Automated GitHub Repository Quality Evaluation: A Metrics-Based Approach

Dmytro Polishchuk

Github overview

Ex. https://github.com/ansible/ansible

1. Commits

Commits represent individual changes to a repository's codebase. Each commit records a snapshot of the project's state at a specific point in time.

2. Issues

Issues represent tasks, bugs, or feature requests within a repository. They are often used to track the progress of software development and maintain a record of discussions around various problems or enhancements.
https://github.com/ansible/ansible/issues/84909

3. Pull Requests (PRs)

Pull requests are used to propose changes to a repository. They are essential for collaborative workflows, as they allow other contributors to review code changes before merging them into the main codebase.
https://github.com/ansible-community/antsibull-nox/pull/26

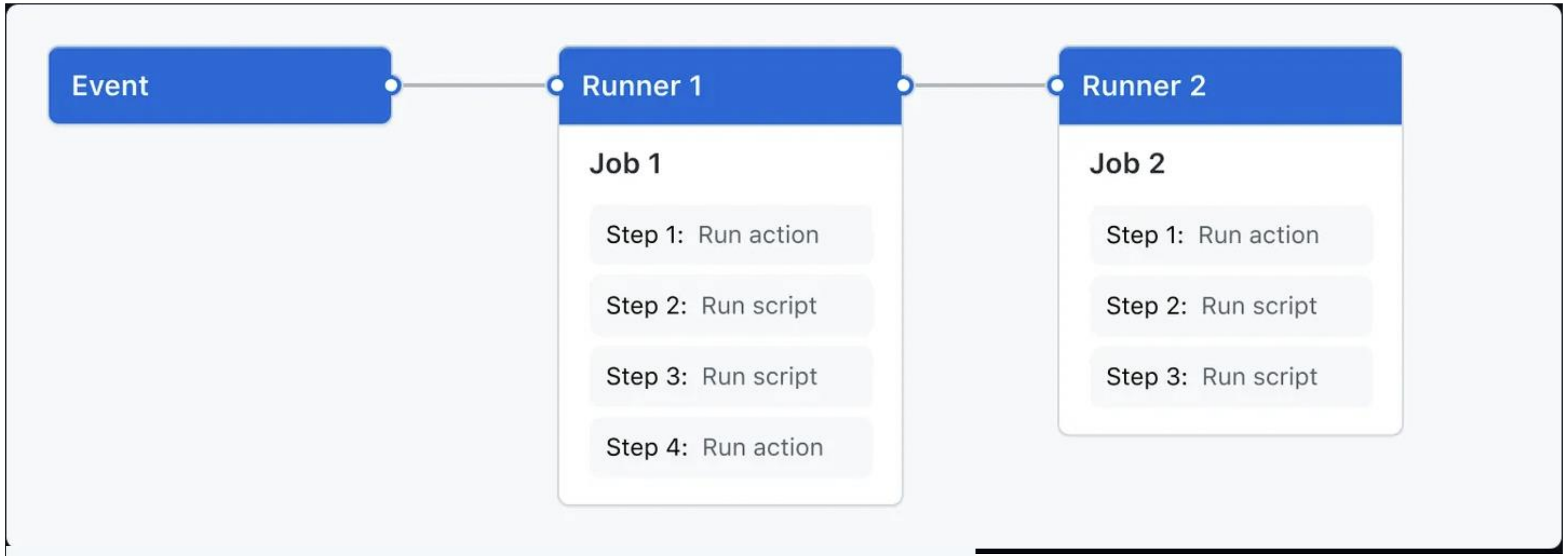
4. Actions

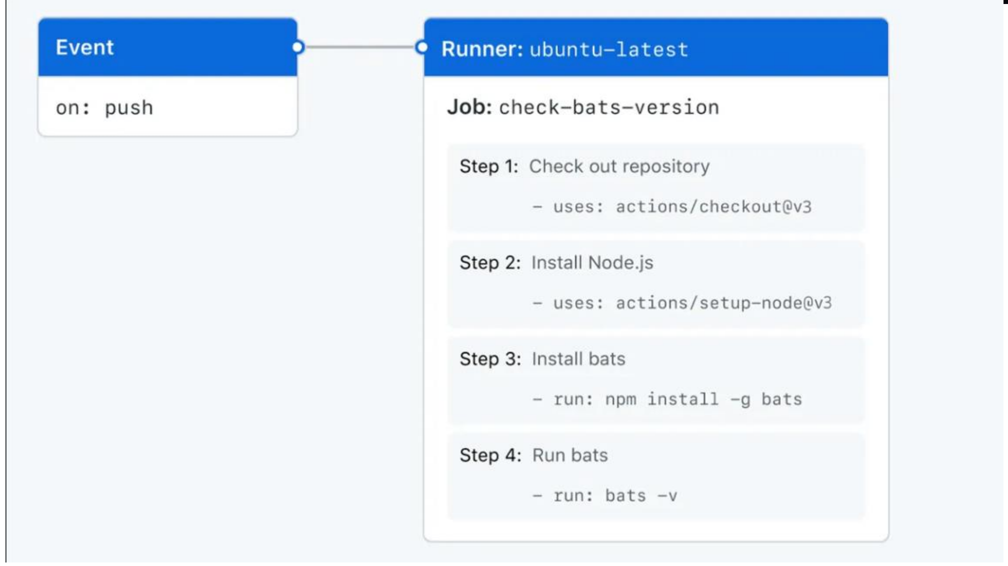
GitHub Actions is a CI/CD tool that allows developers to automate workflows, such as running tests, building applications, and deploying software when certain events occur in a repository.
https://github.com/ansible-community/antsibull-nox/pull/26/checks

5. Timeline

The GitHub timeline is a chronological sequence of events that provides an overview of all activities related to an issue or pull request. It captures all updates, comments, status changes, and relevant interactions.
https://github.com/ansible-community/antsibull-nox/pull/26/commits/8b7f4e2dd368c7aa471bffd2ca6a3f7c5f918408

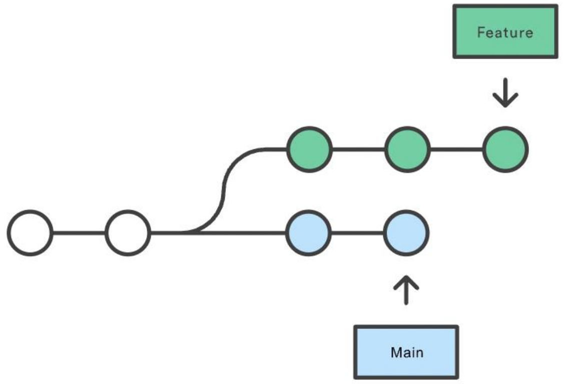
Github workflows:



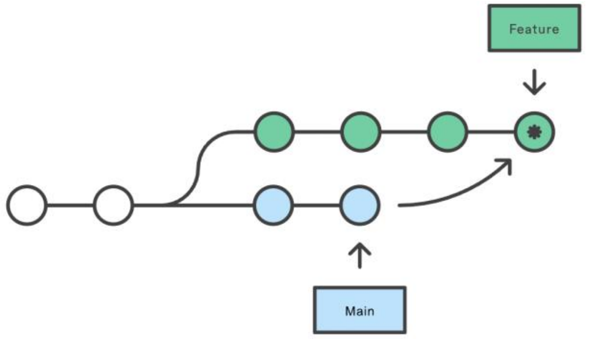


Possible scenarios:

