Table of Contents

1. Introduction

Our dataset involves the Bugzilla data from Mozilla over the period 1997-2003. Bugzilla was used to track and manage bugs and other issues related to Mozilla's projects, such as the web browser Firefox. The Bugzilla data includes bug reports submitted by users and developers, along with information such as bug descriptions, comments, timestamps, status changes, and other relevant metadata. The data could be analysed to identify patterns, track bug resolution times, identify recurring issues, and extract valuable information to improve software and other development practices.

2. Data Manipulation, Cleaning and Structuring

Our first step was importing the data into the database management system, in this case, MongoDB. We then performed some cleaning checks but decided to split the data into collections before finalising the cleaning.

Bugzilla data involves complex and hierarchical structures, including nested fields, arrays, and potentially changing schemas. MongoDB's document-oriented model and flexible schema make it well-suited for storing and querying such data. If the Bugzilla data has a highly variable structure or requires frequent schema changes, MongoDB may offer more flexibility compared to the rigid table-based structure of PostgreSQL. MongoDB's distributed architecture and sharding capabilities can provide better scalability and performance compared to PostgreSQL; also, the ability to distribute data across multiple servers allows for efficient scaling.

MongoDB allows for complex aggregations, grouping, and pipeline stages, enabling efficient data transformations without the need for extensive data movement. Applications like MongoDB Compass and Atlas make it easy to explore and manipulate data, create and manage indexes, run queries, and perform administrative tasks. MongoDB Compass is also a powerful tool to analyse data within various schemas and present data in the form of charts, plots, histograms, and even geographical representations, making visual inspection easy.

2.1 Data Manipulation

Using the PyMongo Python library we efficiently imported all pickle files, extracting the value of DateTime objects and converting them to a format compatible with MongoDB. We then inserted all bugs data into a collection in our MongoDB database. To manage the large number of files, we created a for loop to locate and handle all pickle files within a specific folder. Using this iterative method, we

streamlined the data entry procedure, transforming the DateTime objects and uploading the documents to MongoDB in batches of 1000 JSON files, enabling us to handle the substantial dataset with ease.

2.2 Quality Checks

Initially, we looked at the whole database and conducted some quality analysis. We found the dataset contained several fields with too many missing values, some columns with more than 99% of their values being "---", and several DateTime fields with the wrong datatype. Besides, we also found 1860 documents had only an id so we dropped them as well. We did not find any duplicated observations.

At this stage, we decided to update the DateTime fields to all be DateTime datatypes, but we did not drop any fields or documents because we wanted to see whether they played an important role in the individual collections.

2.3 Structure

After performing the quality checks, we decided to restructure the dataset, so it is easier to manipulate and understand. After considering the definition of each field (MozillaWiki, 2022), we split the whole dataset into the following collections as these were the main 'themes' we identified:

- Main bugs information
- Tracking information
- People information
- References
- Further details

With this clearer and more manageable structure, we continued our data cleaning process collection by collection.

2.4 Data Cleaning by Collections

2.4.1 Main information

The main collection consisted of the field 'cf_fx_points' which had less than 1% of actual values so we dropped it.

2.4.2 Tracking collection

In Tracking section, we combined similar fields into one dictionary field for a hierarchical structure (See Figure 1).

We combined the tracking flags ("cf_status_..." & "cf_tracking_...") into one field; each product (e.g. Firefox, Seamonkey) is a key, the values are the flags' info, in listed dictionary form, each dictionary contains the version, status flag, and tracking flag.

We also combined the 4 project flags[1] into a 'project_flags' field, and all the status information fields[2] into a 'status_info' field.

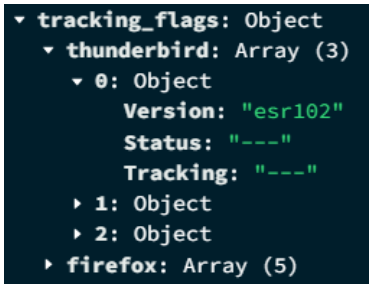

Figure 1: *Structure of Tracking Flags*

2.4.3 People collection

The People collection required minimal pre-processing as there were only some null values in the "qa_contact_detail" field and we decided not to examine the "_detail" fields as our focus was on descriptive statistics for bugs related to specific roles, and personal information was deemed irrelevant.

2.4.4 References collection

We dropped the "regressions" and "regressed_by" fields because they contained no values.

2.4.5 Extra information collection

This collection is a mix of details and information on some bugs, and some might be valuable, but other fields such as 'cf_qa_whiteboard' and 'alias' provide no information (+99% missing), so we removed them.

[1] 1) cf_a11y_review_project_flag, 2) cf_accessibility_severity, 3) cf_performance_impact, 4) cf_webcompat_priority'

[2] 1) status, 2) resolution, 3) dupe_of, 4) duplicates

3. Descriptive Insights

Following our structure logic, we will provide descriptive insights for each collection as a first approach to analyse this data.

