# Quantitative OSS project analysis

In Section 1.2.3 we have seen how the heterogenity of open source projects makes it hard to draw conclusions from the network statistics over time. We have seen some correlations between the releases and the metrics, but they weren't applicable to all repositories. Furthermore, the relatively small nubmer of repositories also prevented generalizing the findings, for example the metrics, which proved to be effective in pandas like clustering coefficient or degree centrality, might also apply to the majority of open-source repositories, but our project choices did not include any, that falls into this group. Therefore, in this section we will conduct a quantitative research on a large number of randomly selected repositories to rule out any bias we might have applied during the pattern analysis, where the main focus was to understand the collaboration network changes over time.

First, we query a large number of GitHub repositories. As we have seen in the previous chapter, the analysis works best on really large repositories, otherwise the activity tends to be zero between releases. Therefore, we filter for only the largest projects available on GitHub with specifying inclusion and exclusion criteria. As a next step, we randomly choose a number of projects from the result set, which we mine using git2net and repo\\_tools. Then we run a linear regression on the number of issues, the releases and the network statistics before and after a release to discover the correlations between these variables. Lastly, we discuss our findings and point to future research topics.

## Inclusion and exclusion criteria

We use the GitHub Search tool maintained by SEART to query the Github repositories. The tool allows us to set a detailed search criteria to find repositories. We use the following settings for querying:

* Commits: minimum 5000
* Contributors: minimum 5
* Issues: minimum 10
* Pull requests: minimum 10
* Releases: minimum 2
* Stars: minimum 10
* Watchers: minimum 10
* Forks: minimum 2
* Exclude forks: True
* Has open issues: True
* Has open pull requests: True

The main selection criteria is the number of commits, as this measures the best how much work has been put into the project. All other variables might change in any direction over time, for example repos can be unwatched, unstarred, forks can be deleted or merged and contributors might decline over time. However, the number of commits always rises as time passes.

In our qualitative analysis and during its mining process we have seen, that the larger projects, which showed activity in a 28-day window regardless whether a release was coming up or not, had at least 5000 commits, this is why we used this number as a filter. The other criteria (contributors, issues, pull requests, etc\dots) are filtered, so that we are making sure that all of the GitHub functionalities are being used, e.g. issues are managed in GitHub, it has at least a minimum number of contributors so the networks will not be empty, and it has at least a small community due to having stars, watchers and being forked. We exclude forked results, as we are only interested in the main source code, and we mark the 'Has open issues' and 'Has open pull requests' to filter for active projects.

Applying the above inclusion and exclusion criteria, the search resulted in 2211 repositories in total. We further narrow this result set down to 110 by randomized selection. Our goal is to analyse about 100 repositories, but due to various mining and release problems, this number is expected to be further reduced, therefore we choose an extra 10 repositories to compensate the expected reduction.

Since following the semantic versioning standards is a crucial part of the analysis, we also exclude the projects, which do not follow the major-minor-patch versioning convention, or if they have multiple parallel editions, which are versioned separately. We have to exclude the latter category, as the version is determined based on the version number change compared to the previous version, and if the previous is denoting a different edition of the software, the version type can be misidentified. For example, the chakra-ui/chakra-ui repository contains releases named 'select@X.Y.Z', 'props-docs@X.Y.Z' and also 'progress@X.Y.Z', where X, Y and Z are integers. If the 'select@1.0.1' release is followed by the 'props-docs@1.1.6', then based on our version denoting algorithm, this would be categorized as a major release, even though the latter release is just a patch to 'props-docs@1.1.5', just chronologically it is not the preceeding release. Since exceptions and personalizations are common between repositories regaring versioning, we decided not to investigate further how these projects could be incorporated, as this would make the classifying algorithm significantly more complex. Instead, we propose an improved algorithm for future research.

## Repository mining

The above exclusion criteria results in 84 repositories, which we start mining with both git2net and repo\\_tools. Just like in the qualitative analysis, the data regarding the commits and edits, which are used to generate the collaboration networks, are gathered by git2net, and the releases and issues details are collected with repo\\_tools. During the two-week long mining process, 6 repositories were skipped due to the commits and edits mining running into an error or getting stuck. Presumably this is due to certain files (possibly large binary files), which require too many resources to compare the changes line by line. Part of the mining process is the disambiguation of commit authors, as renamed users can show up multiple times within the collaboration network. The original author names are kept in .csv files for each mined repo.