1. Merge main into feature

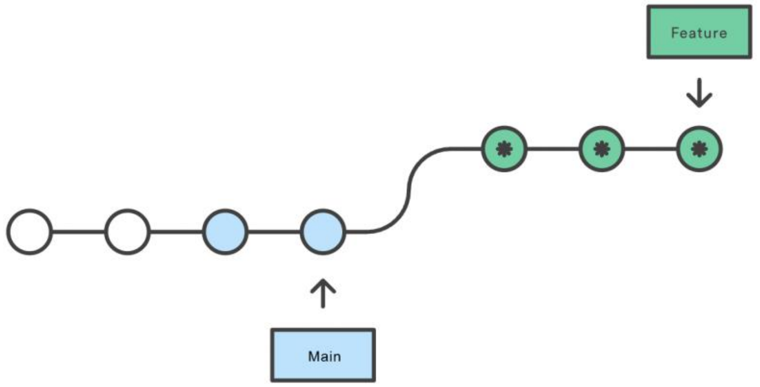


git checkout feature
git merge main



2. Rebase feature onto main

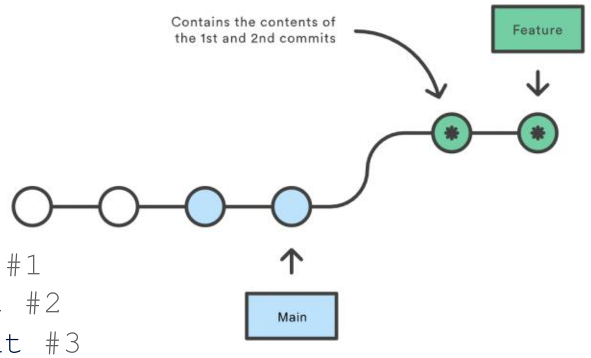
git checkout feature
git rebase main



```
pick 33d5b7a Message for commit #1
pick 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

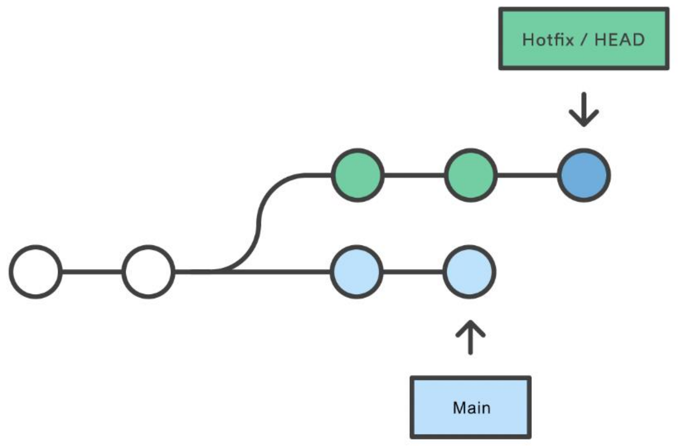
```
pick 33d5b7a Message for commit #1
fixup 9480b3d Message for commit #2
squash 5c67e61 Message for commit #3
```

Interactive rebase
git rebase -i main

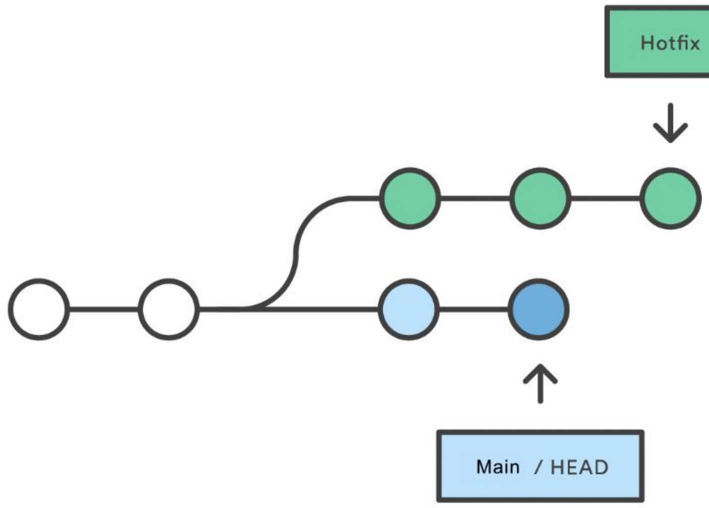


Reset/Checkout/Revert

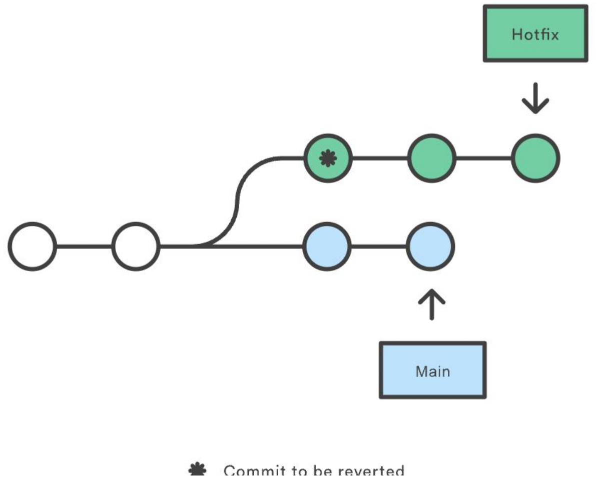
Before Resetting



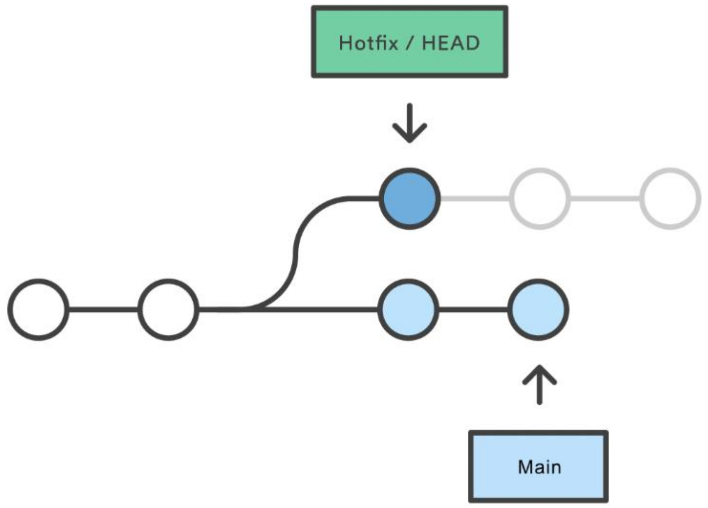
Before Checking Out



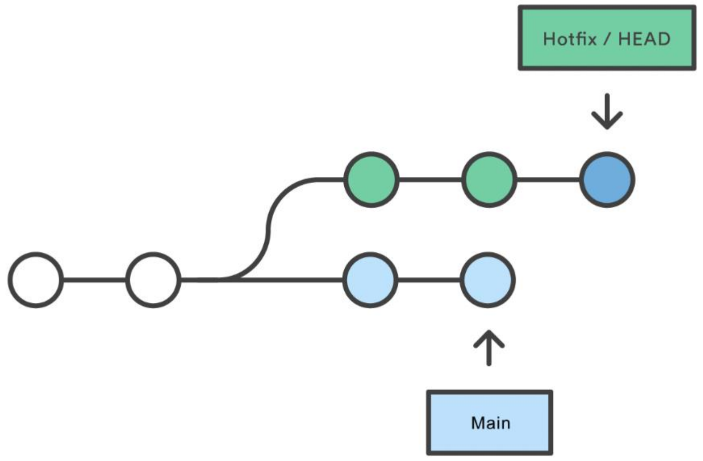
Before Reverting



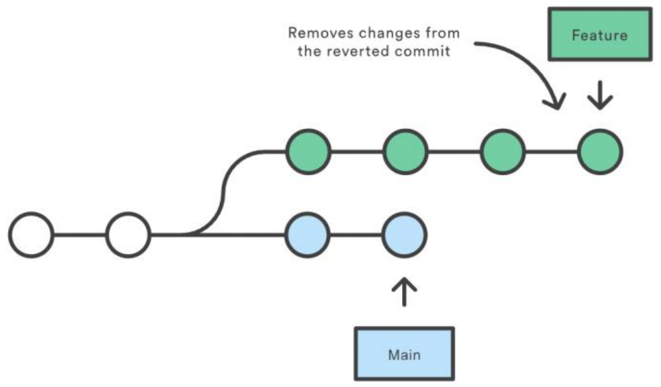
After Resetting



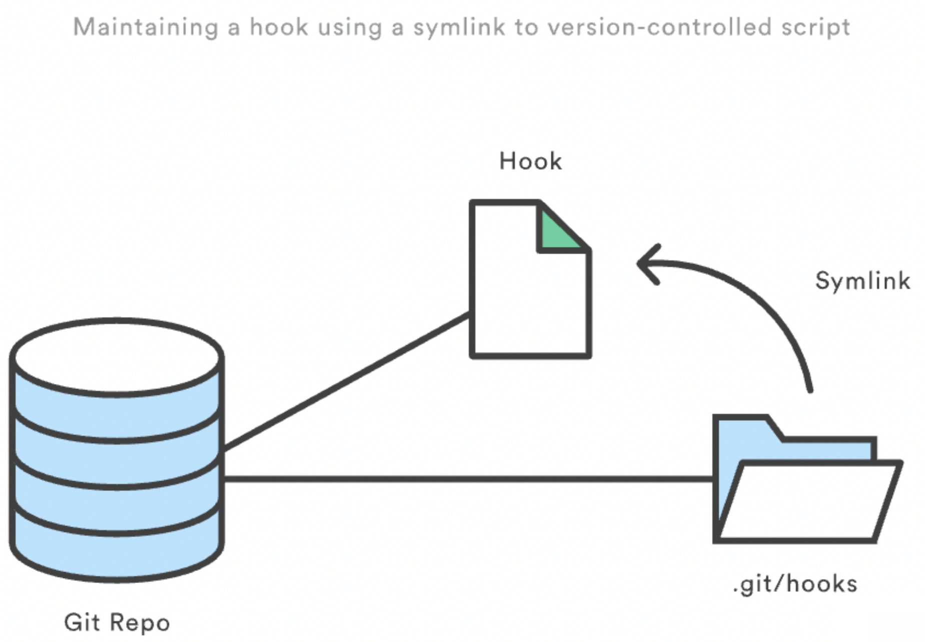
After Checking Out

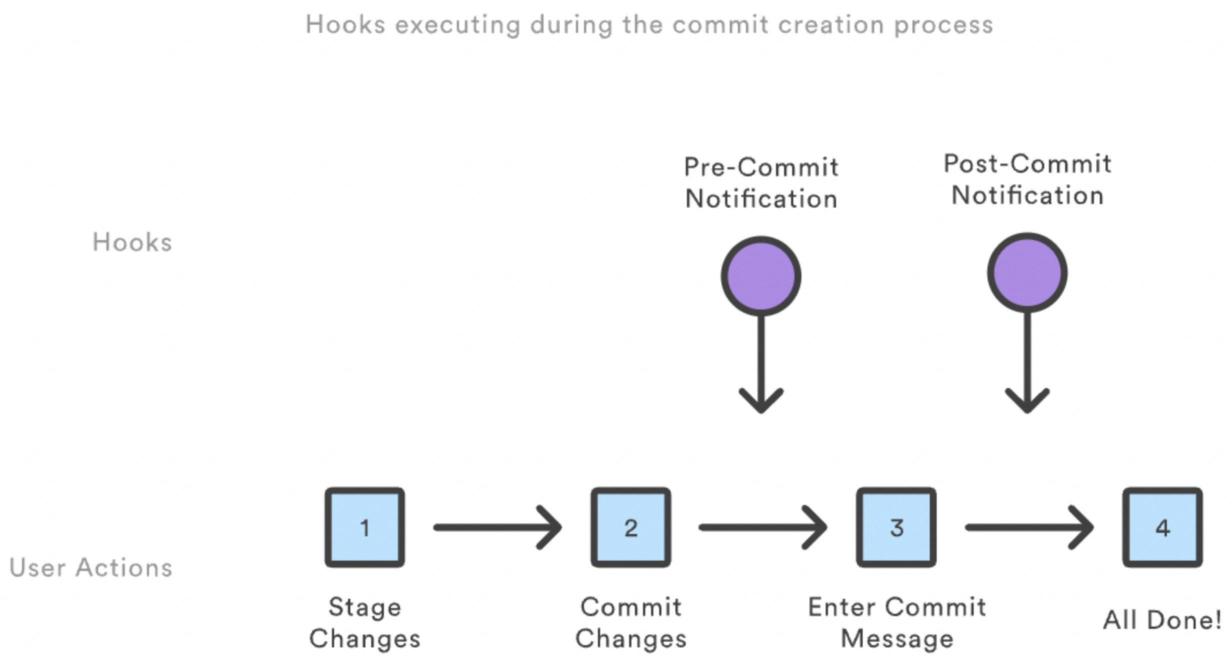


After Reverting

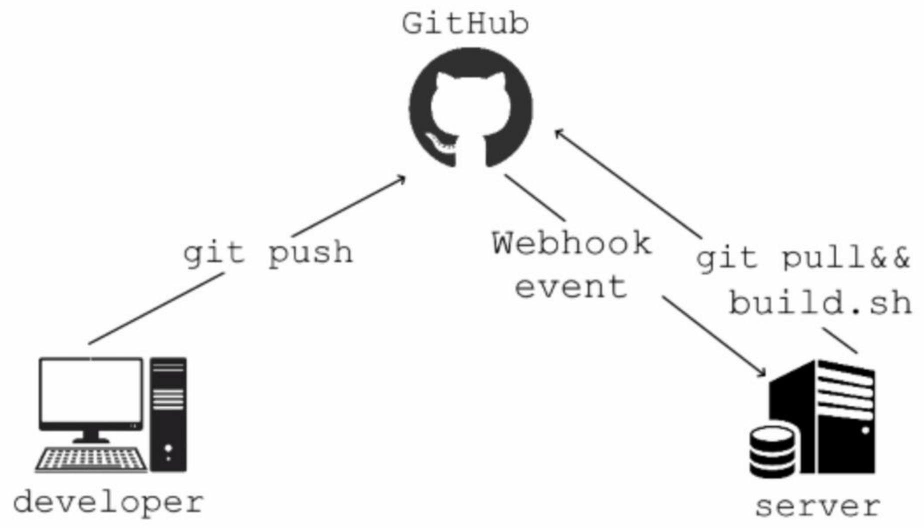


Git Hooks


Maintaining a hook using a symlink to version-controlled script
Hook
Symlink
Git Repo
.git/hooks



GitHub WebHooks



Best practices:

Reviewing a feature with a pull request

If you use pull requests as part of your code review process, you need to avoid using git rebase after creating the pull request. As soon as you make the pull request, other developers will be looking at your commits, which means that it’s a *public* branch. Re-writing its history will make it impossible for Git and your teammates to track any follow-up commits added to the feature. Any changes from other developers need to be incorporated with git merge instead of git rebase. For this reason, it’s usually a good idea to clean up your code with an interactive rebase *before* submitting your pull request.

Integrating an approved feature

After a feature has been approved by your team, you have the option of rebasing the feature onto the tip of the main branch before using git merge to integrate the feature into the main code base.

This is a similar situation to incorporating upstream changes into a feature branch, but since you’re not allowed to re-write commits in the main branch, you have to eventually use git merge to integrate the feature. However, by performing a rebase before the merge, you’re assured that the merge will be fast-forwarded, resulting in a perfectly linear history. This also gives you the chance to squash any follow-up commits added during a pull request.

2. Related Work and Background

2.1 Software Quality Models