3.1 Main Collection

We had varied information on the main collection about the bugs classification, the products or the priority and severity.

Firstly, we consider bug classification (See Figure 2) and Operating System (See Figure 3). The figures clearly indicate many of the reported bugs are classified under components or client software, providing valuable insights into the primary sources of these issues.

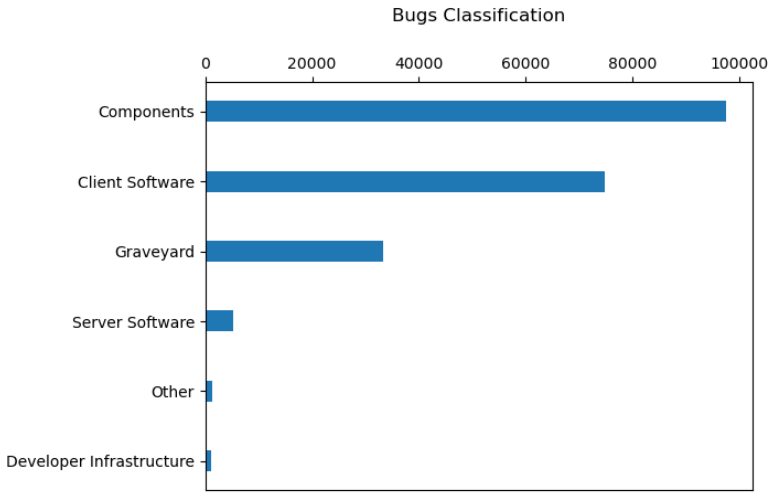

Figure 2: *Frequency of specific bug classification*

Notably, Windows emerges as the most affected operating system, although a deeper analysis shows the number of reported bugs associated with Windows (+15) is notably higher compared to other operating systems. Additionally, the presence of some bugs affecting all systems poses further challenges for the team, as they continue to grapple with addressing these issues.

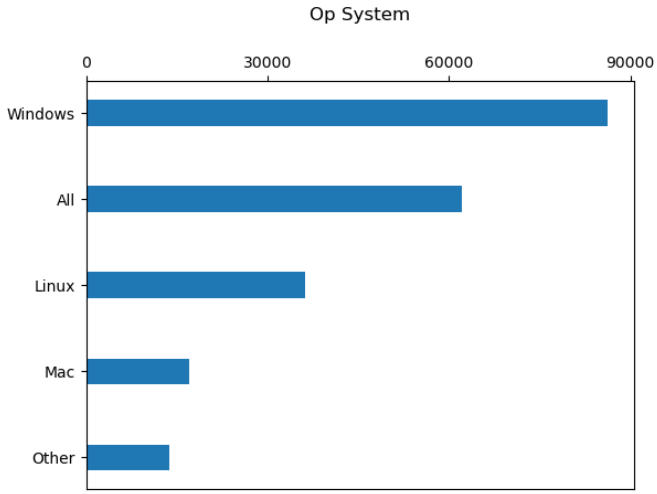

Figure 3: *Frequency of bugs in specific operating systems*

One of our focuses of interest was the bug's priority and severity attributes. It was observed (Fig 4) that a significant majority of bugs were classified with a 'normal' severity, accounting for over two-thirds of the total.

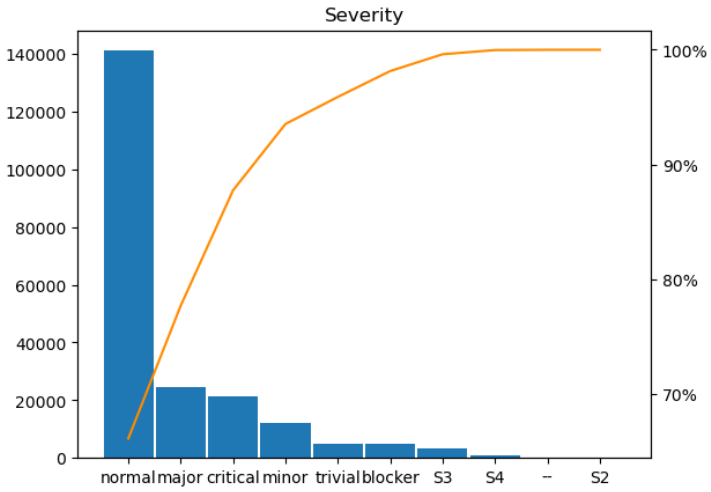

Figure 4: *Pareto chart for Severity ratings of bugs*

However, the prioritization of these bugs displayed a less polarized distribution (See Figure 5). Notably, a substantial portion of the bugs had not been assigned any priority, as denoted by the label '--' indicating a pending decision. Further investigation into the relationship between these two variables will be conducted in subsequent sections.

