Metrics

Release 2021-03

https://chaoss.community/metrics

Contents

Evolution WG **2**
- Focus Area - Code-Development-Activity . . . 3
  - Branch Lifecycle . . . 4
  - Code Changes Lines . . . 7
  - Code Changes . . . 10
- Focus Area - Code-Development-Efficiency . . . 13
  - Change Request Acceptance Ratio . . . 14
  - Change Requests Accepted . . . 17
  - Change Requests Declined . . . 20
  - Change Requests Duration . . . 22
- Focus Area - Code-Development-Process-Quality . . . 24
  - Change Requests . . . 25
- Focus Area - Community-Growth . . . 28
  - Contribution Attribution . . . 29
  - Inactive Contributors . . . 32
  - New Contributors Closing Issues . . . 33
  - New Contributors . . . 34
- Focus Area - Issue-Resolution . . . 36
  - Issue Age . . . 37
  - Issue Resolution Duration . . . 39
  - Issue Response Time . . . 41
  - Issues Active . . . 43
  - Issues Closed . . . 46
  - New Issues . . . 49

Evolution WG

Focus Area - Code-Development-Activity

Goal: Learn about the types and frequency of activities involved in developing code.

Metric	Question
Branch Lifecycle	How do projects manage the lifecycle of their version control branches?
Code Changes Lines	What is the sum of the number of lines touched (lines added plus lines removed) in all changes to the source code during a certain period?
Code Changes	How many changes were made to the source code during a specified period?

Branch Lifecycle

Question: How do projects manage the lifecycle of their version control branches?

Description

This metric makes the lifecycle of version control branches visible. A branch lifecycle includes acts of branch creation and deletion, as well as persistence of version control branches. When writing code, a development team may create multiple branches, focused around specific features. Subsequently, those branches may be merged into more persistent branches, such as the main branch of a repository. Some branches persist for the life of the repository, while others are deleted after code is merged into a more persistent branch. By understanding these patterns, we can learn how branch creation, destruction, and merging reflect the development practices of a particular project. A repository's typical branch lifecycle and management approach can be used to help identify a project's repository management style.

Objectives

This metric's objective is to increase the visibility of a repository's volume and frequency of branch creation and deletion, as well as persistence of version control branches, in the context of other project characteristics. In turn, this helps potential contributors understand how to initiate and sustain participation in a project. This metric may help potential contributors understand how to initiate and sustain participation in a project. Questions that can be addressed through this metric include:

- How many branches have been merged but not deleted?
- How long has a branch been merged and not deleted?
- How many branches have existed longer than a certain number of days, months, or years?
- How often do projects or repositories create branches?
- In the aggregate, how long do branches usually live?
- How can we distinguish between branch "death" (i.e., never intended to be used again; deletion) or branch "dormancy" (i.e., inactive for long periods of time, but may be used again) in cases where branches are infrequently deleted in a repository?

Implementation

The stated advice regarding management of the branch lifecycle for a project may be visible in a CONTRIBUTING.md document, and these documents can be compared across repositories using linguistic analysis, and contrasted with data derived from actual project practices to draw insights within, and across repositories. In most cases, however, the data we focus on in this metric is quantitative in nature.

Aggregators:

- Count of branches created.
- Count of branches deleted.
- Count of branches merged.

- Count of branches abandoned (had unique commits, but never got merged before it was deleted)
- Total count of branches.
- Average age of open branches.
- Rate/ratio at which branches are created vs. deleted.

Parameters:

- Period of time. Start and finish date of the period. Default: forever.
- Period during which change requests are considered.

Filters (optional)

- Collections of repositories
- Default branch name versus descriptive name with regard to branch persistence

Tools Providing the Metric (optional)

Metric must be currently deployed/available, in contrast to a tool having the ”potential” to provide the metric. Provide direct link to implementation/documentation, if applicable

Augur Community Reports (https://github.com/chaoss/augur-community-reports)

Data Collection Strategies (Optional)

Specific description: Git

Git branching data exists at several different levels in a version control ecosystem, usually both locally on a developer’s machine as well on a hosted platform like GitHub or BitBucket. This branch data on the hosted platform may also differ from each developer’s machine, and additionally, different developers may have different branch data on their machine even for the same repository.

In the specific case of Git, a significant amount of additional complexity is introduced due to Git’s design as a distributed version control system, which means that Git allows multiple remotes for a single repository (for example, a user’s fork at github.com/user/project and the upstream version at github.com/chaoss/project). More often than not, many individual contributors may work in the same branch locally, and push changes to the remote repository. The local copies, therefore, will sometimes be different than the remote, hosted internally or on platforms like GitHub, GitLab, and BitBucket., since they’re likely either being managed by different people (likely with different branching styles) or they are both being used by one person to “silo” the work they are doing.

Data about Git branches can be derived from a Git log directly, or through a Git platform’s API.

Is picking a “canonical” version of the repo for branching data is useful? This gives a meaningful base for comparing between instances of the repo with different data. Could also prove to

`git branch -a` will list all existing branches.

References

Blog posts, websites, academic papers, or books that mention the metric and provide more background.

Adopting a Git Branching Strategy: https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops

Choose the Right Git Branching Strategy: https://www.creativebloq.com/web-design/choose-right-git-branching-strategy-121518344

The Effect of Branching Strategies on Software Quality by Emad Shihab, Christian Bird, and Thomas Zimmermann https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/shihab-esem-2012.pdf

OpenStack: https://docs.openstack.org/project-team-guide/other-branches.html

Code Changes Lines

Question: What is the sum of the number of lines touched (lines added plus lines removed) in all changes to the source code during a certain period?

Description

When introducing changes to the source code, developers touch (edit, add, remove) lines of the source code files. This metric considers the aggregated number of lines touched by changes to the source code performed during a certain period. This means that if a certain line in a certain file is touched in three different changes, it will count as three lines. Since in most source code management systems it is difficult or impossible to tell accurately if a lines was removed and then added, or just edited, we will consider editing a line as removing it and later adding it back with a new content. Each of those (removing and adding) will be considered as ”touching”. Therefore, if a certain line in a certain file is edited three times, it will count as six different changes (three removals, and three additions).

For this matter, we consider changes to the source code as defined in Code Changes. Lines of code will be any line of a source code file, including comments and blank lines.

Objectives

- Volume of coding activity:
  Although code changes can be a proxy to the coding activity of a project, not all changes are the same. Considering the aggregated number of lines touched in all changes gives a complementary idea of how large the changes are, and in general, how large is the volume of coding activity.

Implementation

Aggregators:

- Count. Total number of lines changes (touched) during the period.

Parameters:

- **Period of time:** Start and finish date of the period. Default: forever.
  Period during which changes are considered.
- **Criteria for source code; Algorithm Default:** all files are source code.
  If we are focused on source code, we need a criterion for deciding whether a file is a part of the source code or not.
- **Type of source code change:**
  - Lines added
  - Lines removed
  - Whitespace

Filters

- By actors (author, committer). Requires actor merging (merging ids corresponding to the same author).

- By groups of actors (employer, gender...). Requires actor grouping, and likely, actor merging.
- By tags (used in the message of the commits). Requires a structure for the message of commits. This tag can be used in an open-source project to communicate to every contributors if the commit is, for example, a fix for a bug or an improvement of a feature.

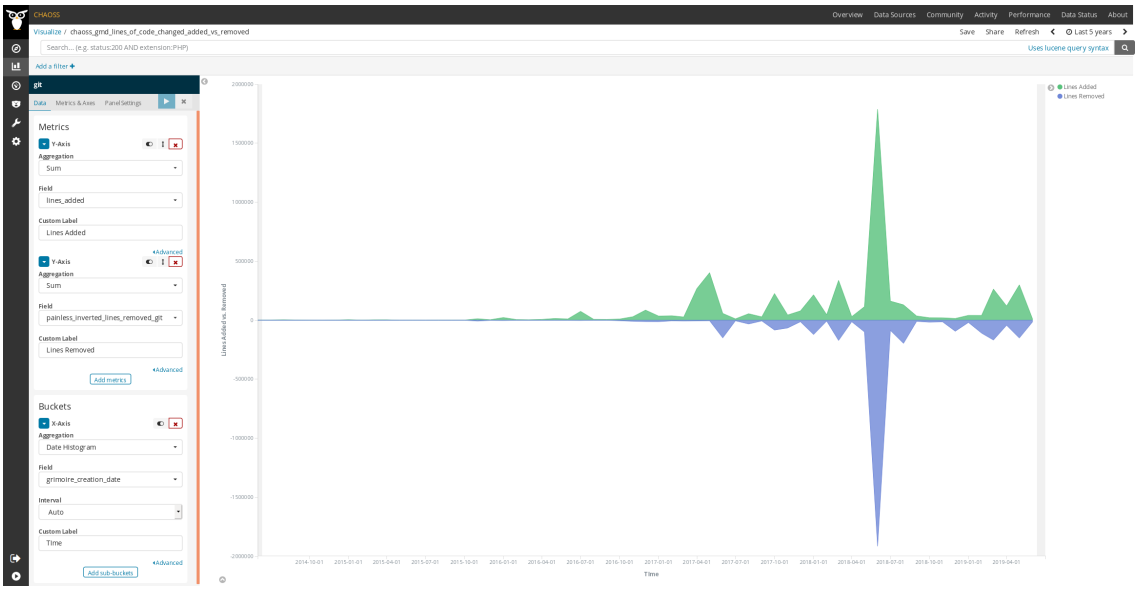
Visualizations

- Count per month over time
- Count per group over time

These could be represented as bar charts, with time running in the X axis. Each bar would represent a code changes during a certain period (eg, a month).

