SMM695

Data Management Systems

Descriptive Analysis

Chuqiao Xiao

22.Sep.2023

Acknowledgement:

This report has been adapted from the SMM695 final group project and refined to highlight my individual contributions. The link to the original project can be found below:

https://github.com/lyh1068/DBMS-SMM695

Introduction:

This report seeks to explore data based on bug records for the Mozilla project. Mozilla is a peculiar example of open-source software (OSS) development originating from Netscape, a computer services company. Mozilla has employed the Bugzilla software as a bug-tracking system since its early stages. For this project, bugs for the 1997-2003 development window will be analysed. A bug is a defect in the design, manufacture or operation of software generating undesired results or impeding operation.

Data is stored in pickle format:

folder	content	time frame	size	#bugs
archive-mozilla-bugs	bug history	1997-03-19 - 2003-08-05	2.9 GB	215,173

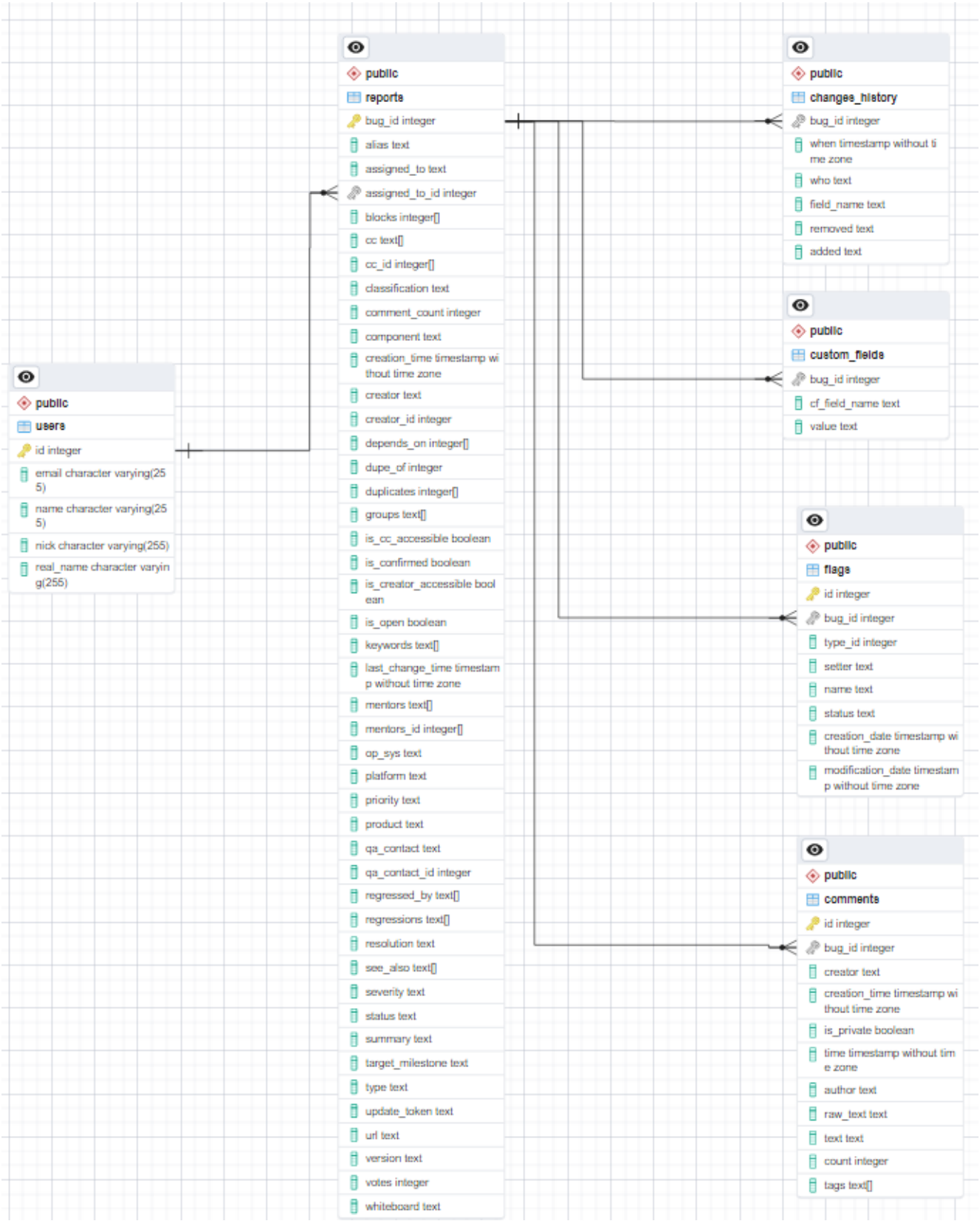
In the 1st section, the data was cleaned, manipulated and structured by Python. PostgreSQL was used to store and extract data. In particular, this part was conducted by my colleagues and is kept in this report to retain clarity. In the 2nd section, data was queried and analysed. Descriptive analysis results were concluded, and business insights were summarised based on the data visualisations.