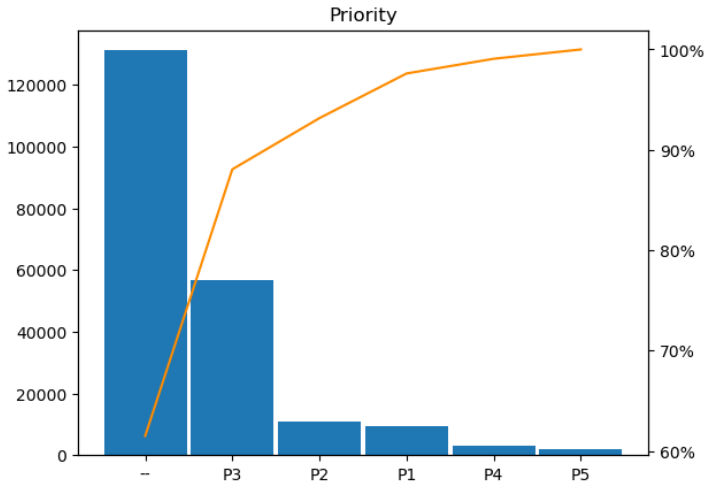

Figure 5: *Pareto chart for Priority ratings of bugs*

To conclude this subsection, our analysis reveals that 'Core' and 'SeaMonkey' stand out as the products with the highest incidence of reported bugs with 75% of the bugs, as is represented on Fig 6. Further investigation into these specific products could provide valuable insights into the underlying factors contributing to this outcome. Regrettably, at this time, we lack access to detailed data information that could facilitate a deeper understanding of the root causes behind these results.

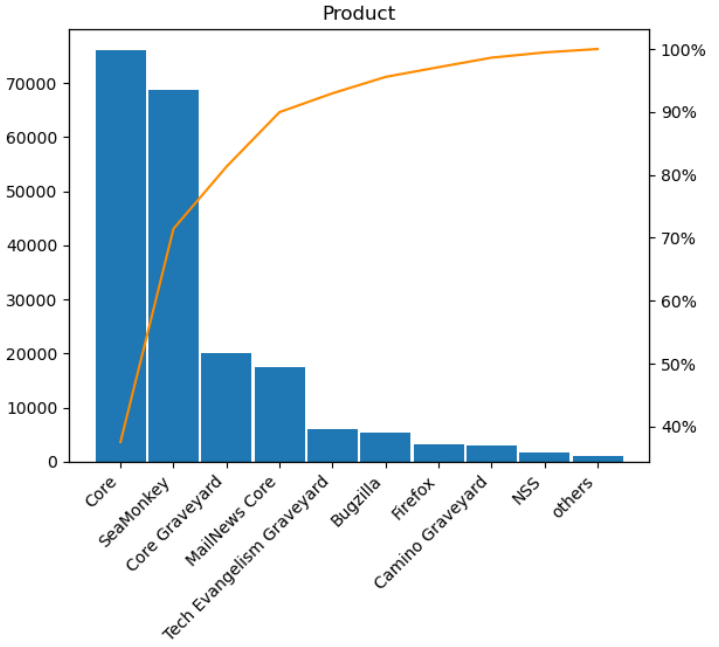

Figure 6: *Pareto chart for products classification*

3.2 Tracking Collection

Once reported, a bug will be assigned a status viz. unconfirmed, new, assigned, resolved, verified, reopened, and closed. ‘unconfirmed’ and ‘new’ statuses are assigned when a bug is reported. ‘assigned’ status is for bugs that are assigned to a developer to find possible fixes. Once a bug is resolved by the developer it is verified by quality assurance and is either assigned status ‘closed’ or ‘reopened’. A bug having status ‘resolved’, ‘verified’ and ‘closed’ can be considered to be in the final stages of resolution. This process is represented in Figure 7.

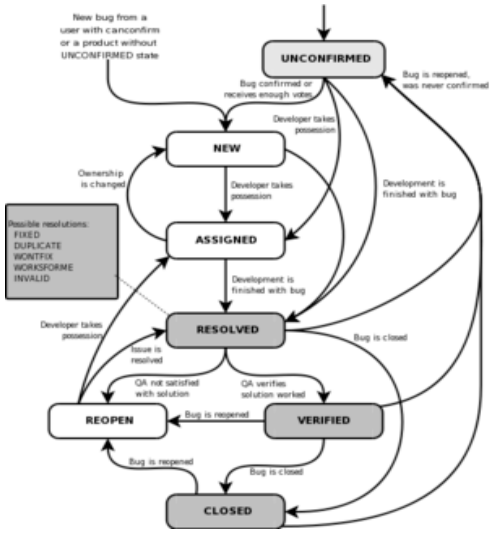

Figure 7: *Process of assigning a bug status*

To analyse the resolution of reported bugs, we have selected "id", and “status” data fields from the ‘Tracking’ collection and plotted Figure 8 below. We have segregated the ‘ids’ into buckets of 10,000 bug ids each. The bug_ids are assigned in order of reporting. We can clearly observe that a ‘bug_id’ is more likely to have its status ‘verified’/’closed’/’reopened’ if the ‘bug_id’ value itself is small i.e., the bug was reported earlier rather than later.