As a last step in the mining process, we mine the repositories with the repo\\_tools miner. Of the remaining 78 projects, 10 could not be mined for various reasons, including the miner getting stuck with some of them during the commits mining or the repository being renamed and therefore does not match the provided URL anymore. We show the summary of the 78 projects in Table 1 and the main programming languages and their distribution in Table 1. We also included the 10 repositories skipped, because they finished for the git2net mining process, but we will exclude them during the analysis going forward, and we will only work with the fully successful 68 repos. The full list of the 110 randomized repositories can be found in Appendix A.

## Metrics and release stats

We investigate the correlation between the network metrics and the releases with linear regression. First, we compose a list of all the mined repositories and all of their releases with the release date. Then the release regularity is calculated for each repository using the coefficient of variance value descibed in Section 1.2.3. Since this value requires more than one release, the CV is measured per repository and not per each release. Lastly, we calculate all of the collaboration network metrics for the network containing the past 28-day activity before the release, and the network covering the 28 days right after the release.

### Release classification

To calculate the CV value, the patch releases must be excluded as we mentioned in Section 1.2.3. First, we classify each release into major, minor or patch releases. Because the semantic version can only be determined by comparing a release with its preceeding version, the first release in each repository is dropped because its version will always be unknown as it lacks a previous release. Furthermore, this type of classification also has issues categorizing parallel version maintenance. For example, if a project parallelly supports a version v2 and a version v3, it issues new minor and patch releases for each version, so we will see v2.1.4, v3.0.2, v2.1.5 and v3.0.3 in chronological order. In this theoretical example, the releases v3.0.2 and v3.0.3 will be categorized as major releases, because their preceeding release has a lower major number. However, it is clear, that these should be patch releases, since they follow up on the previous patch for the respective version. In order to circumvent this issue, we would need to update the classifying algorithm to consider all the preceeding versions. However, this would significantly inclrease the complexity of the program, and therefore it is out of scope for our work. We mark a robust classification algorithm as a possible future topic to be researched.

Another problem with the version classification are the irregular release names, such as beta releases or release candidates. These versions usually have the same semantic version number, and they are only different in the optional tags at the end. As an example, during major releases it is common to see multiple release candidate versions following eachother, like v3.9.12 followed by v4.0.0rc1 and v4.0.0rc2. The issue rises from the fact that by looking at only the release number, the latter two releases are the same to the classifying algorithm (4.0.0), which leads to v4.0.0rc1 being classified as a major release, and v4.0.0rc2 will be unknown. We could mark these versions as rc versions, and skip them during the analysis, but the heterogenity of conventions is a great issue, as some projects mark release candidate, beta and alpha versions as v4.0.0-rc1, v4.0.0beta5 or even subversioning them such as v4.0.0-rc1.2. Some projects also have different editions marked with an \@ symbol, for example v3.1.5@latest and v3.1.5@browser-only, which also results in the same issue. To resolve all the releases, which are uncategorizable, we could only make it 100\% accurate without any unknowns if we manually modified or deleted the release data. Provided, that the 68 mined repositories contain over 5200 distinct releases, this task would be too cumbersome, and therefore we marked all releases affected by this as 'unknown'. Because we would like to perform a linear regression on our dataset, we must convert all nominal categorical labels to numbers, thus we add 3 columns to the current dataset named 'major', 'minor' and 'patch', where these values will have 1 in their respective release type column, and the other two will be 0. All three columns are zero if the release type is unknown. We also keep the nominal category in case of models, which can handle categorical values.

Because the semantic version can be highly unreliable in certain repositories, we take a look at how much was changed between two releases to identify the significance of the release. We do this by taking all commits between two relositories, and adding up the number of lines added and number of lines deleted. Since this method only requires the preceeding version's release date, it provides a measure of significance without the classification errors mentioned above. However, it is also important to note that the version type (major, minor, patch) is not necessarily related to the number of lines changed. The resulting dataframe is depicted in Table 1.

### Semantic version number and number of changed lines

First, we observe how the semantic version numbers predict the number of changed lines by setting the major, minor and patch columns as predictors and the Change size as the intecept (y value). Our null hypothesis is that the semantic version number and type does not have an effect on the number of lines changed in a new release compared to the previous version.