Tools Providing the Metric

- GrimoireLab provides this metric out of the box.
  - View an example on the CHAOSS instance of Bitergia Analytics.
  - Download and import a ready-to-go dashboard containing examples for this metric visualization from the GrimoireLab Sigils panel collection.
  - Add a sample visualization to any GrimoreLab Kibiter dashboard following these instructions:
    - Create a new `Area` chart
    - Select the `git` index
    - Y-axis 1: `Sum` Aggregation, `lines_added` Field, `Lines Added` Custom Label
    - Y-axis 2: `Sum` Aggregation, `painless_inverted_lines_removed_git` Field, `Lines Removed` Custom Label
    - X-axis: `Date Histogram` Aggregation, `grimoire_creation_date` Field, `Auto` Interval, `Time` Custom Label
  - Example screenshot:



Data Collection Strategies

Specific description: Git

In the cases of git, we define ”code change” and ”date of a change” as we detail in Code Changes. The date of a change can be defined (for considering it in a period or not) as the author date or the committer date of the corresponding git commit.

Since git provides changes as diff patches (list of lines added and removed), each of those lines mentioned as a line added or a line removed in the diff will be considered as a line changed (touched). If a line is removed and added, it will be considered as two ”changes to a line”.

Mandatory parameters:

- Kind of date. Either author date or committer date. Default: author date.
  For each git commit, two dates are kept: when the commit was authored, and when it was committed to the repository. For deciding on the period, one of them has to be selected.
- Include merge commits. Boolean. Default: True.
  Merge commits are those which merge a branch, and in some cases are not considered as reflecting a coding activity

References

- https://www.odoo.com/documentation/13.0/reference/guidelines.html#tag-and-module-name

Code Changes

Question: How many changes were made to the source code during a specified period?

Description

These are changes to the source code during a certain period. For ”change” we consider what developers consider an atomic change to their code. In other words, a change is some change to the source code which usually is accepted and merged as a whole, and if needed, reverted as a whole too. For example, in the case of git, each ”change” corresponds to a ”commit”, or, to be more precise, ”code change” corresponds to the part of a commit which touches files considered as source code.

Objectives

- Volume of coding activity. Code changes are a proxy for the activity in a project. By counting the code changes in the set of repositories corresponding to a project, you can have an idea of the overall coding activity in that project. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Count. Total number of changes during the period.

Parameters:

- Period of time. Start and finish date of the period. Default: forever. Period during which changes are considered.
- Criteria for source code. Algorithm. Default: all files are source code. If focused on source code, criteria for deciding whether a file is a part of the source code or not.

Filters

- By actors (author, committer). Requires actor merging (merging ids corresponding to the same author).
- By groups of actors (employer, gender...). Requires actor grouping, and likely, actor merging.
- By tags (used in the message of the commits). Requires a structure for the message of commits. This tag can be used in an open-source project to communicate to every contributors if the commit is, for example, a fix for a bug or an improvement of a feature.

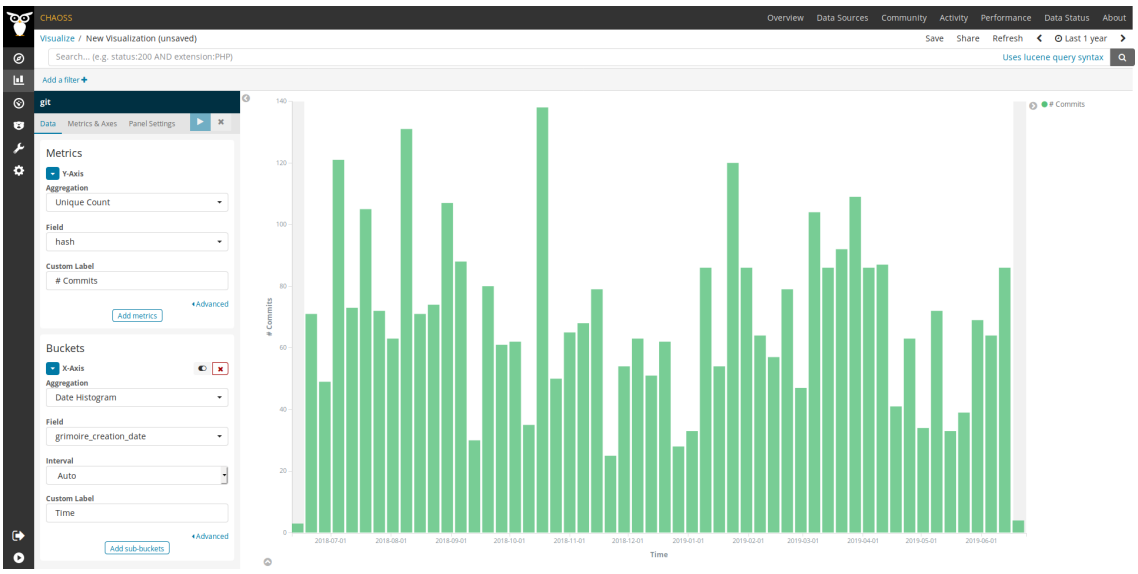
Visualizations

- Count per month over time
- Count per group over time

These could be represented as bar charts, with time running in the X axis. Each bar would represent a code changes during a certain period (eg, a month).

Tools Providing the Metric

- GrimoireLab provides this metric out of the box.
  - View an example on the CHAOSS instance of Bitergia Analytics.
  - Download and import a ready-to-go dashboard containing examples for this metric visualization from the GrimoireLab Sigils panel collection.
  - Add a sample visualization to any GrimoreLab Kibiter dashboard following these instructions:
    - Create a new `Vertical Bar` chart
    - Select the `git` index
    - Y-axis: `Unique Count` Aggregation, `hash` Field, `# Commits` Custom Label
    - X-axis: `Date Histogram` Aggregation, `grimoire_creation_date` Field, `Auto` Interval, `Time` Custom Label
  - Example screenshot:



- Augur provides this metric both as Code Changes and as Code Changes Lines. Both metrics are available in both the `repo` and the `repo_group` metric forms - more on that in the Augur documentation.
- Gitdm

Data Collection Strategies

Specific description: Git

See reference implementation for git

Mandatory parameters (for Git):

- Date type. Either author date or committer date. Default: author date.
  For each git commit, two dates are kept: when the commit was authored, and when it was committed to the repository. For deciding on the period, one of them has to be selected.

- Include merge commits. Boolean. Default: True.
  Merge commits are those which merge a branch, and in some cases are not considered as reflecting a coding activity.
- Include empty commits. Boolean. Default: True.
  Empty commits are those which do not touch files, and in some cases are not considered as reflecting a coding activity.

References

- https://www.odoo.com/documentation/13.0/reference/guidelines.html#tag-and-module-name

Focus Area - Code-Development-Efficiency

Goal: Learning how efficiently activities around code development get resolved.

Metric	Question
Change Request Acceptance Ratio	What is the ratio of change requests accepted to change requests closed without being merged?
Change Requests Accepted	How many accepted change requests are present in a code change?
Change Requests Declined	What change requestsended up being declined during a certain period?
Change Requests Duration	What is the duration of time between the moment a change request starts and the moment it is accepted or closed?

Change Request Acceptance Ratio

Question: What is the ratio of change requests accepted to change requests closed without being merged?

Description

Each change request can be in one of three states: open, merged (accepted), and closed without merge (declined). This metric measures the ratio of change requests merged (accepted) vs change requests closed without being merged.

Objectives

The ratio of change requests merged to change requests closed without merging provides insight into several repository characteristics, including openness to outside contributions, growth of the contributor community, the efficiency of code review processes, and, when measured over time, the trajectory of a project in its evolution. Different ratios should be interpreted in the context of each repository or project.

Implementation

Parameters Time Period Granularity (Weekly, Monthly, Annually). Change in ratio over the period of time. Show contributor count Origin of change request: branch or fork? Change requests from repository forks are more commonly from outside contributors, while branch originating change requests come from people with repository commit rights.

Aggregators Total change requests merged (accepted) Total change requests closed without merge Total change requests in an open state

Visualizations (optional)

CHAOSS tools provide a number of visualizations for this metric. The first visualization shows the accepted and declined change requests organized annually, from which ratios can be derived.

Figure One:

Closed Pull Request Volume

Consistent Increase In Total Volume And Great Ratio Of Merged / Rejected Pull Requests

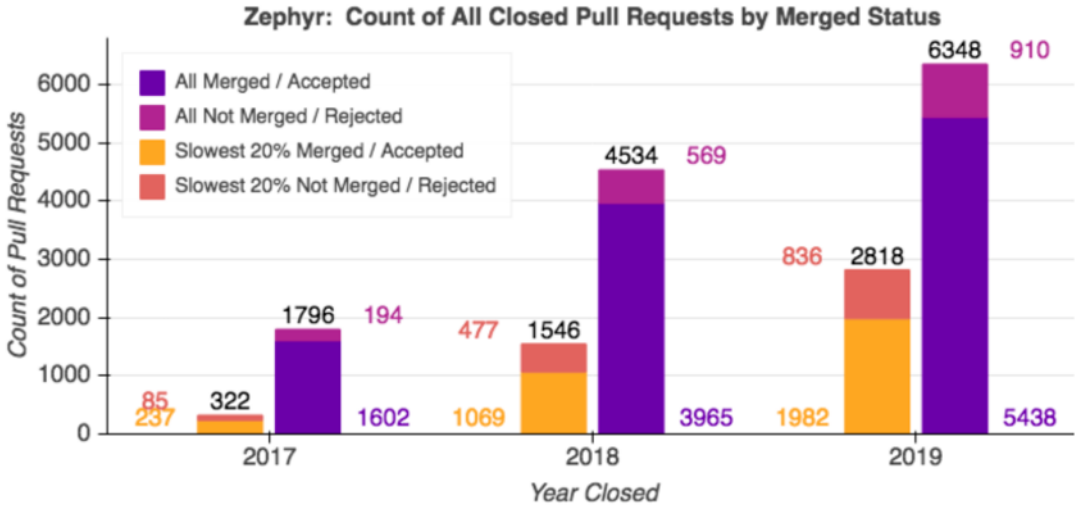


Figure Two:

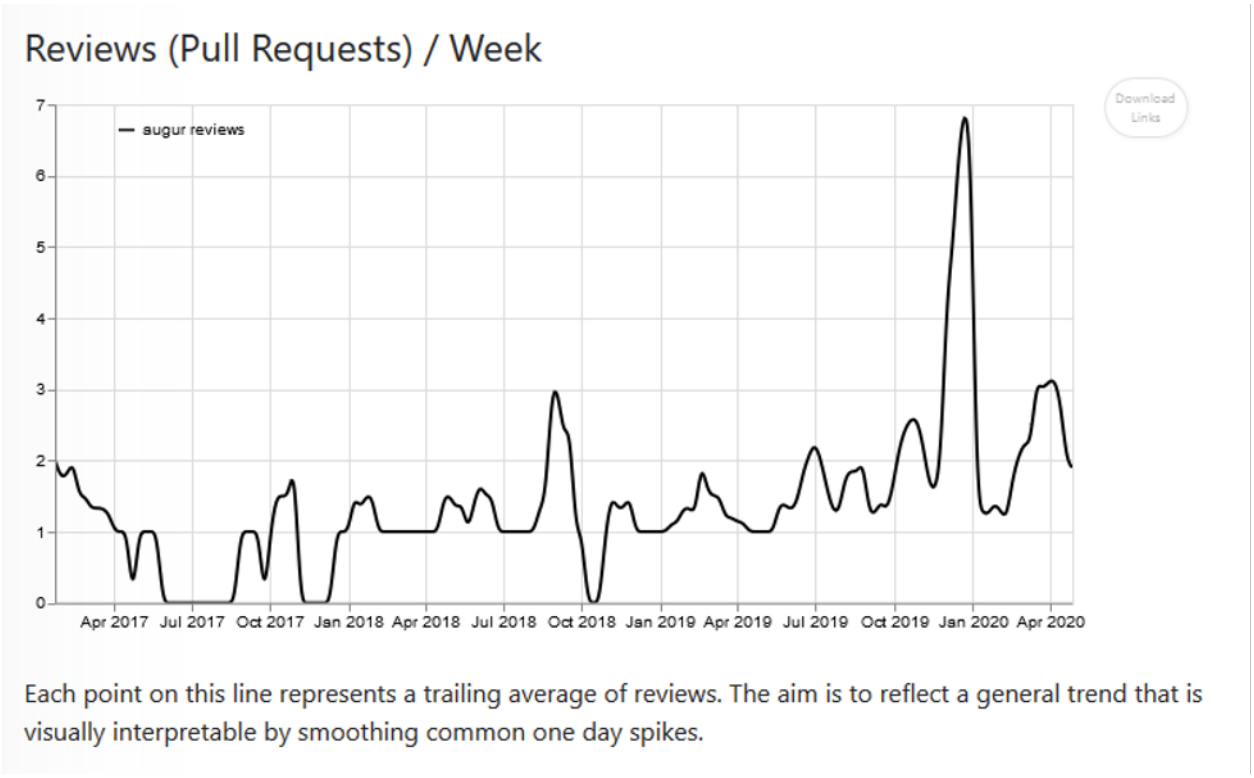
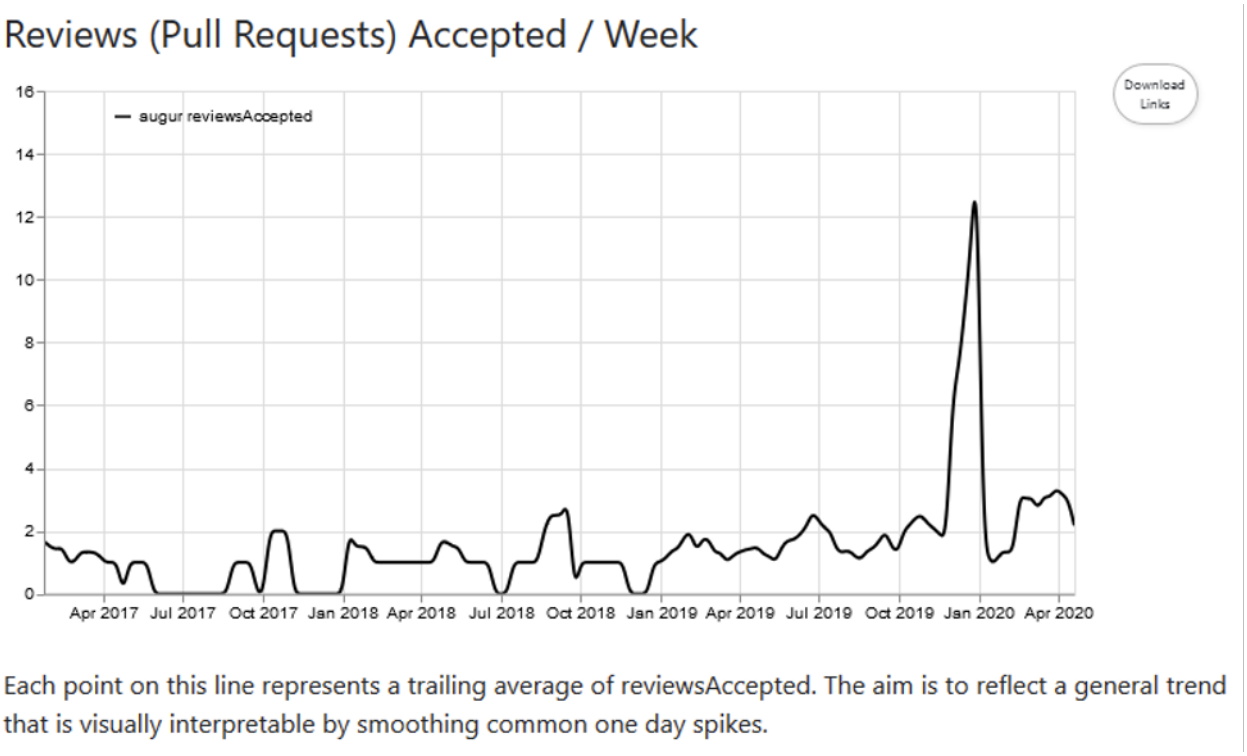


Figure Three:



Tools Providing the Metric (optional)

https://github.com/chaoss/augur https://github.com/chaoss/augur-community-reports

Data Collection Strategies

Accepted change requests are defined as in the Change Requests Accepted metric, and Declined change requests are defined as in the Change Requests Declined metric.

References

Augur Zephyr report on pull requests: https://docs.google.com/presentation/d/11b48Zm5Fwsmd1OIHg4bse5ibaVJUWkUIZbVqxTZeStg/edit#slide=id.g7ec7768776_1_56

Change Requests Accepted

Question: How many accepted change requests are present in a code change?

Description

Change requests are defined as in Change Requests. Accepted change requests are those that end with the corresponding changes finally merged into the code base of the project. Accepted change requests can be linked to one or more changes to the source code, those corresponding to the changes proposed and finally merged.

For example, in GitHub when a pull request is accepted, all the commits included in it are merged (maybe squashed, maybe rebased) in the corresponding git repository. The same can be said of GitLab merge requests. In the case of Gerrit, a change request usually corresponds to a single commit.

Objectives

- Volume of coding activity.
  Accepted change requests are a proxy for the activity in a project. By counting accepted change requests in the set of repositories corresponding to a project, you can have an idea of the overall coding activity in that project that leads to actual changes. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Count. Total number of accepted change requests during the period.
- Ratio. Ratio of accepted change requests over total number of change requests during that period.

Parameters:

- Period of time. Start and finish date of the period during which accepted change requests are considered. Default: forever.
- Criteria for source code. Algorithm. Default: all files are source code.
  If we focus on source code, we need a criterion for deciding whether a file belongs to the source code or not.

Filters

- By actor type (submitter, reviewer, merger). Requires merging identities corresponding to the same actor.
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.

Visualizations

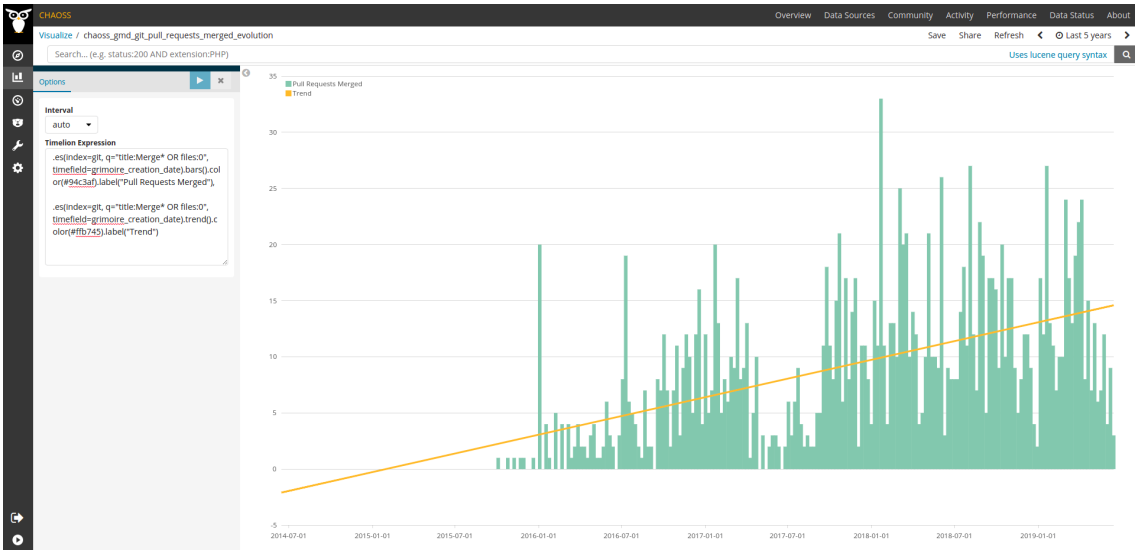
- Count per time period over time

- Ratio per time period over time

These could be grouped by actor type or actor group by applying the filters defined above. These could be represented as bar charts, with time running in the X axis. Each bar would represent accepted change requests to change the code during a certain period (eg, a month).