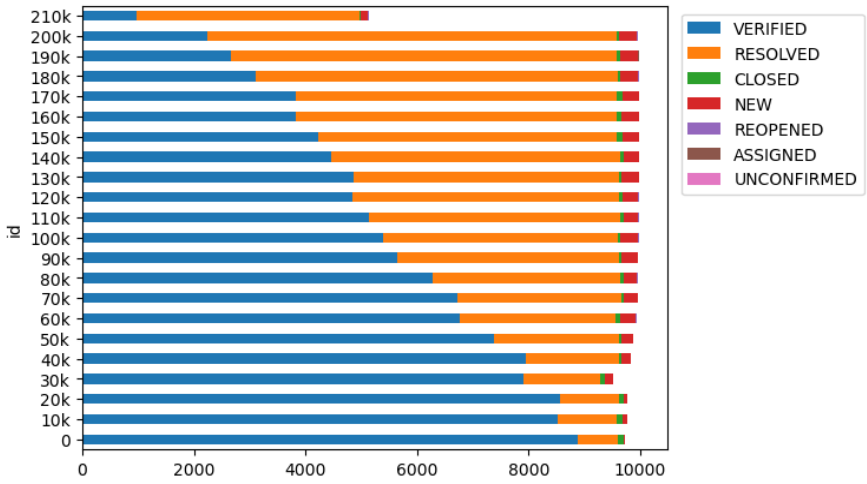

Figure 8: *Status distribution by id groups*

We also plotted the resolution distribution with respect to the status of the bugs (See Figure 9). Here again, we observe that as the bug_id increases in value the probability of it being fixed decreases as newer bugs will need time to reach the 'fixed' resolution status.

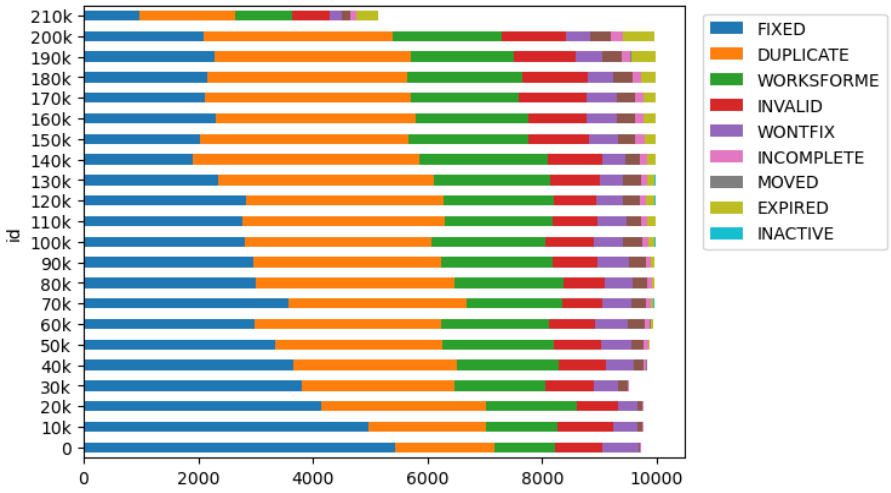

Figure 9: *Resolution Distribution by id groups*

Considering the resolution of bugs more broadly, we plotted the distribution of time needed to resolve a bug at the monthly (See Figure 10) and daily level (See Figure 11). We calculated the time needed as the difference between 'cf_last_resolved' and 'creation_time'.