The OLS regression results with patch being the reference level can be seen in Table 1. We used the formula log(change) \sim C(repo) + C(type) , where the change is the number of lines changed, repo is the repository's name and type is the semantic version type as string (major, minor or patch). The log() is used for the number of lines changed because a larger release is expected to have exponentially more lines changed, and the C() signals the statsmodels package that the variable is a nominal variable. The resulting linear regression equation

with R^2 = .355 is found to be significant, based on which we reject the null hypothesis. Surprisingly, the major version's coefficient is twice lower than the minor version's, which is contrary to our expectations. The number of lines changed were expected to be higher in a major release than in a minor, but this was found not to be true. We partially contribute this to the misclassification due to parallel versioning, since most projects support only major versions parallelly - if they do. Because of this, the most falsely labelled versions are major versions, which causes inconsistency when determining the number of lines changed.

### Correlation between network metrics

In Section 1.2.3 we observed, that the collaboration network metrics can be dependent on eachother. In this section, we analyse their correlation by creating a matrix of correlations, which is depicted in Figure 1.

Correlation coefficients, whose value is between 0.9 and 1.0 are classified as very highly correlated. Values between magnitudes 0.7 and 0.9 are considered as highly correlated, between 0.5 and 0.7 the two variables are moderately correlated, and datasets with a correlation coefficient between 0.3 and 0.5 are categorized as having low correlation. Variables, whose correlation coefficient is between 0 and 0.3 do not have any significant correlation, as the two datasets are so different, that the occasional similarities are most likely caused by chance and no underlying correlation. These ranges are also applicable to values below 0 with a reversed sign, and the negative sign signaling an inverse relationship.

Based on these categories, the following correlation types can be seen:

* High positive linear correlation:
  + Number of nodes ~Connected components (0.88)
  + Number of nodes ~K-core count (0.76)
  + Core/periphery (k-core) ~Core/periphery (degree) (0.74)
  + Mean degree ~K-core count (0.73)
* Moderate positive linear correlation:
  + Mean degree ~Clustering coefficient (0.61)
  + Network density ~Core/periphery (k-core) (0.59)
  + Connected components ~K-core count (0.54)
  + Degree centrality core ~Core/periphery (degree) (0.53)
  + Network density ~Clustering coefficient (0.52)
* Low positive linear correlation:
  + Degree centrality ~Mean path length (0.43)
  + Number of nodes ~Mean degree (0.41)
  + Network density ~Mean degree (0.39)
  + Network density ~Core/periphery (degree) (0.39)
  + Mean degree ~Degree centrality core (0.39)
  + K-core count ~Degree centrality core (0.33)
  + Clustering coefficient ~K-core count (0.31)
  + Network density ~Mean path length (0.30)
* Low negative linear correlation:
  + Number of nodes ~Core/periphery (k-core) (-0.46)
  + Network density ~Connected components (-0.44)
  + Number of nodes ~Core/periphery (degree) (-0.37)
  + Degree centrality ~Degree centrality core (-0.36)
  + Connected components ~Core/periphery (degree) (-0.32)
* Moderate negative linear correlation:
  + Degree centrality ~Core/periphery (degree) (-0.57)
  + Connected components ~Core/periphery (k-core) (-0.51)

From these significant correlations and based on Figure 1 we can see that the number of nodes has one of the largest correlation with all the other variables. This rpoves, that one of the greatest influence on the network metrics is the number of developers, and in extension, the activity within the repository. The k-core count, which measures the number of nodes having a degree number within the top 20th percentile, is highly positively correlated with the number of nodes, which indicates, that the number of core developers in the network increases in proportion to the network size. However, the number of core developers classified by the degree centrality method does not indicate any relationship, which leads to the conclusion that these two methods do not identify the same collaborators as core members all the time, and there are significant differences. The low positive linear correlation between the two variables also confirms this observation.

The two derived core and periphery measures, however, are both negatively correlated with the number of nodes, indicating that as the network grows, the core becomes relatively smaller within the graph, meaning that most of the network growth can be contributed to the increased number of periphery developers. This observation is in line with the findings of McClean et. al. , who found that the number of core members tend not to change. The two core-periphery ratio metrics are also show a high positive correlation, which shows that they both cover the same concept and measure the same property of the network.

### Pre- and post-release network metrics and lines changed

We theoretize based on our observations so far, that the larger the change is within the network, the larger the release is, and consequently more lines are changed. We test this theory with an OLS linear regression for four chosen network metrics, which proved to be closely related to the collaboration network in any relation.