Assessing software quality is a central concern in software engineering research, with various models developed to quantify and measure different aspects of quality. Traditional approaches, such as the ISO/IEC 25000:2014 (https://www.iso.org/standard/64764.html) series (derived from ISO/IEC 25002:2024 https://www.iso.org/standard/78175.html), provide structured quality evaluation frameworks focused on attributes like maintainability, reliability, and usability. These models define software quality from a product-oriented perspective but often overlook the dynamic aspects of software development, such as the role of development workflows, community activity, and the structure of version control histories. Several models extend these traditional frameworks to better suit the open-source software (OSS) landscape. The "Measuring the Quality of Open Source Software Ecosystems Using QuESo" (https://www.researchgate.net/publication/300335654_Measuring_the_Quality_of_Open_Sou rce_Software_Ecosystems_Using_QuESo) model was introduced to address sustainability and maturity in open-source projects by considering both technical and community-related factors. The model highlights the importance of process stability, development continuity, and project governance in assessing long-term software quality. Similarly, QuESo expands the scope by incorporating ecosystem-level quality metrics, including project interdependencies, contributor engagement, and software reuse. These ecosystem oriented models provide valuable insights into software quality at a higher level but do not focus on how specific development structures such as commit history topologies affect software maintainability and defect rates. The role of software ecosystems in ensuring the sustainability of OSS projects has been widely studied. The " QuESo" introduces a comprehensive framework for evaluating OSS ecosystems based on multiple quality dimensions, such as activity levels, stability, governance, and modularity. The study highlights that a well-structured and active software ecosystem contributes to the long live stability and reliability of OSS projects, as continuous contributions and frequent interactions between developers help maintain a project's health. Another relevant study, "Quality Evaluation Meta-Model for OSS – A Multi-Method Validation Study" https://www.researchgate.net/publication/378003488_Quality_evaluation_meta- model_for_open-source_software_multi-method_validation_study presents a meta-model for software quality assessment, creating multiple prior approaches into a unified framework. The study emphasizes the importance of historical project data, including commit activity, developer collaboration, and repository structure, as indicators of software sustainability and robustness. However, while these studies explore high-level ecosystem characteristics and development patterns, they do not directly investigate the commit structure as a directed acyclic graph (DAG). The DAG structure in version control systems encodes essential information about how a project evolves over time, capturing branching strategies, merge practices, and code integration workflows. Understanding how these structural properties correlate with software quality can provide new insights into best practices for OSS development.