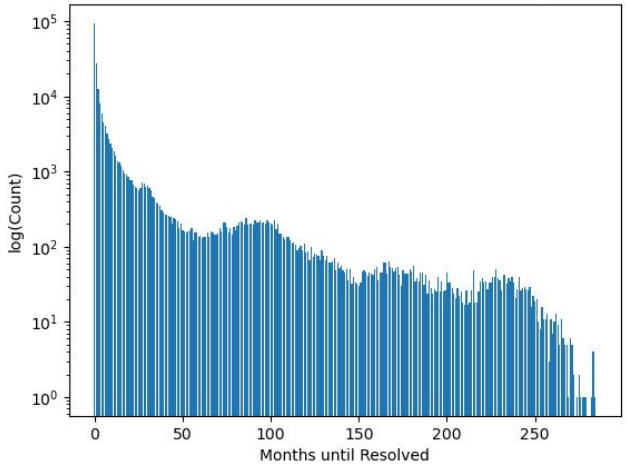

Figure 10: *Distribution of time needed to resolve a bug in Months*

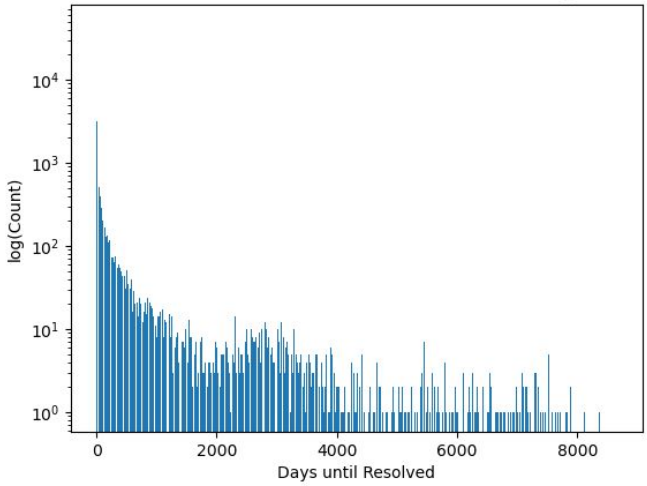

Figure 11: *Distribution of time needed to resolve a bug in Days*

Figures 10 and 11 indicate that while a lot of bugs are solved in a short amount of time, there are still some bugs that can take many months, indicating there might be some inefficiency in the bug resolving process.

We can also observe a clear increasing trend with the duplicate bugs. To analyse the duplication of bugs in more detail we have plotted a histogram of frequency (Log transformed) of duplicate 'bug_ids' (See Figure 12). We can observe that maximum frequency can be observed for zero duplicates i.e., unique bugs. But observing the distribution of the data suggests a normal range of 0 to 50 duplicate bugs are observed for a given 'bug_id'. We can also observe a few extreme cases with more than 250 duplicates.

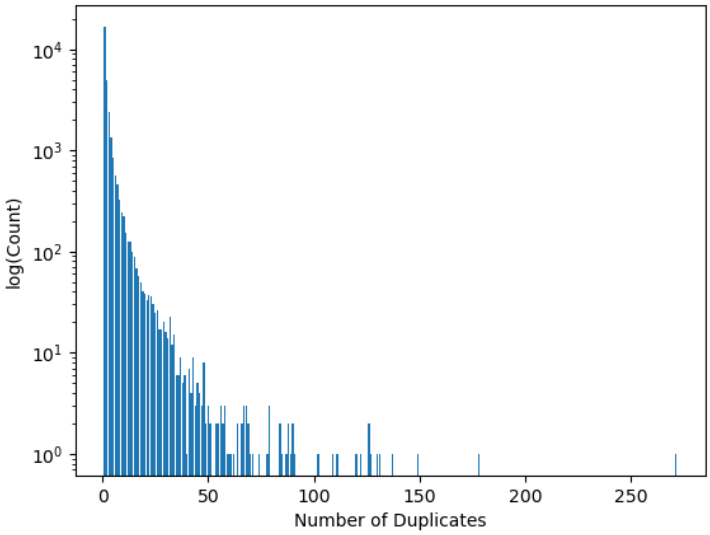

Figure 12: *Distribution of Number of Duplicates*

3.3 People Collection

The boxplots for all roles, except the mentors, revealed significant variability, with a wide range of data points (See Figure 13). Additionally, the data exhibited skewness, with a small number of individuals accounting for a majority of bugs, leading to some outliers with unusually high frequencies. This skewness might be attributed to the length of an individual's tenure at Mozilla, as our dataset spans 6 years, long-serving individuals would intuitively have more frequent occurrences than new recruits.