The mean path length shows what is the average of the shortest distances between all pairs of developers in the network. From an organizational point of perspective, this translates to how fast information can reach a developer from another developer in the best-case scenario (meaning always taking the shortest route) on average. Our theory is that during normal, 'business as usual' times, there is a core developer team, which is collaborating with all the periphery users, who are reporting bugs, suggesting new features and helping in the development. This causes the mean path length to be relatively high between periphery members, as they are all connected to the core, but not necessarily to eachother. However, during the preparation of a (larger) release, the tasks shift from development to testing, reviewing and integration. This could prompt the periphery members to interact with other periphery members, and not directly just with the core, which ultimately lowers the mean path length before a release as less interdependence is required. After a release, we theoretize, that the focus shifts from development to planning, which again emphasizes the core developers and their activities, thus creating a more centralized structure and raising the mean path length.

Although mean path length indicates the shortest path for information to travel, which somewhat describes the level of collaboration as well, we use the clustering coefficient to measure the clusteredness of the network. The clustering coefficient is calculated by finding triangles within the graph: 'Is the neighbour of my neighbour also my neighbour?'. A high clustering coefficient indicates that a relatively large number of these triangles are present compared to all the triangles possible. From a project management perspective, a high clustering coefficient means strong groupwork and coordination, as the developers are clustered together, where most of them are connected to eachother. We expect the clustering coefficients to rise before a release, because the tasks shift from individual development to integration and regression testing, plus the coordination effort from the core members also increases, which promotes cross-checking tasks and reviewing eachother's code. The clustering coefficient is expected to drop after a release, as activity becomes normal again, when everyone is working on their own sections and functions.

Hierarchy, as the third choice of metrics, considers both the clustering coefficient and the degree number by setting a trendline on a clustering coefficient-degree number plot, and takes the slope of this trend. Hierarchy describes how strict the management roles are in the network. A node with a low clustering coefficient, but a high degree number is likely to play an organizer or management role in the development process, as they are not collaborating on the source code as much as the other developers, but rather they enable said developers by connecting them to other groups. In networks, where there is a high rate of hierarchy, the clustering coefficient decreases much faster by the increase of degree number, which results in a steeper declining trendline. Therefore the lower the slope is, the mire hierarchical the network is. Regarding the releases, due to the increase in activity before a release from all parties, our expectation is that the network gets less hierarchical due to the more cross-group collaboration, which causes the slope to rise. We expect the hierarchy to rise immediately after the release, because the next development cycle needs to be planned, which requires organization and a highly structured breakdown of the backlog. Then later it gets back to normal levels between two releases.

Lastly, we take the core-periphery ratio measure. This metric is a percentage of how many developers in the network are classified as core members compared to the overall number of nodes within it. We have two methods to classify a developer as core. The first is the K-core method, which is the number of nodes having at least k degrees, and the second is the degree centrality core method, with which we simply calculate the degree centrality for each node. For both methods, we consider a developer core, if they are in the top 20th percentile of nodes. For both K-core and degree centrality core, the percentile is calculated starting from 0 and not from the lowest value within the network. This ensures, that in time periods, when only the core are active, all of the core members can be still identified as core, since their absolute values are still high. Before a release, the core-periphery ratio is expected to fall, because more periphery developers are being involved in the testing and intergartion tasks, but the number of core members stays the same during the project's lifecycle. This causes the ratio to fall as the overall number of nodes increase, but not the number of cores. After a release, this is expected to rise again due to the project focus shifting to planning and organizing, which involves more the core development team. \\

In order to confirm or reject the above hypotheses, we perform an Ordinary Least Squares (OLS) linear regression on the mentioned network statistics before and after a new release. Our assumption is, that the larger the network change is before and after the new version, the larger the release is. We use the following formula for this:

where m\_1 is the network metric value for one of the four chosen measures before the release, m\_2 is directly after the release, N\_1 is the number of nodes in the network before the release, and C(repo) is the categorical value of the repository name as a dummy variable. The 'before' metric is calculated for the network generated from the period starting 28 days before and ending on the release day, and the 'after' value is calculated from the release day until 28 days past the release. We use the difference between the 'before' and 'after' values to represent the change. The number of nodes is also added in the regression to normalize for the inherent differences that stem from the project's size.