Tools Providing the Metric

- Grimoirelab provides this metric out of the box for GitHub Pull Requests and also provides data to build similar visualizations for GitLab Merge Requests and Gerrit Changesets.
  - View an example on the CHAOSS instance of Bitergia Analytics.
  - Download and import a ready-to-go dashboard containing examples for this metric visualization based on GitHub Pull Requests data from the GrimoireLab Sigils panel collection.
  - Example screenshot:



Data Collection Strategies

Specific description: GitHub

In the case of GitHub, accepted change requests are defined as ”pull requests whose changes are included in the git repository”, as long as it proposes changes to source code files.

Unfortunately, there are several ways of accepting change requests, not all of them making it easy to identify that they were accepted. The easiest situation is when the change request is accepted and merged (or rebased, or squashed and merged). In that case, the change request can easily be identified as accepted, and the corresponding commits can be found via queries to the GitHub API.

But change requests can also be closed, and commits merged manually in the git repository. In this case, commits may still be found in the git repository, since their hash is the same found in the GitHub API for those in the change request.

In a more difficult scenario, change requests can also be closed, and commits rebased, or maybe squashed and then merged, manually. In these cases, hashes are different, and only an approximate matching via dates and authors, and/or comparison of diffs, can be used to track commits in the git repository.

From the point of view of knowing if change requests were accepted, the problem is that if they are included in the git repository manually, the only way of knowing that the change request was accepted is finding the corresponding commits in the git repository.

In some cases, projects have policies of mentioning the commits when the change request is closed (such as ”closing by accepting commits xxx and yyyy”), which may help to track commits in the git repository.

Mandatory parameters (for GitHub):

- Heuristic for detecting accepted change requests not accepted via the web interface. Default: None.

Specific description: GitLab

In the case of GitLab, accepted change requests are defined as ”merge requests whose changes are included in the git repository”, as long as it proposes changes to source code files.

Mandatory parameters (for GitLab):

- Heuristic for detecting accepted change requests not accepted via the web interface. Default: None.

Specific description: Gerrit

In the case of Gerrit, accepted change requests are defined as ”changesets whose changes are included in the git repository”, as long as they proposes changes to source code files.

Mandatory parameters (for Gerrit): None.

References

Change Requests Declined

Question: What change requestsended up being declined during a certain period?

Description

Change requests are defined as in Change Requests. Declined change requests are those that are finally closed without being merged into the code base of the project.

For example, in GitHub when a pull request is closed without merging, and the commits referenced in it cannot be found in the git repository, it can be considered to be declined (but see detailed discussion below). The same can be said of GitLab merge requests. In the case of Gerrit, code reviews can be formally ”abandoned”, which is the way of detecting declined change requests in this system.

Objectives

- Volume of coding activity. Declined change requests are a proxy for the activity in a project. By counting declined change requests in the set of repositories corresponding to a project, you can have an idea of the overall coding activity in that project that did not lead to actual changes. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Count. Total number of declined change requests during the period.
- Ratio. Ratio of declined change requests over the total number of change requests during that period.

Parameters:

- Period of time. Start and finish date of the period during which declined change requests are considered. Default: forever.
- Criteria for source code. Algorithm. Default: all files are source code. If we focus on source code, we need a criterion to decide whether a file belongs to the source code or not.

Filters

- By actors (submitter, reviewer, merger). Requires merging identities corresponding to the same actor.
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.

Visualizations

- Count per period over time
- Ratio per period over time

These could be grouped (per actor type, or per group of actors) by applying the filters, and could be represented as bar charts, with time running in the X axis. Each bar would represent declined change requests during a certain period (e.g., a month).

Data Collection Strategies

Specific description: GitHub

In the case of GitHub, declined change requests are defined as ”pull requests that are closed with their changes not being included in the git repository”, as long as it proposes changes to source code files.

See the discussion in the specific description for GitHub in Change Requests Accepted, since it applies here as well.

Mandatory parameters (for GitHub):

- Heuristic for detecting declined change requests, telling apart those cases where the change request was closed, but the changes were included in the git repository manually. Default: None.

Specific description: GitLab

In the case of GitLab, declined reviews are defined as ”merge requests that are closed with their changes not being included in the git repository”, as long as it proposes changes to source code files.

Mandatory parameters (for GitLab):

- Heuristic for detecting declined change requests, telling apart those cases where the merge request was closed, but the changes were included in the git repository manually. Default: None.

Specific description: Gerrit

In the case of Gerrit, declined change requests are defined as ”changesets abandoned”, as long as they propose changes to source code files.

Mandatory parameters (for Gerrit): None.

References

Change Requests Duration

Question: What is the duration of time between the moment a change request starts and the moment it is accepted or closed?

Description

Change requests are defined as in Change Requests. Accepted change requests are defined in Change Requests Accepted.

The change request duration is the duration of the period since the change request started, to the moment it ended (by being accepted and being merged in the code base). This only applies to accepted change requests.

For example, in GitLab a change request starts when a developer uploads a proposal for a change in code, opening a change request. It finishes when the change request is finally accepted and merged in the code base, closing the change request.

In case there are comments or other events after the code is merged, they are not considered for measuring the duration of the change request.

Objectives

- Duration of acceptance of change requests. Review duration for accepted change requests is one of the indicators showing how long does a project take before accepting a contribution of code. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Median. Median (50% percentile) of change request duration for all accepted change requests in the considered period of time.

Parameters:

- Period of time. Start and finish date of the period. Default: forever.
  Period during which accepted change requests are considered. An accepted change request is considered to be in the period if its creation event is in the period.
- Criteria for source code. Algorithm. Default: all files are source code.
  If we are focused on source code, we need a criteria for deciding whether a file is a part of the source code or not.

Filters

- By actors (submitter, reviewer, merger). Requires actor merging (merging ids corresponding to the same author).
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.

Visualizations

- Median per month over time
- Median per group over time

These could be represented as bar charts, with time running in the X axis. Each bar would represent accepted change requests to change the code during a certain period (e.g., a month).

- Distribution of durations for a certain period

These could be represented with the usual statistical distribution curves, or with bar charts, with buckets for duration in the X axis, and number of reviews in the Y axis.

Data Collection Strategies

Specific description: GitHub

In the case of GitHub, duration is considered for pull requests that are accepted and merged in the code base. For an individual pull request, duration starts when it is opened, and finishes when the commits it includes are merged into the code base.

Mandatory parameters (for GitHub): None.

Specific description: GitLab

In the case of GitLab, duration is considered for merge requests that are accepted and merged in the code base. For an individual merge request, duration starts when it is opened, and finishes when the commits it includes are merged into the code base.

Mandatory parameters (for GitLab): None.

Specific description: Gerrit

In the case of Gerrit, duration is considered for code reviews that are accepted and merged in the code base. For an individual code review, duration starts when it is opened, and finishes when the commits it includes are merged into the code base.

Mandatory parameters (for Gerrit): None.

References

Focus Area - Code-Development-Process-Quality

Goal: Learning about the processes to improve/review quality that are used (for example: testing, code review, tagging issues, tagging a release, time to response, CII Badging).

Metric	Question
Change Requests	What new change requests to the source code occurred during a certain period?

Change Requests

Question: What new change requests to the source code occurred during a certain period?

Description

When a project uses change request processes, changes are not directly submitted to the code base, but are first proposed for discussion as ”proposals for change to the source code”. Each of these change requests are intended to be reviewed by other developers, who may suggest improvements that will lead to the original proposers sending new versions of their change requests, until reviews are positive, and the code is accepted, or until it is decided that the proposal is declined.

For example, ”change requests” correspond to ”pull requests” in the case of GitHub, to ”merge requests” in the case of GitLab, and to ”code reviews” or in some contexts ”changesets” in the case of Gerrit.

Objectives

- Volume of changes proposed to a project. Change requests are a proxy for the activity in a project. By counting change requests in the set of repositories corresponding to a project, you can have an idea of the overall activity in changes to that project. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Count. Total number of change requests during the period.

Parameters:

- Period of time. Start and finish date of the period. Default: forever.
  Period during which change requests are considered.
- Criteria for source code. Algorithm. Default: all files are source code.
  If we are focused on source code, we need a criteria for deciding whether a file is a part of the source code or not.

Filters

- By actors (submitter, reviewer, merger). Requires actor merging (merging ids corresponding to the same author).
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.
- Status (open, closed)

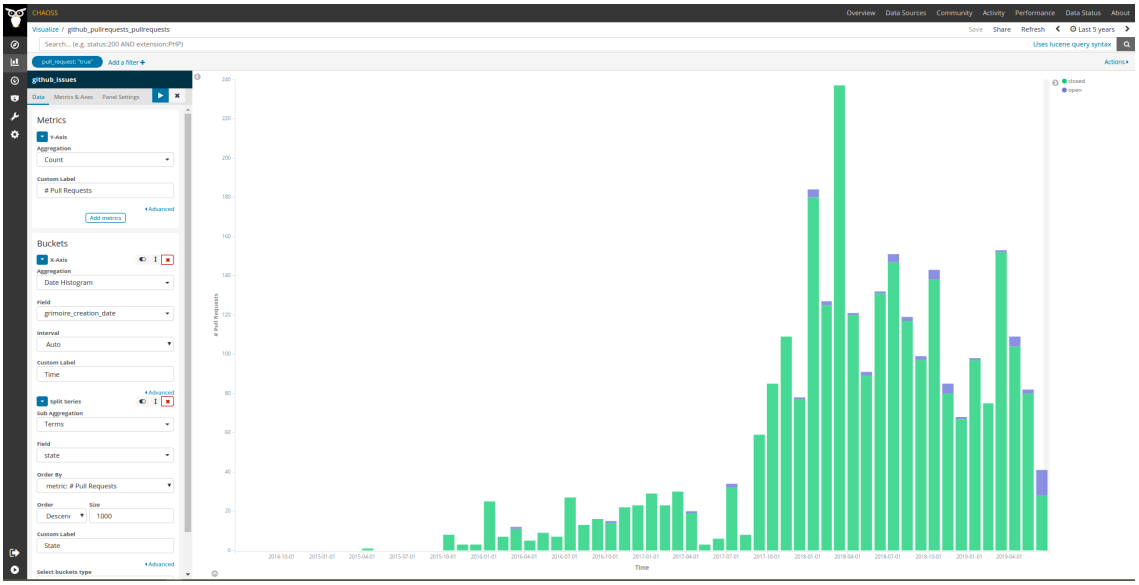
Visualizations

- Count per month over time
- Count per group over time

These could be represented as bar charts, with time running in the X axis. Each bar would represent change requests to change the code during a certain period (eg, a month).