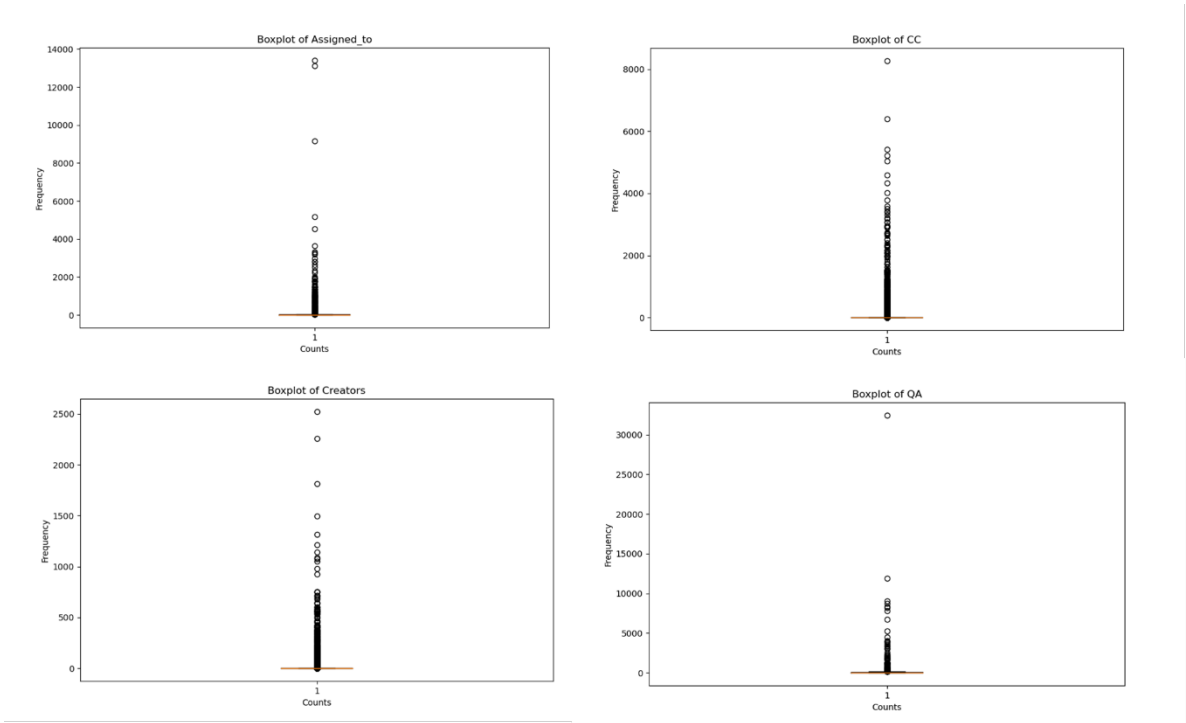

Figure 13: *Boxplots indicating the frequency of a value for the four different roles*

The mentors column contained a limited number of values, all related to one or two projects at a time, perhaps indicating the mentor role is a big job and cannot handle multiple bugs at a time.

During our analysis, we also explored whether individuals took on multiple roles, and indeed, we found instances where values appeared in multiple roles. Regarding the “CC” role (meaning the individual wants to know the progress of this bug but is not directly involved in it), this may indicate some individuals assume this role in different bugs, because they might be waiting for certain bugs to be resolved before addressing their own, prioritizing their bug-solving efforts accordingly.

3.4 References Collection

In reference collection, we have selected "id", "depends_on", "blocks", "priority" and "severity" fields and plotted two figures (Figure 14 and Figure 15) to visualize the dependency and blocking relationships between each bug and their corresponding priority and severity levels. The values on the x-axis, from left to right, represent the older to newer bugs, while the y-axis represents the number of bugs. The figures include two colour scales on the right side, which depict the levels of priority and severity.

The top left figure illustrates the frequency of bugs appearing in the "id" column (depending bug). The average number of depending bugs is close to 1, indicating that at least one bug needs to be resolved before resolving the depending bug. However, in some cases, resolving a specific depending bug may require addressing more than 200 bugs beforehand. Furthermore, the maximum number of bugs that may need to be resolved first can reach around 1200. This represents an extreme case where certain bug IDs have an exceptionally high number of dependencies. Resolving such bugs would involve tackling a substantial number of related issues.

Furthermore, the top right graph illustrates the frequency of bugs appearing in the "depends_on" column (depended bug). Most depended bugs have at least one bug depending on them, and the range of dependencies varies from 10 to over 60. This represents a significant level of interdependence among the bugs within the bug tracking system. Hence, resolving such bugs would involve tackling a substantial number of related issues. Therefore, it is necessary to develop a clear plan to prioritize the task of bug resolution accordingly.

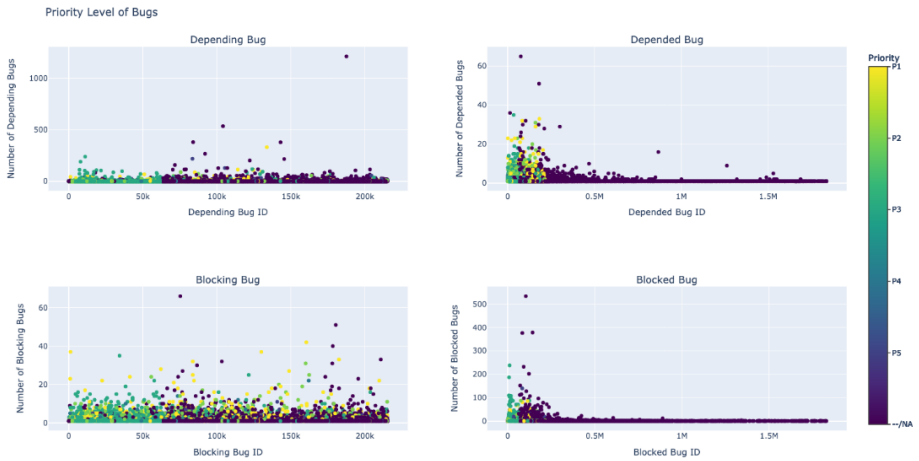

Figure 14: *Number of depending, depended, blocking and blocked bugs colour-coded based on priority level*