The regression results are depicted in Table 1. We run the regressions for both the logarithm of the change size, and the change size itself for all the metrics one by one, a total of 10 runs. With a the alpha \alpha = 0.05 significance level, the only variable where p < \alpha is the clustering coefficient in the normal, non-logarithmic change size regression with a value of p = 0.04. Based on this result, we can only reject the null hypothesis for the clustering coefficient, and we cannot reject it for the other metrics. Surprisingly the coefficient of the clustering coefficient is positive, which is contrary to our expectations. Our theory expected the clustering coefficient (CC) rise before a release, and drop afterwards, which would result in a negative value when subtracting the 'before' value from the 'after' value. Therefore, in the regression model the coefficient of CC would be negative, indicating an inverse relationship between the number of lines changed in the release and the CC change. Our findings, however, provide evidence for a positive linear relationship, which indicates that a unit of increase in the CC after the release corresponds to \num{4.2e+8 more lines changed in the release. Since the CC can only be between 0 and 1, we use 0.1 as a base unit, which means that a 0.1 increase in CC from before to after a new version causes the release to have \num{4.2e+7 more lines changed on average.

### Release size and issues opened and closed

One explanation could be for the increase in clustering coefficient, that a larger release with more lines changed leaves more room for errors and bugs within the code. The increase in issues leads to a higher bug fixing activity within the project, which requires cross-checking and integration tests, and ultimately causes more developer triangles to form, which raises the clustering coefficient value. To test this theory, we perform a similar OLS linear regression, but with two new metrics: number of issues opened and number of issues closed. If our assumptions are correct, there should be also a positive linear relationship between the release size measured in number of lines changed and the difference between number of issues opened after and before the release. We also perform the regression for the number of issues closed, however, since the observed period after a new version is 28 days, but an issue might take longer than that on average to fix, we might not see the surge of new issues closed within this period, therefore we do not expect a significant result for the number of issues closed.

Since the number of issues are not related to the collaboration network adn they are unrelated variables, we leave it out of the regression. Also, we use the values from before and after the release separately, and without subtracting them from eachother. The network metrics needed to consider the change between the 'before' and 'after' values because the change is measured from 0 on a scalar scale, but that is not necessarily the normal value in a network. The number of issues, on the other hand, are 0 by default, when there is no activity or no issues being created.

Table 1 shows the OLS regression results for the number of issues opened and closed, both for 28 days before the release date and 28 days after it, respectively. We run the model for both the logarithm of the n umber of lines changed and the normal values separately. Two of our variables do not exceeed the \alpha = 0.05 significance level, these are the number of issues closed before, and number of issues closed after a release, both with the logarithmic scale of number of lines changed.

### Normalizing the network metrics

The linear regression models helped us understand how much the developer network changes at a new release compared to the release's size by contrasting the immediate period before and after the release. In this section, we investigate how the network metrics can be normalized.\\

Due to the inherent correlations between network size and the other metrics, such as clustering coefficient or average degree, we always need to make sure, that the values being used are normalized or the size difference between networks is controlled for. This is why we used the Z-value in the qualitative analysis section (Section 1.2.3) and this is also the reason why the network size was included in the regression models when the network metrics were used. The Z-value metrics provide a convenient and reliable way of eliminating the role of network size, as the actual network is being compared to many same-sized random networks. However, the Z-value generation only works, when the network size is sufficiently large. As descibed in Section 1.2.3, the Z-value divides by the standard deviation of the random network's metric being calculated. In cases, when the standard deviation is 0, the Z-value cannot be interpreted, as this leads to a division by zero error. The standard deviation can only be 0, if all values within the sample are exactly the same, and this scenario often plays out when the size of the network is small. As the size of the network becomes smaller and smaller, there are fewer variations to organize the N nodes and E edges, which raises the possibility of the random networks to be repeated by generating the same network just by pure chance. Furthermore, when calculating the network metrics, the nodes and edges are not differentiated in any way: the metric will be the same, whether developer 'A' collaborates with developer 'B' or developer 'C'. This variability also increases the random networks having the same values for the measured metrics.

In our collection of repositories, this phenomenon occurs often, because there are a relatively low number of nodes within the networks generated immediately before or after a release (usually around 8-9), and therefore most of the calculated Z-values cannot be interpreted. In order to correct for the disproportionality the network size can cause, so that the graphs can be compared across all projects and sizes, we use an analytical approach to normalize the values. By dividing the theoretical maximum value of a metric in terms of its size, this is achievable.

The mean path length is calculated by averaging all the shortest path lenghts between all pairs of developers. The theoretical maximum value of mean path length l\_G^\* in terms of graph size is l\_G^\* = (n-1)/2 of network G(N,E), where |N| = n the number of nodes. Therefore, we calculate the normalized mean path length as follows:

When it comes to the clustering coefficient, however, it is impacted differently by the size of the network, because the CC is more or less independent from the number of nodes, and it is more related to the number of edges and the degree number. The clustering coefficient takes on its maximum value of 1 only in complete graphs, as the number of triangles within the graph is at maximum when every node is connected to all other nodes. Therefore, we normalize the CC values with the formula used by Gentner et. al. for the maximum clustering coefficient of a k-regular connected graph:

where k is the degree number of nodes ina k-regular graph.