Tools Providing the Metric

- Grimoirelab provides this metric out of the box for GitHub Pull Requests, GitLab Merge Requests, and Gerrit Changesets.
  - View an example on the CHAOSS instance of Bitergia Analytics.
  - Download and import a ready-to-go dashboard containing examples for this metric visualization based on GitHub Pull Requests data from the GrimoireLab Sigils panel collection.
  - Example screenshot:



Data Collection Strategies

Specific description: GitHub

In the case of GitHub, a change request is defined as a ”pull request”, as long as it proposes changes to source code files.

The date of the review can be defined (for considering it in a period or not) as the date in which the pull request was submitted.

Specific description: GitLab

In the case of GitLab, a change request is defined as a ”merge request”, as long as it proposes changes to source code files.

The date of the change request can be defined (for considering it in a period or not) as the date in which the change request was submitted.

Specific description: Gerrit

In the case of Gerrit, a change request is defined as a ”code review”, or in some contexts, a ”changeset”, as long as it proposes changes to source code files.

The date of the change request can be defined (for considering it in a period or not) as the date in which the change request was started by submitting a patchset for review.

References

Focus Area - Community-Growth

Goal: Identify the size of the project community and whether it is growing, shrinking, or staying the same.

Metric	Question
This metric is a release candidate. To comment on this metric please see Issue [	Contribution Attribution
Inactive Contributors	How many Contributors have gone inactive over a specific period of time?
New Contributors Closing Issues	How many contributors are closing issues for the first time in a given project?
New Contributors	How many contributors are making their first contribution to a given project and who are they?

This metric is a release candidate. To comment on this metric please see Issue #410. Following a comment period, this metric will be included in the next regular release.

Contribution Attribution

Question: Who has contributed to an open source project and what attribution information about people and organizations is assigned for a contribution?

Description

This metric evaluates who has worked on the project and specific project tasks and provides the attribution to project contributors and affiliated organizations. The aim is to understand, through insights into the paid contribution dynamics of a community, "how the work gets done."

Objectives

1. Who is working on the project?
2. What is the ratio of volunteer work, sponsored work, and blended work?
3. How many contributions are sponsored?
4. Who is sponsoring the contributions?
5. What types of contributions are sponsored?
6. How diverse is the community of contributors working on a project?

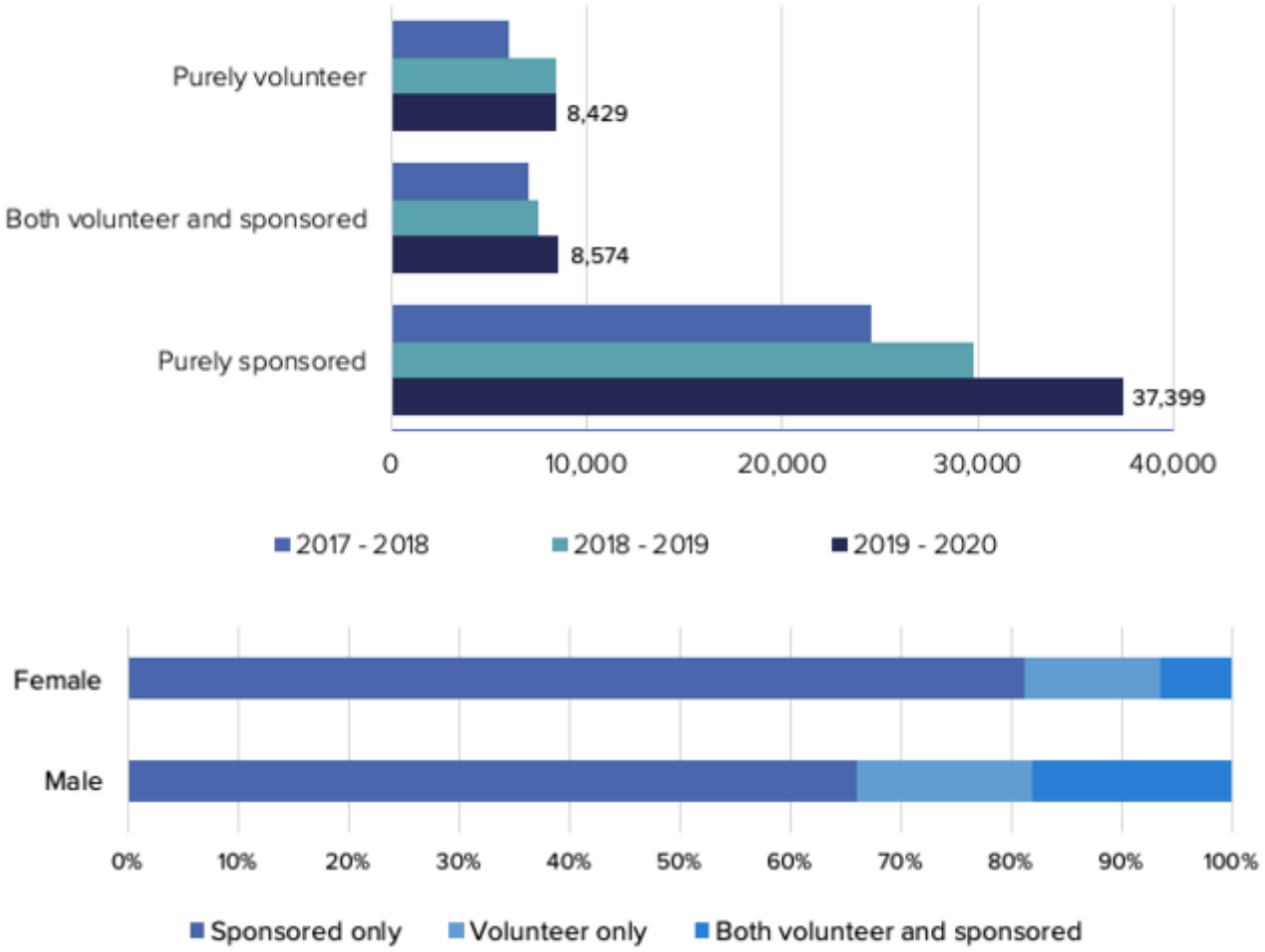
Implementation

Most contributions can be implicitly attributed using trace data, and these attributions are reflected in other metrics. However, this metric relies heavily on data that is volunteered by contributors and interpreted by project leadership. The implementation of this metric demands that the human in the loop determine what organizations, and what individual contributors a contribution is attributed to. Each individual contributor should be offered the opportunity to indicate what firm, foundation, project, and/or client paid for a particular change.

Filters

- Type of Contributor (individual, organization, gender, race, global status, work location)
  - Volunteer
  - Sponsored by a firm and/or client
  - Role that a contributor plays on a project (i.e., maintainer, board member, etc.)
- Type of Contribution
  - Links to contribution artifacts, like merge requests, issues, and the like, where relevant.
  - Indication of contribution types not managed in Git platforms like GitHub, GitLab, and BitBucket.
  - See also: https://chaoss.community/metric-types-of-contributions/
- Volunteer versus sponsored - Related to Organizational DiversityandLabor Investment

Visualizations



Tools Providing the Metric

1. The Drupal community built this tool, began using it in 2015, and has been reporting their results annually
2. There is an issue open with GitLab to implement this functionality

Data Collection Strategies

The Drupal Community implemented an example of how to gather information necessary for this metric to be calculated. It associates individuals, and organizations those individuals indicate as warranting attribution, for each discrete contribution.

Data Ethics Considerations

Although this metric requires the capture of a relationship between individuals and the contributions they make, the intent of this metric is NOT to measure individuals. The aim is to enable a wider understanding of how contributions to this project are motivated. Explicitly, it is not the intent of this metric to contribute to gamification of individual contributor work.

References

- https://dri.es/a-method-for-giving-credit-to-organizations-that-contribute-code-to-open-source
- https://www.drupal.org/blog/who-sponsors-drupal-development
- https://dri.es/who-sponsors-drupal-development-2020
- https://gitlab.com/gitlab-org/gitlab/-/issues/327138

Contributors

- Matthew Tift
- Sean Goggins
- Elizabeth Barron
- Vinod Ahuja
- Armstrong Foundjem
- Kevin Lumbard

Inactive Contributors

Question: How many Contributors have gone inactive over a specific period of time?

Description

A metric that shows how many contributors have stopped contributing over a specific period of time. The period of time required for a contributor to be counted as inactive can be decided by a variable and this metric will display the number of contributors that have been labeled as inactive over a given time frame.

Objectives

The objective is to determine how many people have stopped contributing actively. This could be useful for community managers to determine if key members are losing interest, or are taking a break.

Implementation

The metric will take in two variables - a time period and a interval. The time period will be the period over which the number of inactive members will be displayed. For example if time period=year then it will display the number of contributors that have gone inactive each year. The interval will determine how long it takes for a contributor to be labeled as inactive. If a contributor has not made a contribution for a length of time longer than the interval, they will be counted as inactive.