1. Data cleaning & structuring

The bug data for the Mozilla project was initially cleaned and structured using Python and PostgreSQL. After an initial exploration of the bug dataset, it was observed that there are instances of duplicate information like some assignee information from the "assigned_to_detail" field. Additionally, certain fields contain sub-information that requires cleaning, such as bugs, comments, flags, etc.

To ensure efficient storage, retrieval, and analysis of bug data with consistent integrity, a table called "users" was established. This table contains various personal and contact information such as real names and email addresses. The "users" table accommodates a total of 67,576 user accounts originating from 50,756 distinct users, implying that some individuals possess multiple accounts using various email addresses. To avoid data redundancy and enable efficient storage, the "users" table is designed to store user details only once. Instead of duplicating user information for each bug report, the "reports" table utilizes foreign keys to reference the relevant user details from the "users" table. This approach optimizes the database structure, ensuring consistency and integrity while minimizing data duplication. As a result, bug reports can efficiently link to the corresponding user information through these foreign keys, streamlining the storage and retrieval process while facilitating effective data analysis.

Figure 1: Entity-Relationship Diagram



The “report" table is the primary table in the Entity-Relationship Diagram (ERD). It contains essential information about each bug sourced from the datasets. This table houses a comprehensive collection of bug data, encompassing 213,312 bugs contributed by 1,776 users. By consolidating key attributes such as bug ID, summary, status, priority, component, version,

target milestone, and additional details into a single cohesive structure, the "reports" table ensures the logical consistency of bug report data. This consolidation not only facilitates better comprehension but also streamlines efficient bug management. For instance, querying the "reports" table directly allows for easy retrieval of bug data with specific statuses or priorities. Furthermore, leveraging the foreign key "assigned_to_id," it becomes effortless to identify users associated with particular bugs. The design choices in the "reports" table enhance data accessibility, accuracy, and relational integrity, making bug analysis and user tracking seamless and effective.

Foreign key constraints play a crucial role in maintaining referential integrity. In the case of four other tables including changes_history, customer_fields, flags, and comments, their foreign key constraints are established to reference the primary key (bug_id) in the reports table.

- The “changes history” table keeps track of the historical modifications made to bug reports. Each entry in this table corresponds to a specific bug report, identified by the “bug_id” foreign key. It stores important details like the timestamp of the change, the user responsible for the modification, the field name that was altered, and the “added” and “removed” values. This information provides valuable insights into how the bug has evolved over time.
- The “customer_fields” table allows for the storage of additional custom fields related to specific bug reports. Each entry in this table is linked to a bug report using the “bug_id” foreign key. It includes the “cf_field_name” to denote the custom field. This table provides flexibility to include user-defined data that might not fit into the standard bug report attributes.
- The “flags” table contains information about various flags associated with bug reports. These flags serve as metadata markers and are linked to the respective bug report through the “bug_id” foreign key. The table holds details such as the type of flag, the “setter” user, the “status” of the flag, and timestamps for “creation_date” and “modification_date.”
- The “comments” table stores the comments and discussions made on specific bug reports. Each comment entry is linked to a bug report using the “bug_id” foreign key. The table includes details such as the “creator” of the comment, the “creation_time” of the comment, the “is_private” status, the “author” of the comment (if different from the creator), the raw text, and the processed “text” of the comment. Additionally, the “count” field keeps track of the total number of comments for a particular bug report, while “tags” allow for additional categorization.

