentation*. Marked.js Official Site. [Online]. Available: https://marked.js.org/

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

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

[6] Tailwind CSS. *Documentation*. Tailwind CSS Official Site. [Online]. Available: https://tailwindcss.com/docs

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

[8] Pallets Projects. *Flask Documentation*. [Online]. Available: https://flask.palletsprojects.com/

[9] Flask-CORS. *Documentation*. [Online]. Available: https://flask-cors.readthedocs.io/

[10] python-dotenv. *PyPI Page*. [Online]. Available: https://pypi.org/project/python-dotenv/

[11] html2canvas. *Official Documentation*. [Online]. Available: https://html2canvas.hertzen.com/

[12] 维基百科. *燃尽图*. [在线]. 可用: https://zh.wikipedia.org/wiki/燃尽图

[13] Teamwork. *Project Management Software*. Teamwork.com. [Online]. Available: https://www.teamwork.com/

[14] M. Kopel et al., “Implementing AI for Non-Player Characters in 3D Video Games,” in Intelligent Information and Database Systems, Switzerland: Springer International Publishing AG, 2018, pp. 610–619. doi: 10.1007/978-3-319-75417-8_57

[15] G. N. Yannakakis and J. Togelius, Artificial Intelligence and Games, 1st ed. Cham: Springer Nature, 2018. doi: 10.1007/978-3-319-63519-4

[16] T. B. Brown *et al.*, “Language Models are Few-Shot Learners,” 2020, doi: 10.48550/arxiv.2005.14165