Data Set Collection

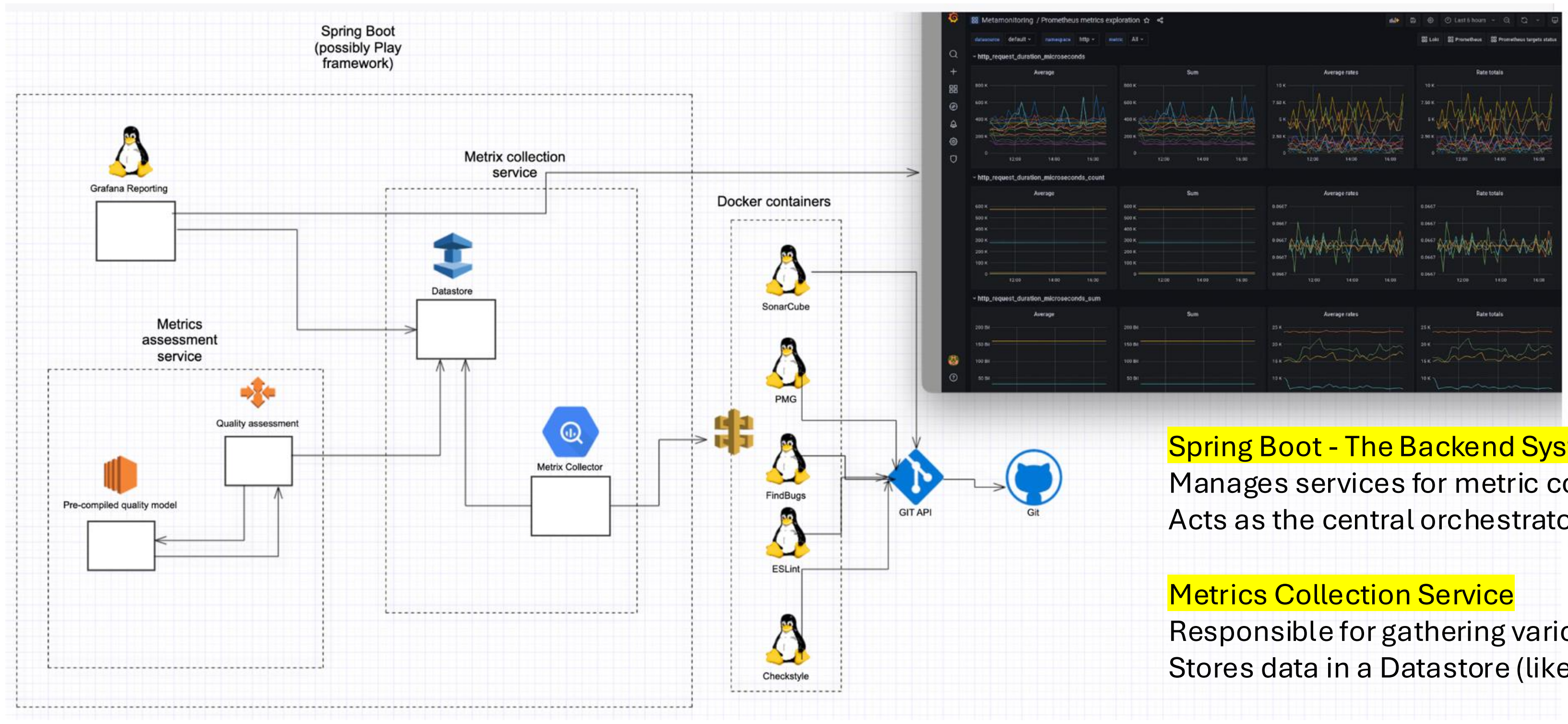
To study software evolution, bug-fixing patterns, and code maintenance, we need **real-world data**.
Without a dataset, research becomes **theoretical** and lacks empirical evidence (other researches are based on expert opinion in building evaluation model).
To ensure a comprehensive dataset, I selected:

- Top 5 GitHub projects based on popularity, activity, and long-term maintenance.
- Projects with at least 15+ years of history to capture long-term software evolution.
- A proportional number of pull requests (PRs) and issues, ensuring a balanced dataset with various software development patterns.
- Preference for repositories with high collaboration and structured development workflows.

Collecting data from GitHub is not straightforward due to API constraints:

- The GitHub API limits requests to 1,000 per hour per user, making large-scale data collection difficult.
- Workarounds such as caching responses and rate limit handling were implemented.
- Orphaned Commits & Forks
- Some commits do not belong to any branch, making them difficult to trace (orphaned commits).
- Implemented an enhanced SZZ algorithm to recover these orphaned commits.
- Data Integrity & Completeness
- Many PRs do not directly link to issues, requiring heuristics to infer bug-fixing commits.
- Actions & workflows in CI/CD pipelines add additional complexity in identifying changes triggered by bug fixes.

Architectural design:



Spring Boot - The Backend System
Manages services for metric collection and quality assessment.
Acts as the central orchestrator of the pipeline.

Metrics Collection Service
Responsible for gathering various software metrics.
Stores data in a Datastore (likely a database or time-series storage).

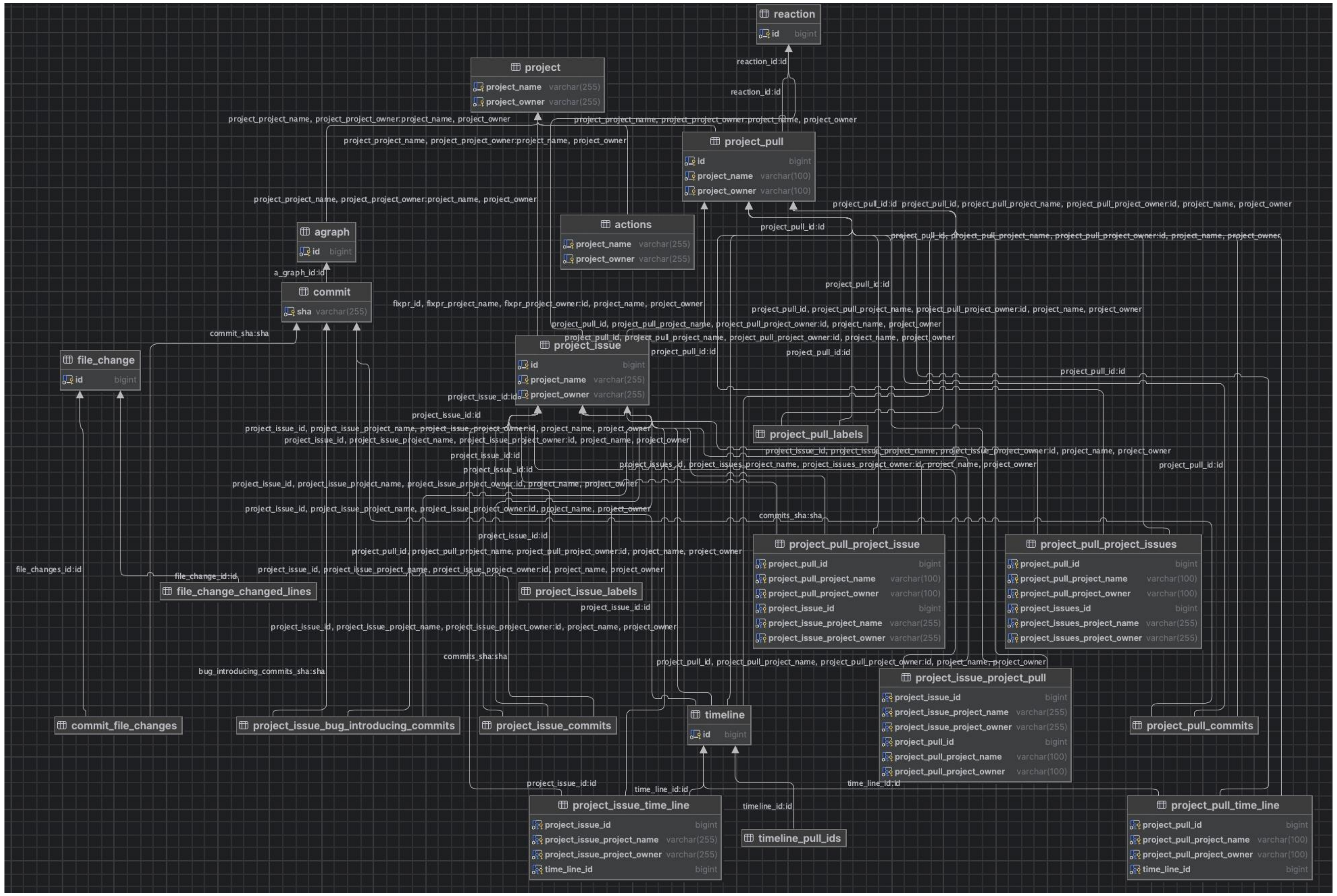
Docker Containers - Static Code Analysis Tools
SonarCube, PMG, FindBugs, ESLint, Checkstyle run inside Docker containers.
These tools analyze the code and generate reports on code quality, security, and bugs.

Metrics Assessment Service
Quality assessment component evaluates the collected data.
Uses a pre-compiled quality model to assess software quality.

Git API & GitHub Integration
Git API fetches repository data (e.g., commits, PRs, issues).

Reporting
Presents visual insights on collected data.

Proposed Schema


reaction
id bigint
project
project_name varchar(255)
project_owner varchar(255)