The metric will work by:

1. getting a list of all contributors
2. checking the last contribution date
3. if the last contribution date is before the cutoff then add them to the inactivity count of the period they last contributed in.
4. create list of inactive contributors

Aggregators:

- inactive: number of inactive contributors

Filters

- Minimum contributions required to be considered active
- Period of time to determine inactivity
- Start date/End date
- Period of graph

Data Collection Strategies

The list of contributors can be collected using the existing contributors metric. To determine the last contribution date new code may be needed.

New Contributors Closing Issues

Question: How many contributors are closing issues for the first time in a given project?

Description

This metric is an indication of the volume of contributors who are closing issues for their first time within a given project. When a contributor closes an issue for the first time it is some indication of ”stickiness” of the individual within that project, especially for contributors who are not also committers.

Objectives

To know how contributors are moving through the contributor funnel by identifying “closing issues” as a milestone in the contributor journey.

Implementation

Aggregators:

- Count. Total number of contributors closing issues on this project for the first time during a given time period.
- Percentage. Proportion of contributors closing issues on this project *for the first time* during a given time period, computed against *all* contributors having closed issues on this project during the same time period.

Parameters:

- Period of time. Start and finish date of the period during which new issue closers are counted. Default: forever (i.e., the entire observable project lifetime)

Filters

- Exclude reopened issues (optionally, filter if reopened in less than 1 hour)

Visualizations

- Table with names of contributors who closed an issue for the first time and when that was.
- Timeline showing the time on the x-axis, and the aggregated metric value on the y-axis for fixed consecutive periods of time (e.g. on a monthly basis). This visualisation allows to show how the metric is evolving over time for the considered project.

Data Collection Strategies

Based on the Issues Closed and Contributor definitions, enrich contributors with the date of their first time closing an issue.

References

- Contributor Funnel by Mike McQuaid

New Contributors

Question: How many contributors are making their first contribution to a given project and who are they?

Description

An increase or decline in new contributors can be an early indicator of project health. Understanding the behavior and barriers of new community members requires knowing who they are -- helping to welcome new contributors and thank them for their efforts.

This is a specific implementation of the Contributors metric.

Objectives

An increase or decline in new contributors can be an early indicator of project health. Understanding the behavior and barriers of new community members requires knowing who they are. Welcome new contributors and thank them for their first contribution.

Implementation

For each Contributor, only consider their first contribution.

Filters

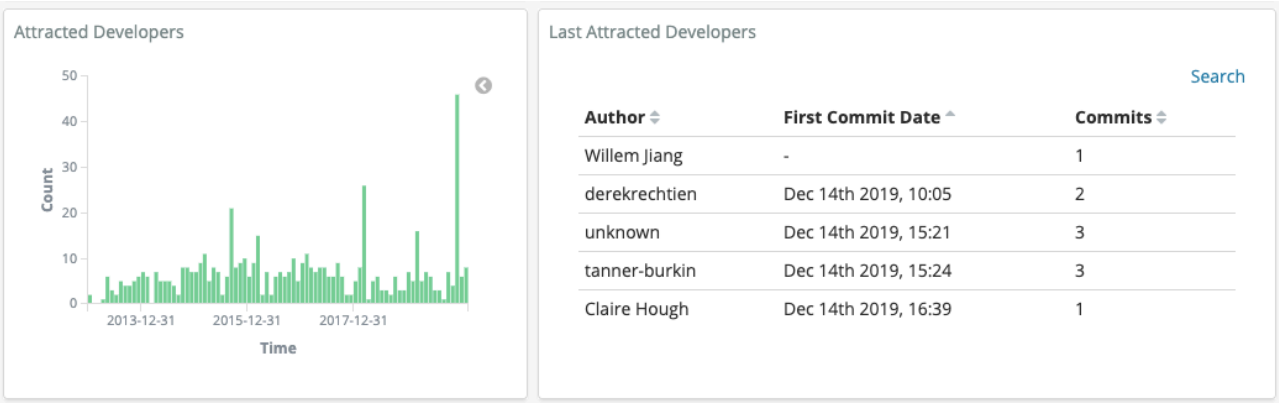
Period of Time: When was the first contribution made.

By location of engagement. For example:

- Repository authors
- Issue authors
- Code review participants
- Mailing list authors
- Event participants
- IRC authors
- Blog authors
- By release cycle
- Timeframe of activity in the project, e.g, find new contributors
- Programming languages of the project
- Role or function in project

Stage of engagement (e.g., opening pull request vs. getting it accepted).

Visualizations



Tools Providing the Metric

Augur GrimoireLab

References

https://opensource.com/article/17/4/encourage-new-contributors

Focus Area - Issue-Resolution

Goal: Identify how effective the community is at addressing issues identified by community participants.

Metric	Question
Issue Age	How long have open issues been left open?
Issue Resolution Duration	How long does it take for an issue to be closed?
Issue Response Time	How much time passes between the opening of an issue and a response in the issue thread from another contributor?
Issues Active	How many issues were active during a certain period?
Issues Closed	How many issues were closed during a certain period?
New Issues	How many new issues are created during a certain period?

Issue Age

Question: How long have open issues been left open?

Description

This metric is an indication of how long issues have been left open in the considered time period. If an issue has been closed but re-opened again within that period if will be considered as having remained open since its initial opening date.

Objectives

When the issue age is increasing, identify the oldest open issues in a project to gain insight as to why they have been open for an extended period of time. Additionally, to understand how well maintainers are resolving issues and how quickly issues are resolved.

Implementation

For all open issues, get the date the issue was opened and calculate the number of days to current date.

Aggregators:

- Average. Average age of all open issues.
- Median. Median age of all open issues.

Parameters:

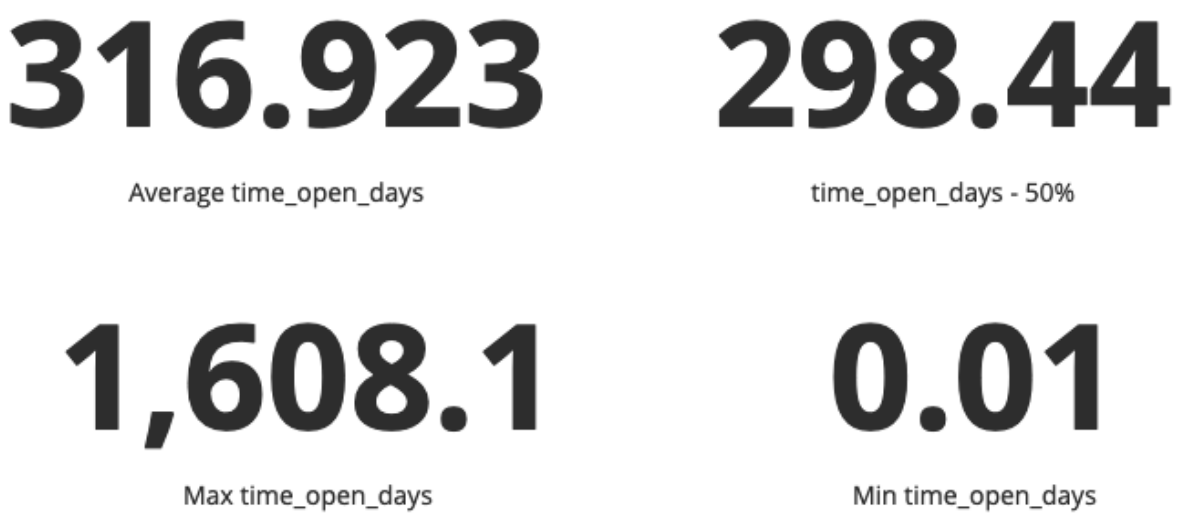
- Period of time. Start and finish date of the period during which open issues are considered. Default: forever (i.e., the entire observable period of the project's issue activity).

Filters

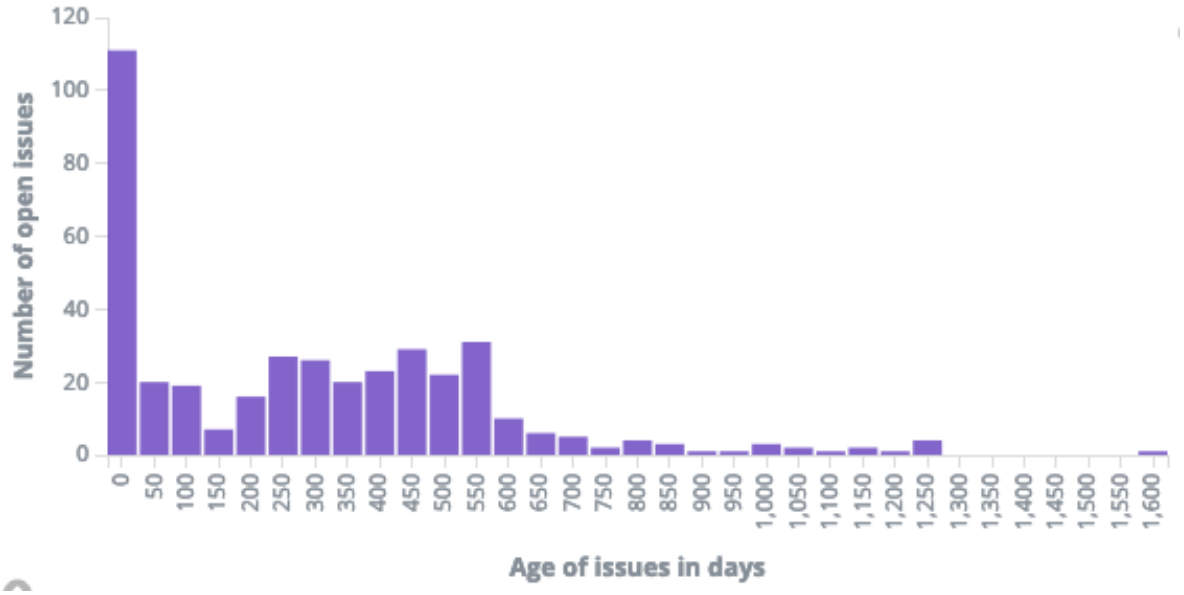
- Module or working group
- Tags/labels on issue

Visualizations

1. Summary data for open issues



2. Count of open issues per day



Tools Providing the Metric

- GrimoireLab
- Augur

Data Collection Strategies

For specific descriptions of collecting data about closed issues, please refer to the corresponding section of Issues New.