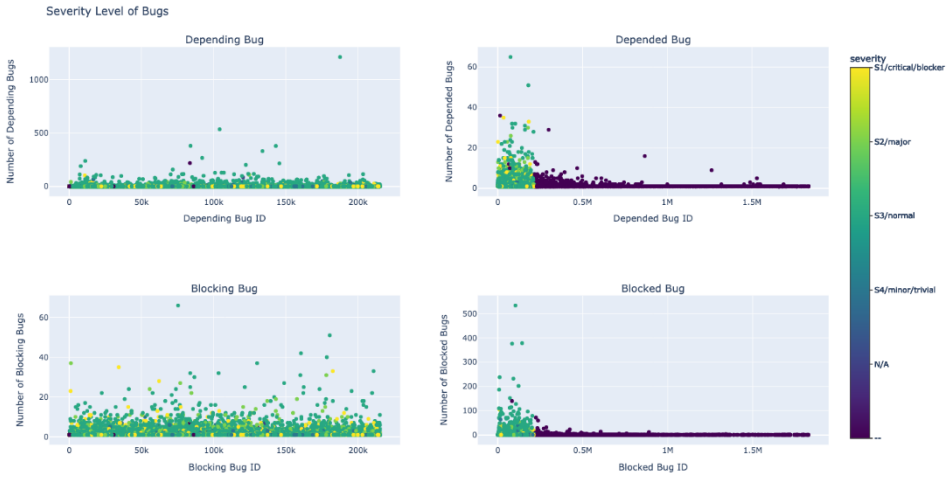

Figure 15: *Number of depending, depended, blocking and blocked bugs colour-coded based on severity level*

In "blocks" column, the bottom left graph illustrates the frequency of bug IDs (blocking bugs) appearing in the "blocks" column. The distribution of the number of blocking bugs is quite diverse, indicating that most bug IDs are blocking not just one bug, but multiple bugs. In extreme cases, a bug ID may be blocking more than 60 bugs. The bottom right graph displays the number of bugs being blocked by other bugs. The number of blocked bugs ranges from 50 to over 500, indicating that resolving these bugs requires prioritizing the order in which the bugs that are blocking them are addressed. Therefore, it is necessary to develop a clear plan to prioritize and resolve the blocking bugs first, as their resolution will have a significant impact on resolving the bugs that is being blocked by them.

3.5 Extra Collection

Our last step regarding insights analysis was to gather all the keywords attached to each bug and represent them (See Figure 16).

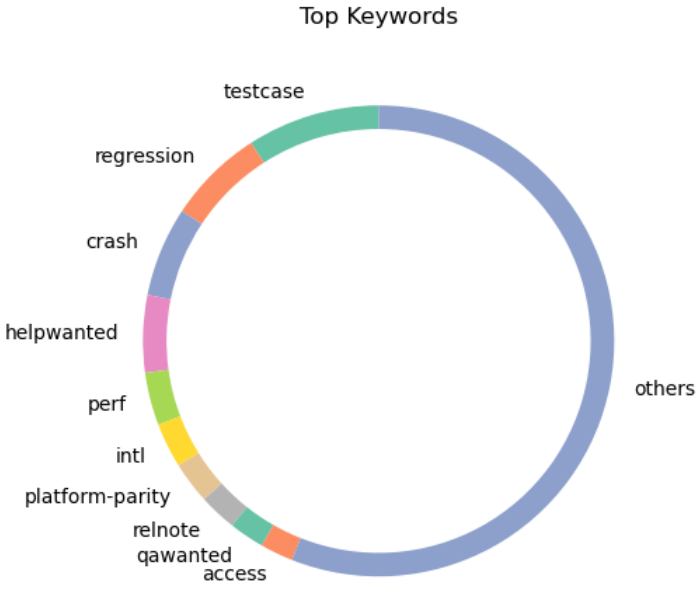

Figure 16: *Top Keywords used to represent bugs*

Our analysis reveals that the testing and regression classes are notably susceptible to bugs in the reporting process. Furthermore, a striking observation is that the top 10 most frequently repeated keywords account for nearly 50% of all the keywords used. This significant concentration of keywords suggests that the classification of keywords is currently being executed accurately and with clarity.

4. Insightful Data Analysis

Reconsidering Figures 14 and 15, we have color-coded each bug ID on every graph based on its priority level, which includes "--", "P1", "P2", "P3", "P4", and "P5". According to the Bugzilla Field Description, "P1" is the highest priority level, followed by "P2", "P3", "P4", and "P5", with "--" representing no decision. Overall, the majority of bugs with a priority level of "--" occupy the largest portion of bugs in all graphs, indicating that their priority levels have not yet been determined. However, for older bugs, other priority levels such as P1, P2 and P3 have been assigned. This suggests that when there were fewer reported bugs, individuals had more time to assign priority levels and address them promptly. As the number of reported bugs increased, prioritizing bug resolution became more challenging.

In depended bug graph, there are bugs that have over 20 dependencies. Among these bugs, some are scheduled to be fixed in the current or next release cycle, which is reasonable as these bugs should be addressed promptly. However, there are extreme cases where bugs have more than 50 dependencies and still have an undecided priority level. This situation can potentially cause delays in resolving the bugs that depend on them. On the other hand, in the blocking bug graph, the priority level is assigned more effectively compared to the dependency graph. Bugs that block more than 20 other bugs are given higher priority and are prioritized to be resolved as soon as possible.