project_pull
id bigint
project_name varchar(100)
project_owner varchar(100)
actions
project_name varchar(255)
project_owner varchar(255)
agraph
id bigint
commit
sha varchar(255)
file_change
id bigint
project_issue
id bigint
project_name varchar(255)
project_owner varchar(255)
project_pull_labels
project_pull_project_issue
project_pull_id bigint
project_pull_project_name varchar(100)
project_pull_project_owner varchar(100)
project_issue_id bigint
project_issue_project_name varchar(255)
project_issue_project_owner varchar(255)
project_pull_project_issues
project_pull_id bigint
project_pull_project_name varchar(100)
project_pull_project_owner varchar(100)
project_issues_id bigint
project_issues_project_name varchar(255)
project_issues_project_owner varchar(255)
file_change_changed_lines
project_issue_labels
project_issue_project_pull
project_issue_id bigint
project_issue_project_name varchar(255)
project_issue_project_owner varchar(255)
project_pull_id bigint
project_pull_project_name varchar(100)
project_pull_project_owner varchar(100)
commit_file_changes
project_issue_bug_introducing_commits
project_issue_commits
timeline
id bigint
project_pull_commits
project_issue_time_line
project_issue_id bigint
project_issue_project_name varchar(255)
project_issue_project_owner varchar(255)
time_line_id bigint
timeline_pull_ids
project_pull_time_line
project_pull_id bigint
project_pull_project_name varchar(100)
project_pull_project_owner varchar(100)
time_line_id bigint

1. Project & Actions

project: Stores project names and owners as a composite primary key.
actions: Stores actions related to a project.

2. File Changes & Commits

file_change: Stores file change details such as additions, deletions, and patch data.
file_change_changed_lines: Links file changes to the number of lines changed.
commit: Stores commit details such as author, message, degree of connectivity, and associated graphs.
commit_file_changes: Links commits to file changes.

3. Graphs

agraph: Stores project graphs and links them to projects.

4. Reactions

reaction: Stores reactions (e.g., thumbs up, hearts) to issues or pull requests.

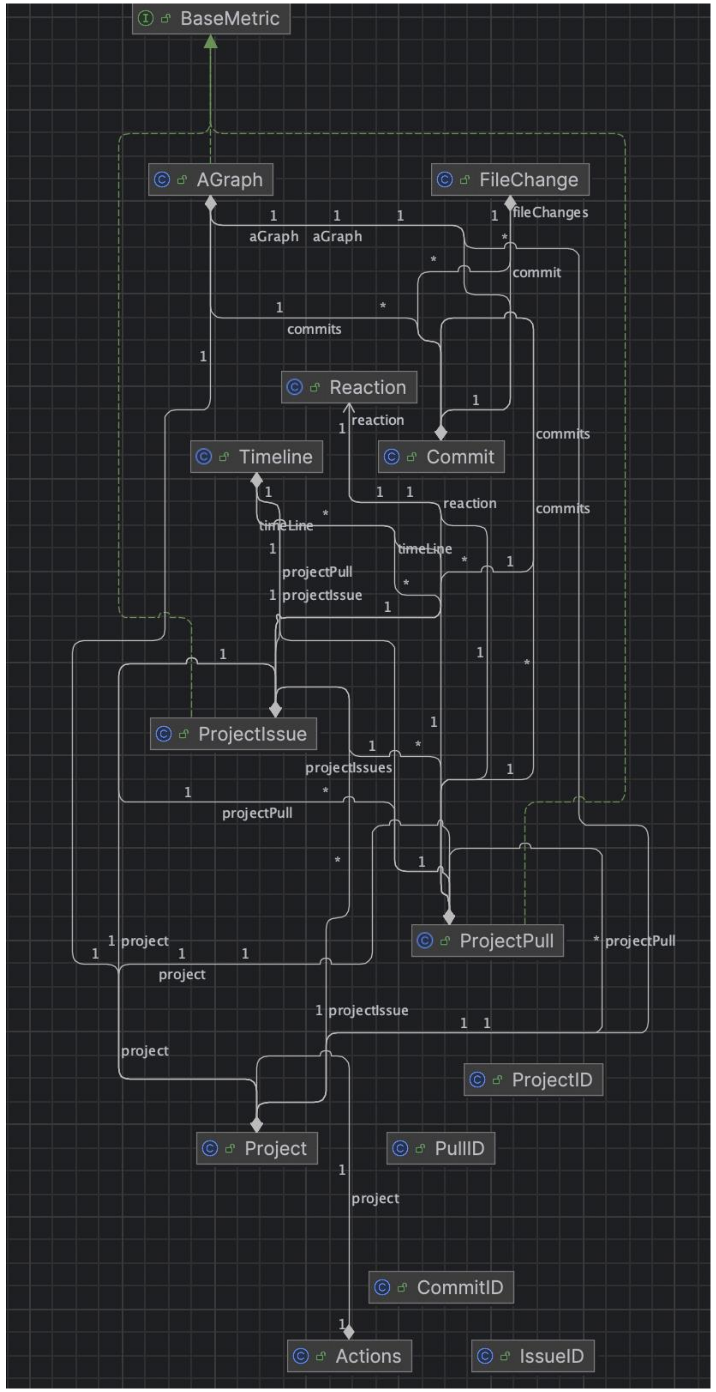
5. Project Issues & Pull Requests

project_issue: Stores issue details, linking them to projects.
project_pull: Stores pull request details, linking them to projects.
project_issue_bug_introducing_commits: Links issues to commits that introduced bugs.
project_issue_commits: Links issues to commits that reference them.
project_issue_labels: Stores labels associated with issues.
project_issue_project_pull: Links issues to pull requests.
project_pull_commits: Links pull requests to commits.
project_pull_labels: Stores labels associated with pull requests.