References

Issue Resolution Duration

Question: How long does it take for an issue to be closed?

Description

This metric is an indication of how long an issue remains open, on average, before it is closed. Issues are defined as in Issues Closed.

For issues that were reopened and closed again, only the last close date is relevant for this metric.

Objectives

This metric can be used to evaluate the effort and time needed to conclude and resolve discussions. This metric can also provide insights to the level of responsiveness in a project.

Implementation

For each closed issue:

- Issue Resolution Duration = Timestamp of issue closed - Timestamp of issue opened

Aggregators:

- Average. Average amount of time (in days, by default) for an issue in the repository to be closed.

Parameters:

- Period of time. Start and finish date of the period. Default: forever.
  Period during which issues are considered.

Filters

- By time. Provides average issue resolution duration time starting from the provided beginning date to the provided end date.
  - By open time. Provides information for how long issues created from the provided beginning date to the provided end date took to be resolved.
  - By closed time. Provides information for how long old issues were that were closed from the provided beginning date to the provided end date took to be resolved.
- By actors (submitter, commenter, closer). Requires actor merging (merging ids corresponding to the same author).
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.

Visualizations

- Average over time (e.g. average for January, average for February, average for March, etc.)

- Average for a given time period (e.g. average for all of 2019, or average for January to March)

Tools Providing the Metric

- Augur provides this metric as Closed Issue Resolution Duration. This metrics are available in both the `repo` and the `repo_group` metric forms - more on that in the Augur documentation.

Data Collection Strategies

For specific descriptions of collecting data about closed issues, please refer to the corresponding section of Issues Closed.

References

Issue Response Time

Question: How much time passes between the opening of an issue and a response in the issue thread from another contributor?

Description

This metric is an indication of how much time passes between the opening of an issues and a response from other contributors.

This metric is a specific case of the Time to First Reponse metric in the Common working group.

Objectives

Learn about the responsiveness of an open source community.

Implementation

Aggregators:

- Average. Average response time in days.
- Median. Median response time in days.

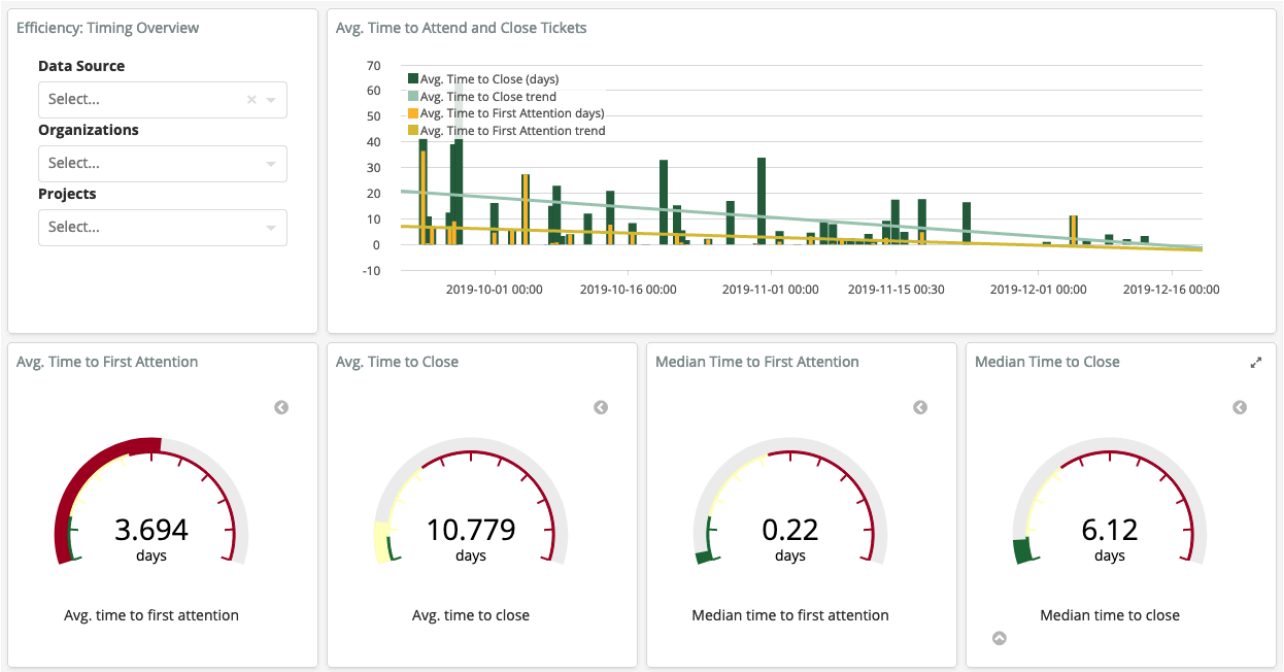
Parameters:

- Period of time. Start and finish date of the period. Default: forever.
- Period during which responses are counted.

Filters

- response from role in project (e.g., first maintainer response)
- bot versus human (e.g., filter out automated “welcome first contributor messages”)
- opened by role in project (e.g., responsiveness to new contributors versus long-timers)
- date issue was opened
- status of issue (e.g., only look at currently open issues)

Visualizations



Tools Providing the Metric

- GrimoireLab: General for any ticketing system, GitHub Issues, GitLab Issues
- Augur

Data Collection Strategies

Look at the Issues New metric for a definiton of “issues.” Subtract the issue opened timestamp from the first response timestamp. Do not count responses if created by the author of the issue.

References

Issues Active

Question: How many issues were active during a certain period?

Description

Issues are defined as in Issues New. Issues showing some activity are those that had some comment, or some change in state (including closing the issue), during a certain period.

For example, in GitHub Issues, a comment, a new tag, or the action of closing an issue, is considered as a sign of activity.

Objectives

- Volume of active issues in a project. Active issues are a proxy for the activity in a project. By counting active issues related to code in the set of repositories corresponding to a project, you can have an idea of the overall activity in working with issues in that project. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Count. Total number of active issues during the period.
- Ratio. Ratio of active issues over total number of issues during that period.

Parameters:

- Period of time. Start and finish date of the period during which issues are considered. Default: forever.
- Criteria for source code. Algorithm. Default: all issues are related to source code. If we focus on source code, we need a criterion for deciding whether an issue is related to the source code or not.

Filters

- By actor (submitter, commenter, closer). Requires merging identities corresponding to the same author.
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.

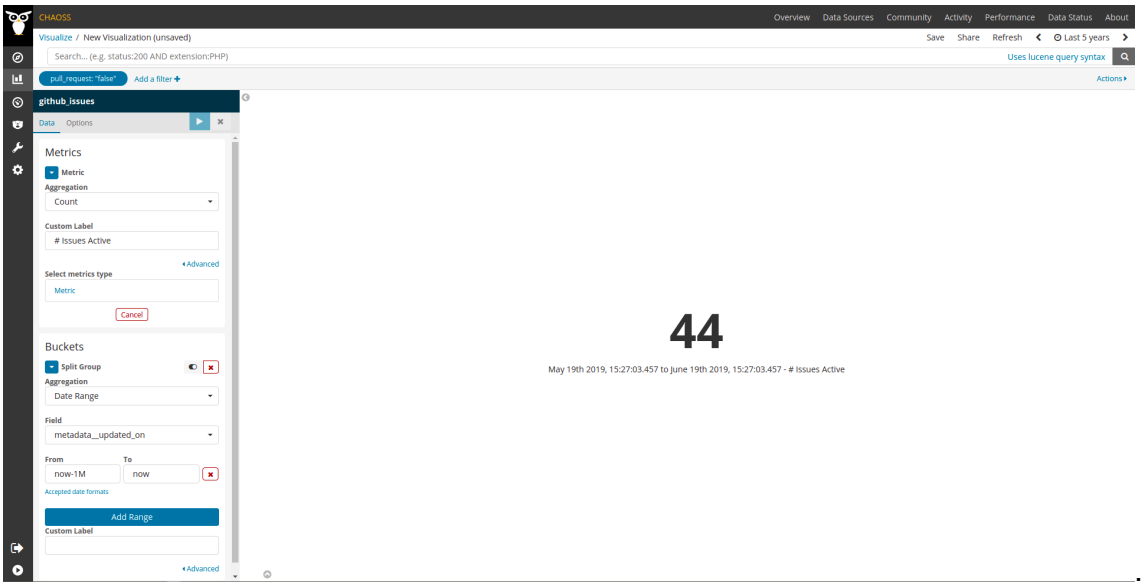
Visualizations

- Count per period over time
- Ratio per period over time

These could be grouped by applying the previously defined filters. These could be represented as bar charts, with time running in the X axis. Each bar would represent proposals to change the code during a certain period (eg, a month).