The core-periphery ratio already takes into account the number of nodes, which is a proxy for the network size, as the CP-ratio is calculated by dividing the number of nodes in the top 20th percentile of either degree centrality or k-cores, and then dividing by the overall number of nodes, meaning there is no need for normalization.

Hierarchy is more complex than the rest of the metrics, as it uses the combination of clustering coefficient and degree number to plot a scatter plot, and the slope of the linear trendline is used as the hierarchy value. By having more nodes in the graph, the trendline does not necessarily change, and it stays the same if the level of hierarchy does not change. For example, if a new team of developers appears in the network, who have a team leader just as all other teams, then the overall hierarchy of the network did not change, as managers still have the same number of developers under them in the organizational hierarchy. In contrast, if more people join the already available developer teams, a team leader will have more developers to manage, which causes the network to become less hierarchical. As we demonstrated, a network can grow in a manner that changes the hierarchy, or independent from it, and therefore we conclude that there is no need for normalizing the hierarchy metric.

### Network measure trends

After applying the normalization described above on our dataset, box plots are created to show the overall trend in measures at a release. However, this requires to gather and calculate metrics for more than two time periods (before and after a release) in order to well represent the changes over time. In order to make the plots clearer, we shift the time window, so that the chosen day is not at the beginning, but in the middle of the observation period. This is depicted on Figure 1, where the numbers on the timeline represent the number of weeks passed relative to the release day.

Each metric's normalized value is calcualted for each release's corresponding five timespans (from t\_{-2 to t\_2). With this approach another issue arises: releases, which were within 2 months of the data collection, will only have partial data available for the last 3 timespans. This can skew the results, since for these values the results will be 0, and it would falsely show a decline in activities. Therefore, we filter out those releases, which were created after 1 January, 2021.

The results for mean path lenght can be seen on Figure 1. Contrary to our expectations, we see an increase in mean path length up until the release, and a slight decrease afterwards. A decrease in mean path length means less interdependency amongst developers, as fewer people are fulfilling the 'middle man' role within the network, which was expected at a release since closer collaboration is required and different tasks, therefore developers have to move out of their usual role and collaborate with those, who they do not collaborate often with. The oppasite in the results suggests that the level of activity is not as high as expected at a release. The likely reason is that around the releases not many changes are approved, and the focus is on finishing up the current tasks. Before the shorter mean path length can be explained with the rush to finish the tasks, and after it can be explained with the bug reports and new issues, and consequently their activity from the developers' side.

Because due to the heterogenity of each project, there could be large differences and patterns different than the one descibed above. Therefore, we plot each repository's mean path length boxplot separately. We exclude repositories, that have less than 10 observed releases before 1 January, 2021, because that would reduce the number of releases aggregated on the plot, and the unique traits of the single particular releases would show.

After observing the total 53 plots, we identify four main reoccurring patterns, that can be observed within the projects. We name these patterns based on the two half-curve's steepness, that would connect the five boxes depicted. These are:

* Flat (11): Repositories followin this pattern barely change at all, and their mean path length stays the same in all five timespans regardless of the release.
* Rise-decline (11): Projects falling into this category mainly follow the overall pattern depicted in Figure 1, which shows a rise until the release, then a decline afterwards.
* Flat-decline (6): These projects stay flat until the release, but after the release, their mean path length drops.
* Decline-rise (6): Repositories following this pattern show the opposite observation than the aggregated view, that is, their metric declines until the release date, then it rises again.

Examples for all four categories are shown in Figure 1.

We also performed the same analysis on the normalized clustering coefficient, however, the results are inconclusive, as the majority of all release's normalized clustering coefficient is 0. Therefore, we removed the releases from the plot data, which had at least one CC value equal to 0. The resulting box plot (shown in Figure 1) follows a flat-rise pattern, which is especially visible at timespan t2, where the box is significantly higher than the other periods. However, the removal of 0-values reduced below 10% the number of releases included in the observation, therefore we cannot conclude anything definitive regarding the clustering coefficient.

The same method is used to visualize the number of issues opened and closed for the releases. This is depicted on Figure 1.

# Discussion and results

# Conclusion and future work