project_pull_project_issue: Links pull requests to issues.

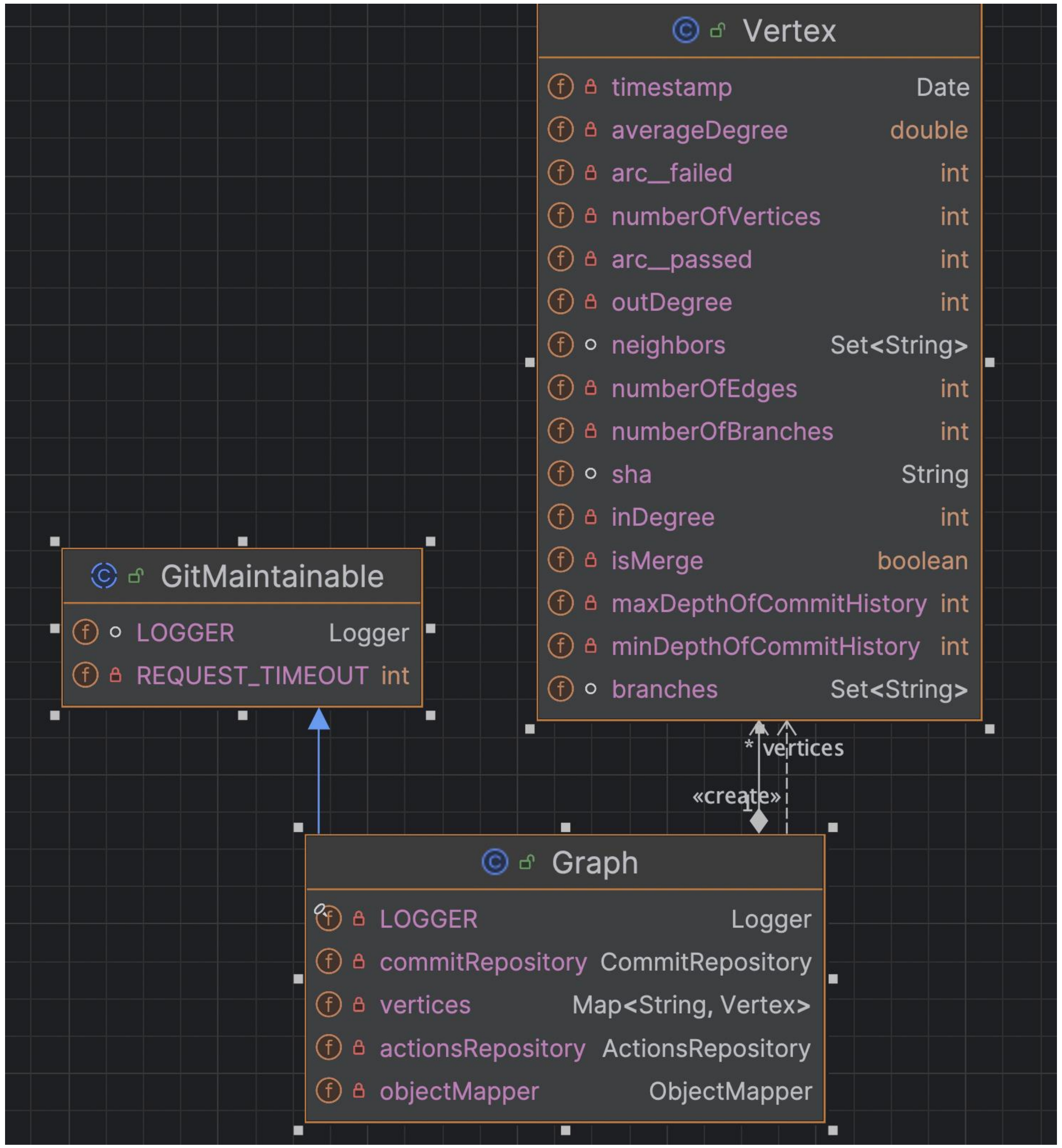
6. Timeline

timeline: Stores timestamps and messages related to issues or pull requests.
project_issue_time_line: Links issues to timeline events.
project_pull_time_line: Links pull requests to timeline events.
timeline_pull_ids: Associates timeline entries with pull requests.

ORM view

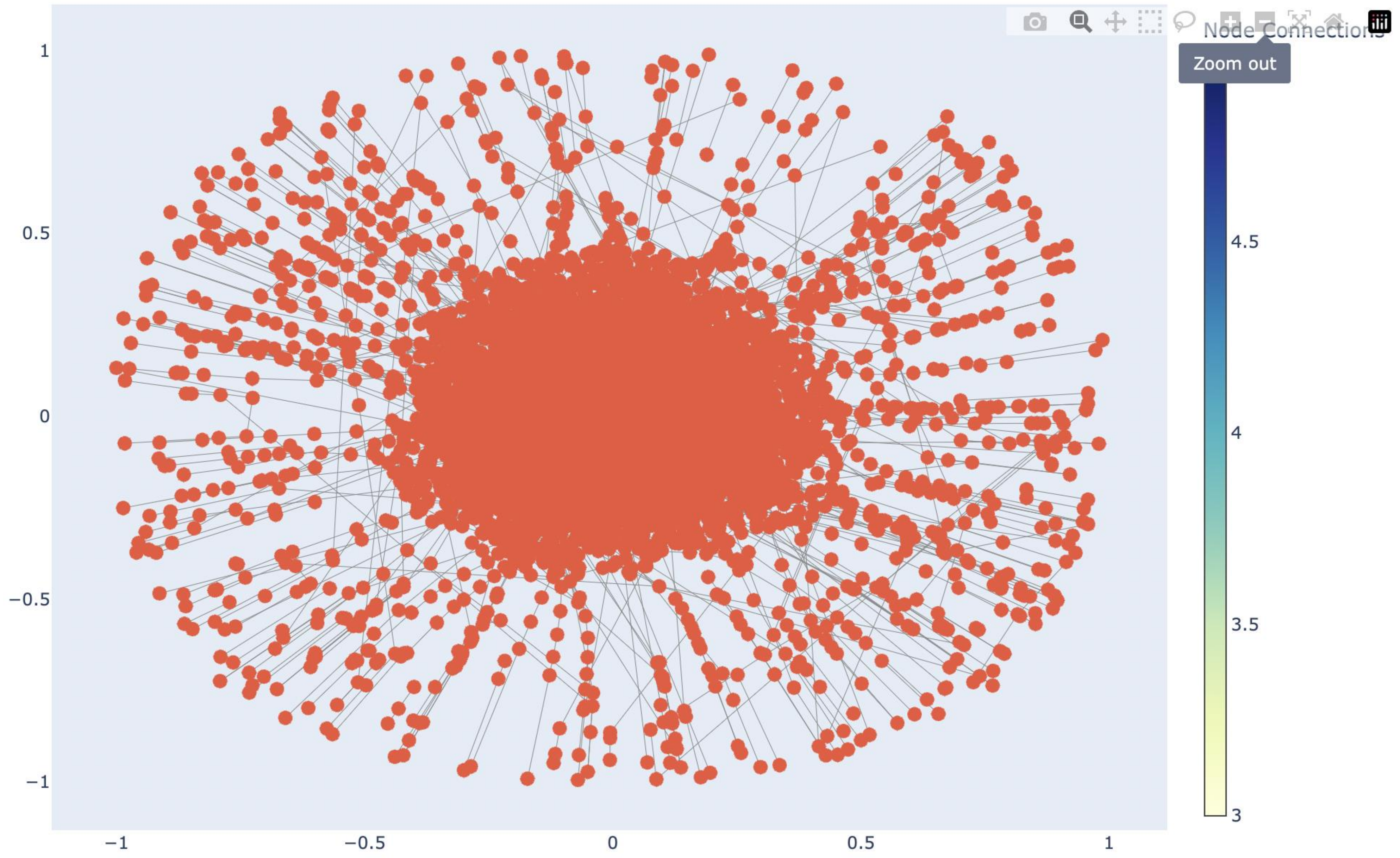

BaseMetric
AGraph
FileChange
Reaction
Timeline
Commit
ProjectIssue
ProjectPull
ProjectID
Project
PullID
CommitID
Actions
IssueID
aGraph
fileChanges
commit
commits
reaction
timeLine
projectPull
projectIssue
projectIssues
project

DAG of commits (possibly demo)

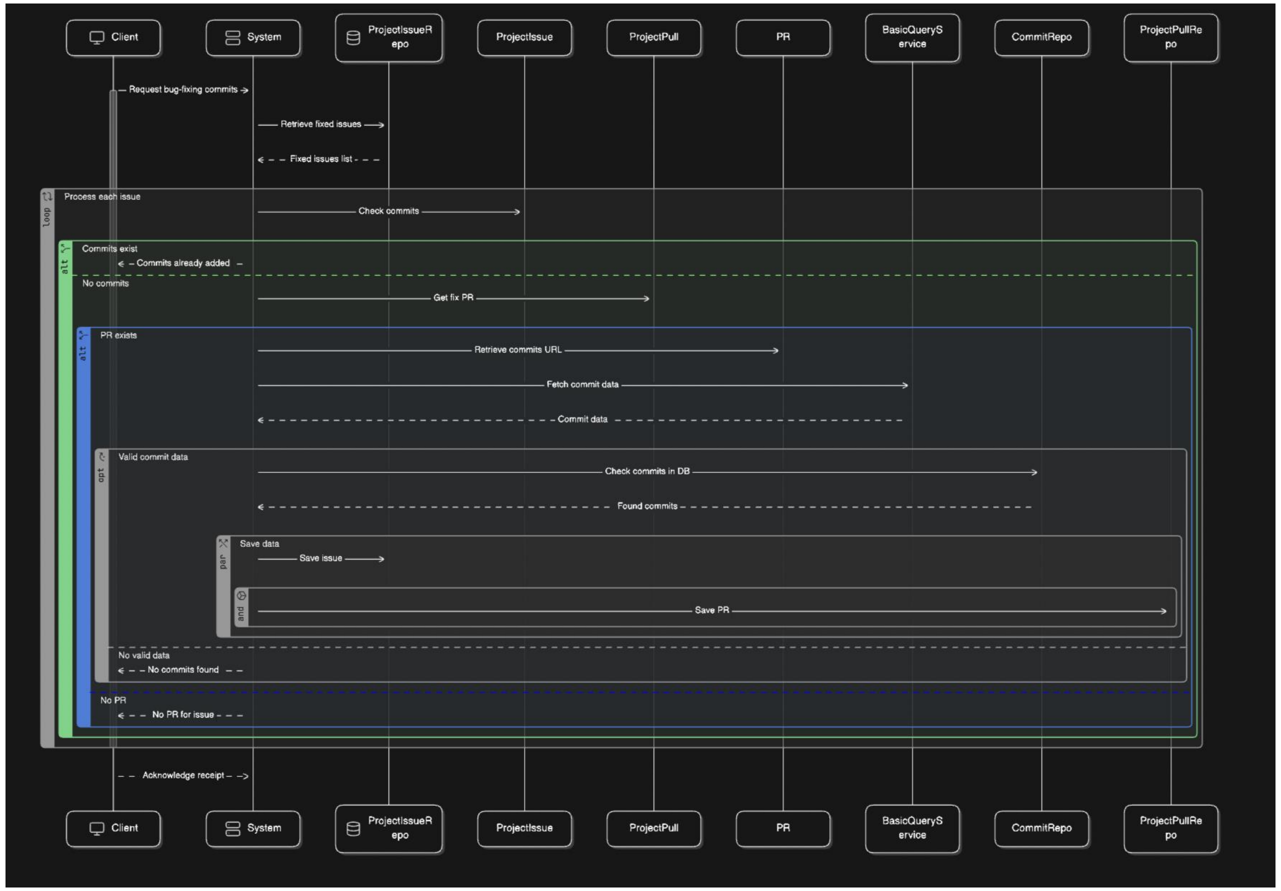


Adding Commits (Vertices):
Each commit (represented by SHA) is added as a vertex in the graph.
The Graph maintains a collection of Vertices (commits), each containing a unique SHA, neighbors (other commits that depend on it), branches (where it belongs), and degree metrics.
Building Commit Relationships (Edges):
When a commit has parent commits (e.g., for merges), the Graph will add edges between these commits.
The addEdge method establishes these relationships and updates the in-degree and out-degree properties of the vertices.
Tracking Commits:
The findParents method finds all parent commits for a given commit (sha).
The findChildren method finds all child commits.
The findCommonAncestors method finds common ancestors between multiple commits, helping identify shared history in the commit graph.
Calculating Metrics:
The Graph computes metrics related to the graph structure (e.g., number of commits, branches, edges).
Vertex tracks the number of neighbors (out-degree) and the number of incoming edges (in-degree), helping measure graph complexity.
Commit Analysis:
The system analyzes commit depth (max/min depth), branching strategies, and merge count, which are stored as properties in each Vertex.
The updateActionResult method links build results (passed/failed) with commits, updating the commit graph accordingly.

Example of graph for ansible project



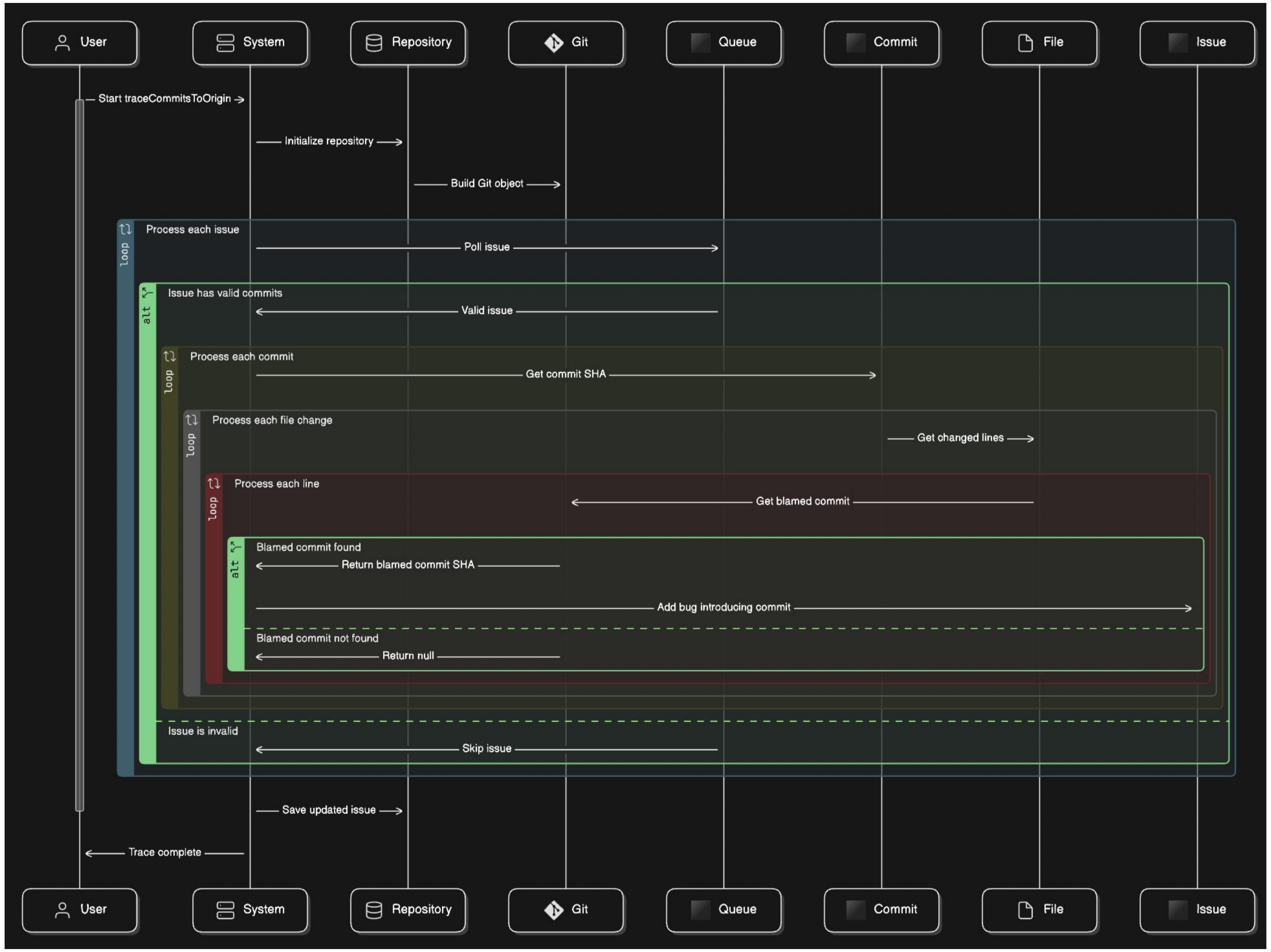
Commits that fix the issue:



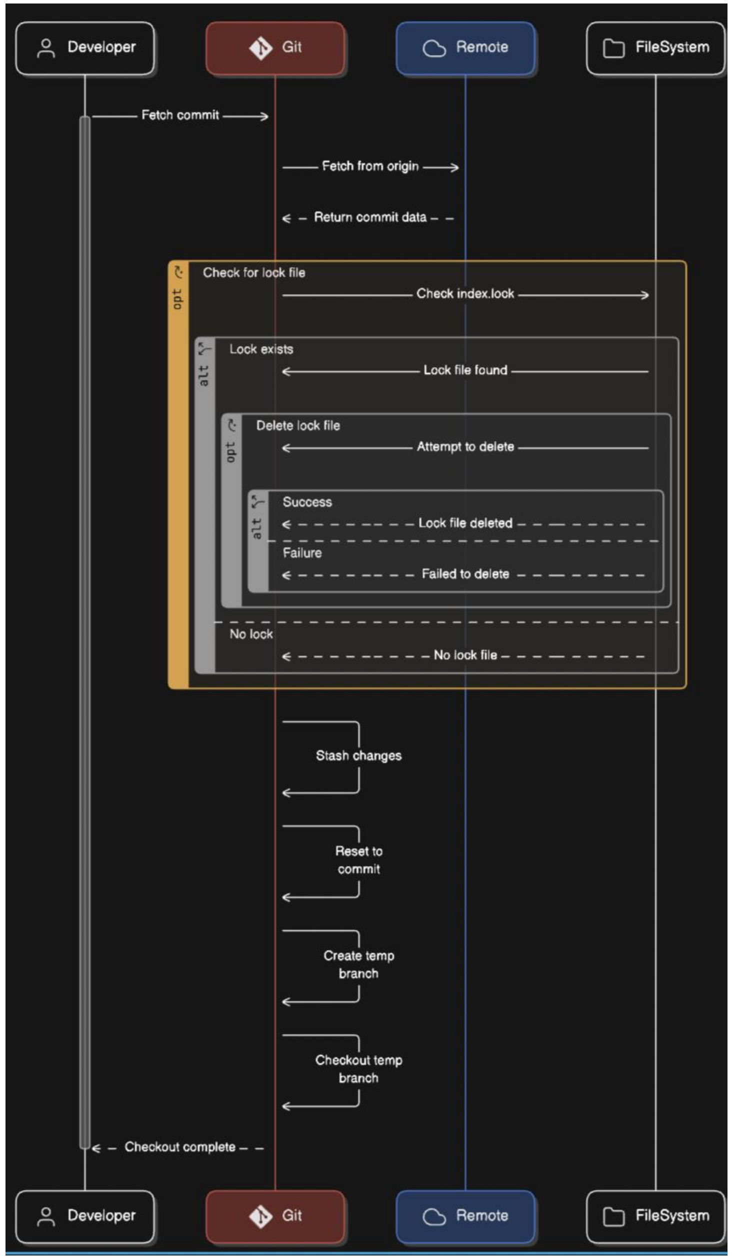
Ex. Find bug fixing commits
https://github.com/facebook/react/issues/10
https://github.com/facebook/react/pull/11
https://github.com/facebook/react/pull/11/commits

SZZ Algorythm (Software Bug-Inducing Commit Identification)

The algorithm tracks bug-fixing commits (commits that fix an issue) and then traces backward through the commit history to identify the bug-introducing commits



Handle orphaned commits:



Orphaned commits example:

https://github.com/google/guava/commit/0ceaed084bb7e52d8fa3b0760cb0ffe486918a00

Working with the dataset

Issues Opened:

```
select concat(project_owner, '_', project_project_name) as project, date(created_at) as time, count(*) as issues_opened from project_issue group by project_owner, project_project_name,
date(created_at) order by date(created_at) asc;
```

Issues Closed:

```
SELECT CONCAT(project_owner, '_', project_project_name) AS project, DATE(closed_at) AS time, COUNT(*) AS issues_closed FROM project_issue WHERE closed_at IS NOT NULL GROUP
BY project_owner, project_project_name, DATE(closed_at) ORDER BY DATE(closed_at) ASC;
```

Total Developers Over Time:

```
SELECT CONCAT(p.project_owner, '_', p.project_name) AS project, DATE(c.commit_date) AS time, COUNT(DISTINCT c.author) AS num_developers FROM commit c JOIN agraph ag ON
c.a_graph_id = ag.id JOIN project p ON ag.project_project_owner = p.project_owner AND ag.project_project_name = p.project_name GROUP BY p.project_owner, p.project_name,
DATE(c.commit_date) ORDER BY DATE(c.commit_date) ASC;
```

Total Developers:

```
SELECT ag.project_project_owner AS repo_owner, ag.project_project_name AS repo_project_name, COUNT(DISTINCT c.author) AS num_developers FROM commit c JOIN agraph ag ON
c.a_graph_id = ag.id GROUP BY ag.project_project_owner, ag.project_project_name ORDER BY num_developers DESC;
```

Total Deletions Per Commit Over Time:

```
SELECT CONCAT(p.project_owner, '_', p.project_name) AS project, DATE(c.commit_date) AS time, SUM(fc.total_deletions) AS total_deletions FROM file_change fc JOIN
commit_file_changes cfc ON fc.id = cfc.file_changes_id JOIN commit c ON cfc.commit_sha = c.sha JOIN agraph ag ON c.a_graph_id = ag.id JOIN project p ON ag.project_project_owner =
p.project_owner AND ag.project_project_name = p.project_name GROUP BY p.project_owner, p.project_name, c.commit_date ORDER BY c.commit_date ASC;
```

Total Changes Per Commit Over Time:

```
SELECT CONCAT(p.project_owner, '_', p.project_name) AS project, DATE(c.commit_date) AS time, SUM(fc.total_changes) AS total_changes FROM file_change fc JOIN commit_file_changes
cfc ON fc.id = cfc.file_changes_id JOIN commit c ON cfc.commit_sha = c.sha JOIN agraph ag ON c.a_graph_id = ag.id JOIN project p ON ag.project_project_owner = p.project_owner AND
ag.project_project_name = p.project_name GROUP BY p.project_owner, p.project_name, c.commit_date ORDER BY c.commit_date ASC;
```

Total Additions Per Commit Over Time:

```
SELECT CONCAT(p.project_owner, '_', p.project_name) AS project, DATE(c.commit_date) AS time, SUM(fc.total_additions) AS total_additions FROM file_change fc JOIN
commit_file_changes cfc ON fc.id = cfc.file_changes_id JOIN commit c ON cfc.commit_sha = c.sha JOIN agraph ag ON c.a_graph_id = ag.id JOIN project p ON ag.project_project_owner =
p.project_owner AND ag.project_project_name = p.project_name GROUP BY p.project_owner, p.project_name, c.commit_date ORDER BY c.commit_date ASC;
```

Commit Activity Over Time:

```
SELECT CONCAT(p.project_owner, '_', p.project_name) AS project, DATE(c.commit_date) AS time, COUNT(c.sha) AS commitCount FROM commit c JOIN agraph ag ON c.a_graph_id = ag.id
JOIN project p ON p.project_owner = ag.project_project_owner AND p.project_name = ag.project_project_name GROUP BY p.project_owner, p.project_name, time ORDER BY time ASC;
```

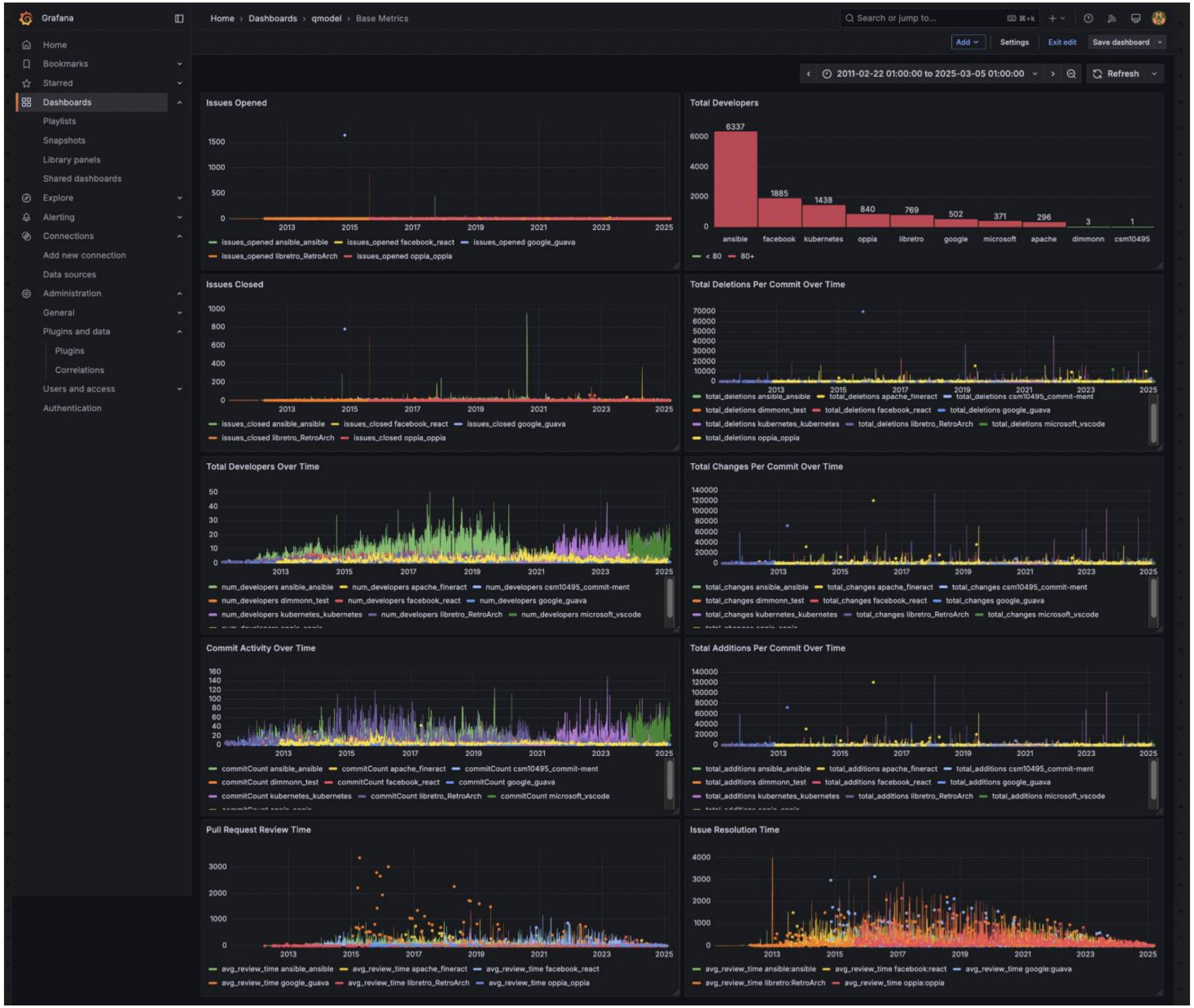
Pull Request Review Time:

```
SELECT CONCAT(project_owner, '_', project_project_name) AS project, DATE(created_at) AS time, AVG(TIMESTAMPDIFF(DAY, created_at, closed_at)) AS avg_review_time FROM project_pull
where project_pull.closed_at IS NOT NULL GROUP BY project_owner, project_project_name, time ORDER BY time ASC;
```

Issue Resolution Time:

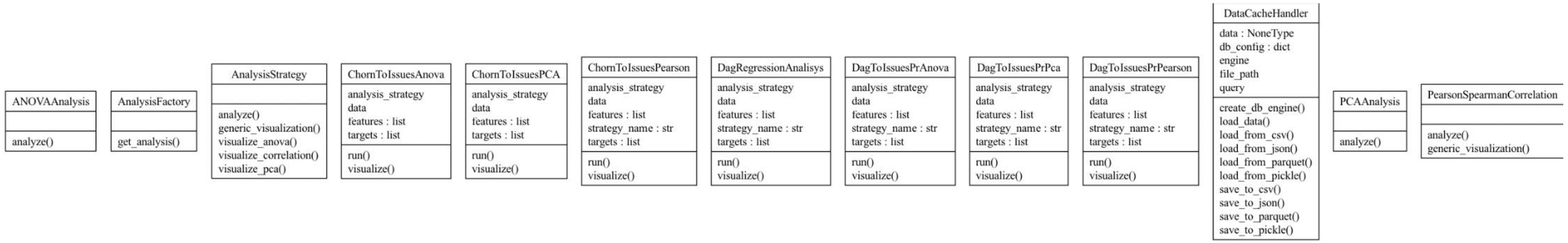
```
SELECT CONCAT(project_owner, ':', project_project_name) AS project, DATE(created_at) AS time, AVG(TIMESTAMPDIFF(DAY, created_at, closed_at)) AS avg_review_time FROM
project_issue where project_issue.closed_at IS NOT NULL GROUP BY project_owner, project_project_name, time ORDER BY time ASC;
```

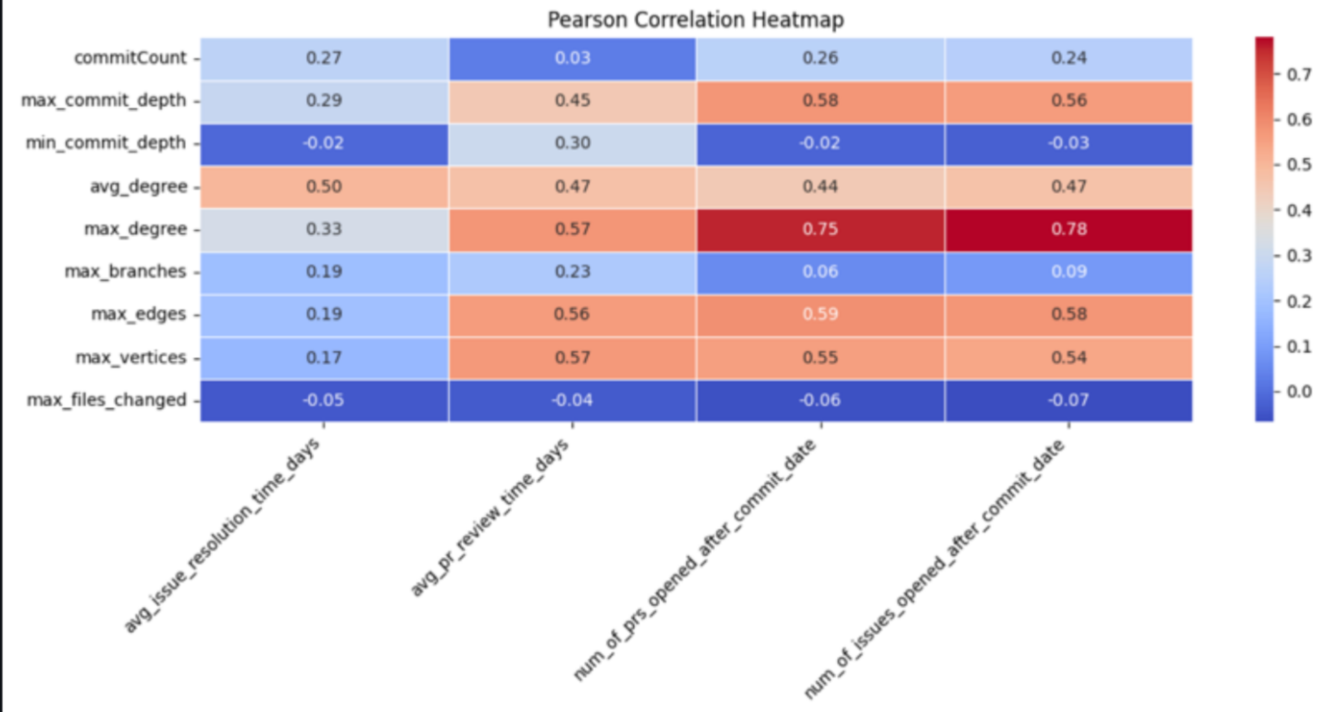
Reporting

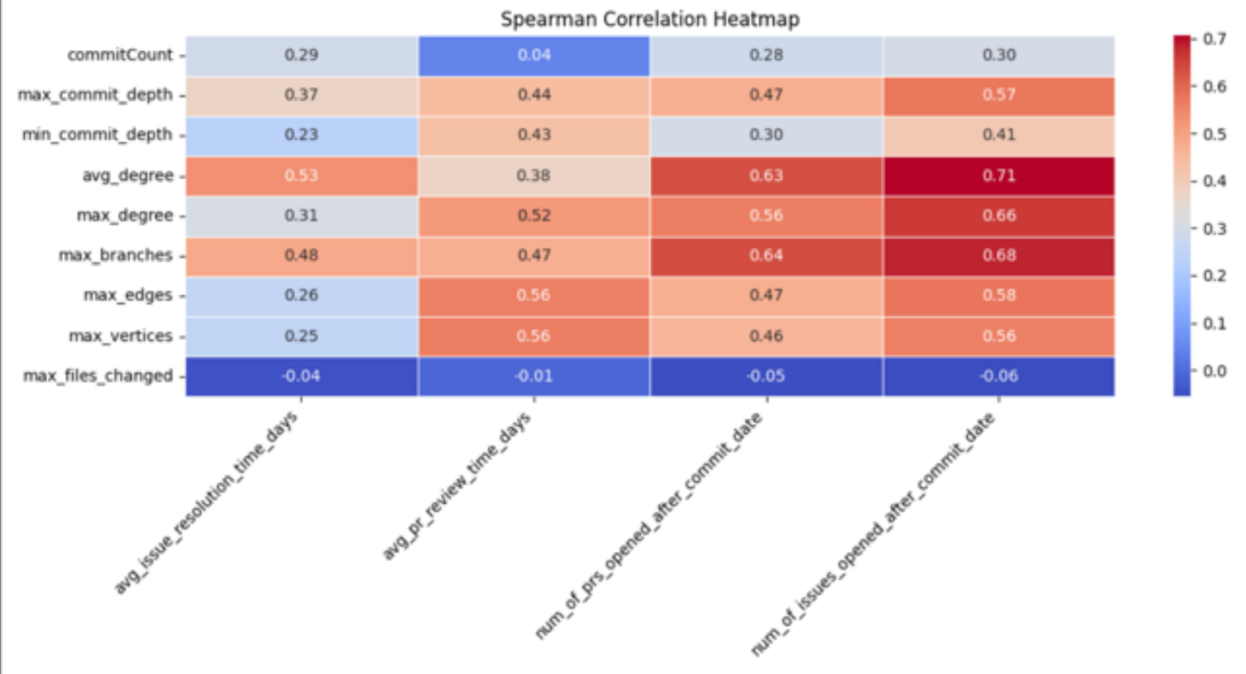

Grafana
Home › Dashboards › qmodel › Base Metrics
Search or jump to...
Add
Settings
Exit edit
Save dashboard
2011-02-22 01:00:00 to 2025-03-05 01:00:00
Refresh
Home
Bookmarks
Starred
Dashboards
Playlists
Snapshots
Library panels
Shared dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration
General
Plugins and data
Plugins
Correlations
Users and access
Authentication
Issues Opened
issues_opened ansible_ansible
issues_opened facebook_react
issues_opened google_guava
issues_opened libretro_RetroArch
issues_opened oppia_oppia
Total Developers
6337
1885
1438
840
769
502
371
296
3
1
ansible
facebook
kubernetes
oppia
libretro
google
microsoft
apache
dimmonn
csm10495
< 80
80+
Issues Closed
issues_closed ansible_ansible
issues_closed facebook_react
issues_closed google_guava
issues_closed libretro_RetroArch
issues_closed oppia_oppia
Total Deletions Per Commit Over Time
total_deletions ansible_ansible
total_deletions apache_fineract
total_deletions csm10495_commit-ment
total_deletions dimmonn_test
total_deletions facebook_react
total_deletions google_guava
total_deletions kubernetes_kubernetes
total_deletions libretro_RetroArch
total_deletions microsoft_vscode
total_deletions oppia_oppia
Total Developers Over Time
num_developers ansible_ansible
num_developers apache_fineract
num_developers csm10495_commit-ment
num_developers dimmonn_test
num_developers facebook_react
num_developers google_guava
num_developers kubernetes_kubernetes
num_developers libretro_RetroArch
num_developers microsoft_vscode
Total Changes Per Commit Over Time
total_changes ansible_ansible
total_changes apache_fineract
total_changes csm10495_commit-ment
total_changes dimmonn_test
total_changes facebook_react
total_changes google_guava
total_changes kubernetes_kubernetes
total_changes libretro_RetroArch
total_changes microsoft_vscode
Commit Activity Over Time
commitCount ansible_ansible
commitCount apache_fineract
commitCount csm10495_commit-ment
commitCount dimmonn_test
commitCount facebook_react
commitCount google_guava
commitCount kubernetes_kubernetes
commitCount libretro_RetroArch
commitCount microsoft_vscode
Total Additions Per Commit Over Time
total_additions ansible_ansible
total_additions apache_fineract
total_additions csm10495_commit-ment
total_additions dimmonn_test
total_additions facebook_react
total_additions google_guava
total_additions kubernetes_kubernetes
total_additions libretro_RetroArch
total_additions microsoft_vscode
Pull Request Review Time
avg_review_time ansible_ansible
avg_review_time apache_fineract
avg_review_time facebook_react
avg_review_time google_guava
avg_review_time libretro_RetroArch
avg_review_time oppia_oppia
Issue Resolution Time
avg_review_time ansible:ansible
avg_review_time facebook:react
avg_review_time google:guava
avg_review_time libretro:RetroArch
avg_review_time oppia:oppia

Data Analisys

https://github.com/dimmonn/qmode_compilation







Higher complexity in commit structures (max_commit_depth, max_degree, avg_degree) is strongly associated with longer issue resolution times and higher PR review times.

More interconnected commit graphs (higher max_degree) correspond with more PRs and issues being opened after commit dates.

The number of branches (max_branches) and number of edges (max_edges) are also positively correlated with issue and PR-related metrics.

Next steps:

Research relations:

- Does the degree of a node (number of merges) correlate with the number of errors in actions in a CI pipeline?
- Does the number of branches commit went through correlate with the number of issues it could cause?
- Do commits with more than one parent (merge commits) have a significantly different number of PRs compared to linear commits?
- Do projects with different numbers of branches have different CI pipeline execution errors?
- What are the main factors influencing repository complexity? (e.g., branches, PRs, actions)
- Try to group repositories based on their commit structure