Tools Providing the Metric

- GrimoireLab provides data for computing a metric close to the one described in this page for GitHub Issues, GitLab issues, Jira, Bugzilla and Redmine. In terms of the metric, **GrimoireLab data have only the date of the last update of each item, which limits computing this metric to time ranges ending on the current date**.
  - Depending on the source API, the definition of what is considered an update on the issue could vary. GrimoireLab uses `metadata__updated_on` to store latest issue update, please check Perceval documentation to look for the specific API field being used in each case and understand its limitations, if any.
  - Currently, there is no dashboard showing this in action. Nevertheless, it is easy to build a visualization that shows the number uses which last activity occurred at some point between a date and current date (we'll do it for GitHub Issues here).
  - Add a sample visualization to any GrimoreLab Kibiter dashboard following these instructions:
    - Create a new `Metric` visualization.
    - Select the `github_issues` index.
    - Filter: `pull_request` is `false` .
    - Metric: `Count` Aggregation, `# Issues Active` Custom Label.
    - Buckets: `Date Range` Aggregation, `metadata__updated_on` Field, `now-1M` From (or whatever interval may fit your needs), `now` To, leave Custom Label empty to see the specific dates in the legend.
    - Have a look at the time picker on the top right corner to make sure it is set to include the whole story of the data so we are not excluding any item based on its creation date.
  - Example screenshot:



Data Collection Strategies

Specific description: GitHub

In the case of GitHub, active issues are defined as ”issues which get a comment, a change in tags, a change in assigned person, or are closed”.

Specific description: GitLab

In the case of GitLab, active issues are defined as ”issues which get a comment, a change in tags, a change in assigned person, or are closed”.

Specific description: Jira

In the case of Jira, active issues are defined as ”issues which get a comment, a change in state, a change in assigned person, or are closed”.

Specific description: Bugzilla

In the case of Bugzilla, active issues are defined as ”bug reports which get a comment, a change in state, a change in assigned person, or are closed”.

References

Issues Closed

Question: How many issues were closed during a certain period?

Description

Issues are defined as in Issues New. Issues closed are those that changed to state closed during a certain period.

In some cases or some projects, there are other states or tags that could be considered as ”closed”. For example, in some projects they use the state or the tag ”fixed” for stating that the issue is closed, even when it needs some action for formally closing it.

In most issue trackers, closed issues can be reopened after they are closed. Reopening an issue can be considered as opening a new issue, or making void the previous close (see parameters, below).

For example, in GitHub Issues or GitLab Issues, issues closed are issues that were closed during a certain period.

Objectives

Volume of issues that are dealt with in a project. Closed issues are a proxy for the activity in a project. By counting closed issues related to code in the set of repositories corresponding to a project, you can have an idea of the overall activity in finishing work with issues in that project. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Count. Total number of active issues during the period.
- Ratio. Ratio of active issues over total number of issues during that period.
- Reactions. Number of ”thumb-ups” or other reactions on issues.

Parameters:

- Period of time. Start and finish date of the period during which issues are considered. Default: forever.
- Criteria for source code. Algorithm. Default: all issues are related to source code. If we focus on source code, we need a criterion for deciding whether an issue is related to the source code or not. All issues could be included in the metric by altering the default.
- Reopen as new. Boolean, defining whether reopened issues are considered as new issues. If false, it means the closing event previous to a reopen event should be considered as void. Note: if this parameter is false, the number of closed issues for any period could change in the future, if some of them are reopened.
- Criteria for closed. Algorithm. Default: having a closing event during the period of interest.

Filters

- By actors (submitter, commenter, closer). Requires merging identities corresponding to the same author.
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.

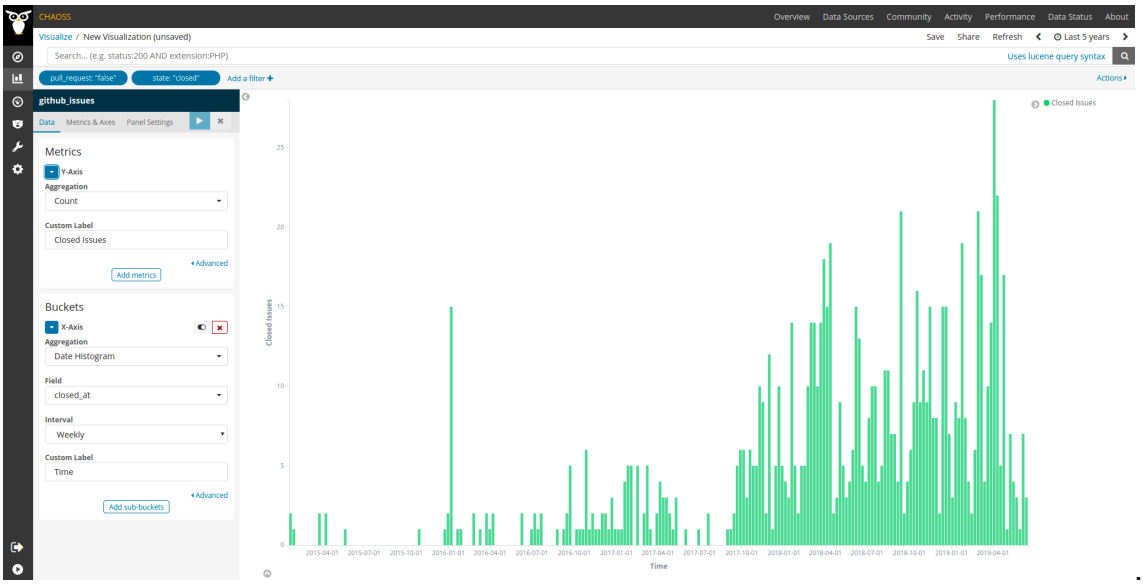
Visualizations

- Count per time period over time
- Ratio per time period over time

These could be grouped by applying the filters defined aboce. These could be represented as bar charts, with time running in the X axis.

Tools Providing the Metric

- GrimoireLab provides data for computing this metric for GitHub Issues, GitLab issues, Jira, Bugzilla and Redmine. Current dashboards show information based on creation date, that means they show current status of the issues that were created during a time period (e.g. GitHub Issues dashboard, you can see it in action). Nevertheless, it is easy to build a visualization that shows issues based on closing date by following the next steps:
  - Add a sample visualization to any GrimoreLab Kibiter dashboard following these instructions:
    - Create a new `Vertical Bar` chart.
    - Select the `github_issues` index.
    - Filter: `pull_request` is `false` .
    - Filter: `state` is `closed` .
    - Metrics Y-axis: `Count` Aggregation, `# Closed Issues` Custom Label.
    - Buckets X-axis: `Date Histogram` Aggregation, `closed_at` Field, `Weekly` Interval (or whatever interval may fit your needs, depending on the whole time range you wish to visualize in the chart), `Time` Custom Label.
  - Example screenshot:



Data Collection Strategies

Specific description: GitHub

In the case of GitHub, closed issues are defined as ”issues which are closed”.

Specific description: GitLab

In the case of GitLab, active issues are defined as ”issues that are closed”.

Specific description: Jira

In the case of Jira, active issues are defined as ”issues that change to the closed state”.

Specific description: Bugzilla

In the case of Bugzilla, active issues are defined as ”bug reports that change to the closed state”.

References

New Issues

Question: How many new issues are created during a certain period?

Description

Projects discuss how they are fixing bugs, or adding new features, in tickets in the issue tracking system. Each of these tickets (issues) are opened (submitted) by a certain person, and are later commented and annotated by many others.

Depending on the issue system considered, an issue can go through several states (for example, ”triaged”, ”working”, ”fixed”, ”won’t fix”), or being tagged with one or more tags, or be assigned to one or more persons. But in any issue tracking system, an issue is usually a collection of comments and state changes, maybe with other annotations. Issues can also be, in some systems, associated to milestones, branches, epics or stories. In some cases, some of these are also issues themselves.

At least two ”high level” states can usually be identified: open and closed. ”Open” usually means that the issues is not yet resolved, and ”closed” that the issue was already resolved, and no further work will be done with it. However, what can be used to identify an issue as ”open” or ”closed” is to some extent dependent on the issue tracking system, and on how a given project uses it. In real projects, filtering the issues that are directly related to source code is difficult, since the issue tracking system may be used for many kinds of information, from fixing bugs and discussing implementation of new features, to organizing a project event or to ask questions about how to use the results of the project.

In most issue trackers, issues can be reopened after being closed. Reopening an issue can be considered as opening a new issue (see parameters, below).

For example, ”issues” correspond to ”issues” in the case of GitHub, GitLab or Jira, to ”bug reports” in the case of Bugzilla, and to ”issues” or ”tickets” in other systems.

Objectives

Volume of issues discussed in a project. Issues are a proxy for the activity in a project. By counting issues discussing code in the set of repositories corresponding to a project, you can have an idea of the overall activity in discussing issues in that project. Of course, this metric is not the only one that should be used to track volume of coding activity.

Implementation

Aggregators:

- Count. Total number of new issues during the period.
- Ratio. Ratio of new issues over total number of issues during that period.

Parameters:

- Period of time. Start and finish date of the period during which issues are considered. Default: forever.
- Criterion for source code. Algorithm. Default: all issues are related to source code. If we focus on source code, we need a criterion for deciding whether an issue is related to the source code or not.

- Reopen as new. Boolean. Default: False.
  Criterion for defining whether reopened issues are considered as new issues.

Filters

- By actors (submitter, commenter, closer). Requires merging identities corresponding to the same author.
- By groups of actors (employer, gender... for each of the actors). Requires actor grouping, and likely, actor merging.

Visualizations

- Count per time period over time
- Ratio per time period over time

These could be grouped by applying the filters defined above. These could be represented as bar charts, with time running in the X axis. Each bar would represent proposals to change the code during a certain period (eg, a month).

Data Collection Strategies

Specific description: GitHub

In the case of GitHub, an issue is defined as an ”issue”.

The date of the issue can be defined (for considering it in a period or not) as the date in which the issue was opened (submitted).

Specific description: GitLab

In the case of GitHub, an issue is defined as an ”issue”.

The date of the issue can be defined (for considering it in a period or not) as the date in which the issue was opened (submitted).

Specific description: Jira

In the case of Jira, an issue is defined as an ”issue”.

The date of the issue can be defined (for considering it in a period or not) as the date in which the issue was opened (submitted).

Specific description: Bugzilla

In the case of Bugzilla, an issue is defined as a ”bug report”, as long as it is related to source code files.

The date of the issue can be defined (for considering it in a period or not) as the date in which the bug report was opened (submitted).

References