Lastly, we have color-coded each bug ID on every graph based on its severity level ("--", "N/A", "S4/minor/trivial", "S3/normal", "S2/major", "S1/critical/blocker"). According to the Bugzilla Field Description, "S1/critical/blocker" represents the highest severity level, followed by "S2/major", "S3/normal", and "S4/minor/trivial". The "--" indicates the default value for new bugs, and "N/A" indicates that the severity definitions are not applicable. In the depending bug and blocking bug graph, most bugs have a very high severity level, regardless of whether they are new or old, or how many bugs they depend on or block. On the other hand, older bugs in the depending bug and blocked bug graph tend to have higher severity levels, while some newer bugs require an update to their severity level, and some of them have severity definitions that are not applicable to them.

We also examined the relationship between bug priority and severity using a heatmap visualization (See Figure 17).

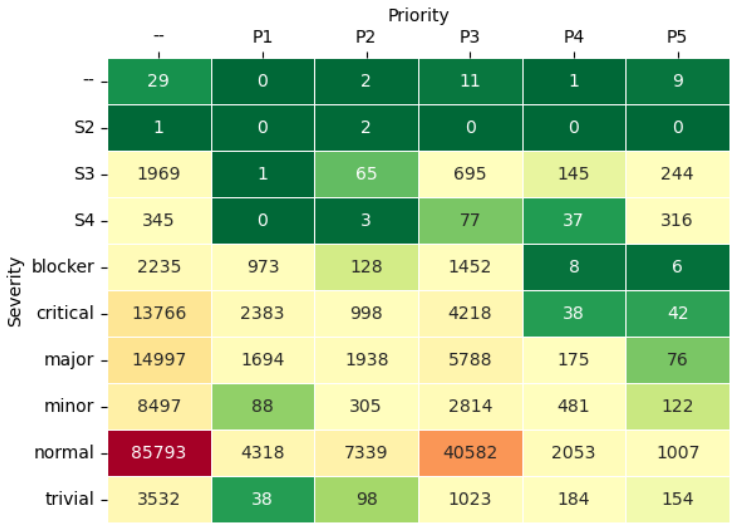

Figure 17: *Relationship between Priority and Severity*

Initially, we anticipated observing a pattern where less severe bugs (trivial, minor, normal, etc.) would be associated with lower priority levels. Surprisingly, this was not the case, suggesting a need for improvement in the classification of bug priorities based on their severity. Particularly, it appears

illogical that bugs categorized as 'normal,' which belong to a non-severe class, are not adequately distributed among the lowest priorities (P4 and P5).

To further investigate potential biases in this relationship, we employed a contingency table analysis (See Figure 18).

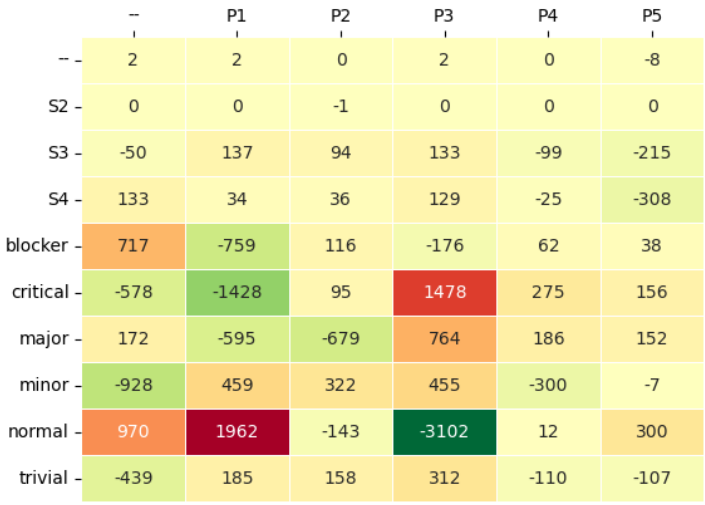

Figure 18: *Contingency table analysis*

The results revealed a significant positive correlation between bugs with 'normal' severity and those assigned to P1 priority. Similarly, a positive relationship was observed between bugs of 'critical' severity and their assignment to P3 priority. These findings imply that there is an overrepresentation of bugs falling into these combinations, diverting valuable resources to address P1 bugs even when there might be a greater number of 'normal' severity bugs within that priority level.

Conversely, we identified a negative relationship between 'critical' severity bugs and their assignment to P1 priority. This suggests that certain critical bugs are misclassified and assigned to priorities other than P1, potentially leading to critical issues not receiving the immediate attention they require during bug resolution.

In conclusion, our analysis highlights the need for refinement in the bug classification process, particularly concerning the alignment of bug severity with appropriate priority levels. Addressing these discrepancies will optimize resource allocation and enhance bug reporting efficiency, ultimately leading to more effective bug resolution practices.

5. References

MozillaWiki. (2022) *BMO/UserGuide/BugFields*. Available at: https://wiki.mozilla.org/BMO/UserGuide/BugFields (Accessed: 30 June 2023).