This design ensures that the data contained within these tables correspond to valid bug reports. In addition, utilising these four tables facilitates sorting and organizing various details linked to a unique bug ID. For instance, the “comments” table allows for the storage of numerous discussions, updates, or user interactions pertaining to a bug, and the comments relevant to a bug report can be looked up using the bug_id foreign key. This functionality ensures that crucial context and historical data are preserved for future reference. Similarly, the changes history table documents the modifications made to a bug report over time, capturing essential information such as the modified field, the responsible user, and the old and new values. This capability enables efficient tracking and review of the bug report’s progression.

As the bug data grows over time, the separate tables can handle an increasing number of records efficiently. Additionally, the design accommodates future changes or additions to the data structure, such as introducing new fields or relationships, without requiring major modifications to the existing schema. This design effectively prevents inconsistencies and inaccurate data associations, ultimately resulting in a more dependable and precise database.

2. Descriptive insights

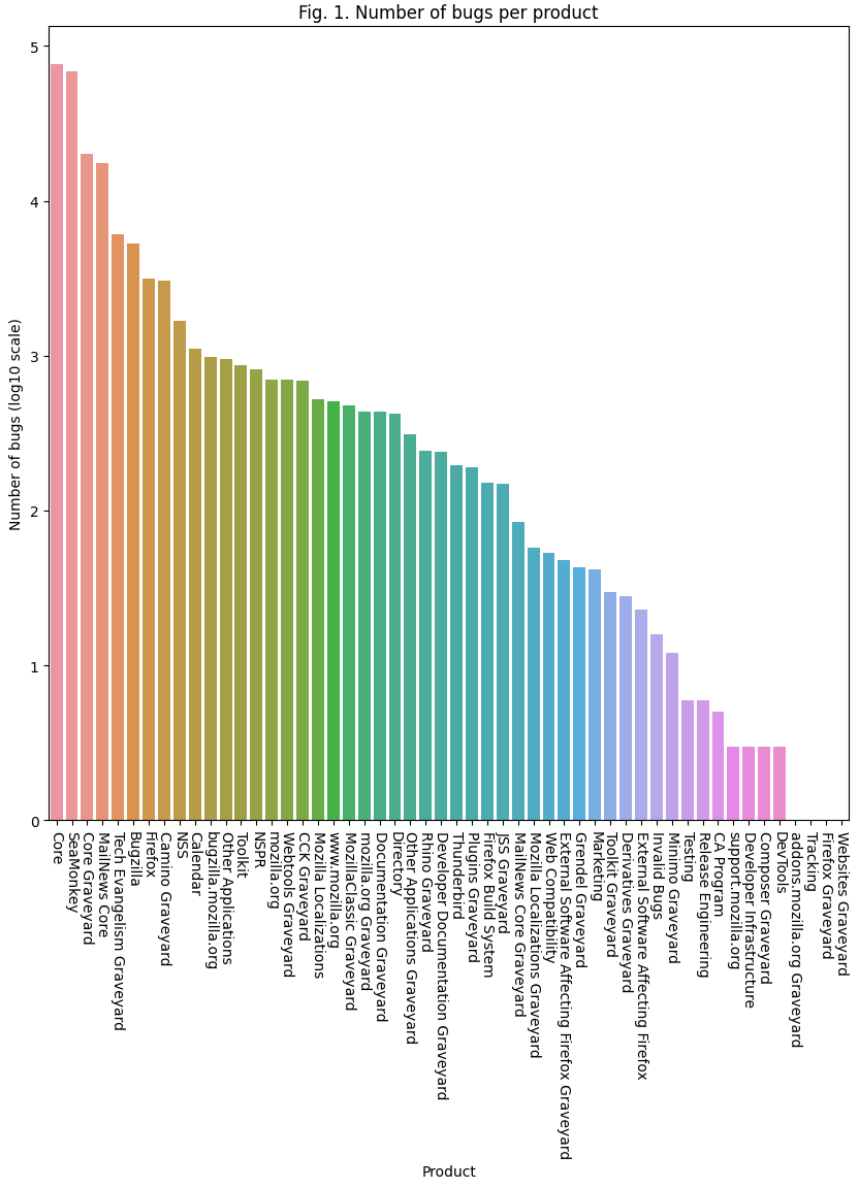
Using descriptive statistics and visualisation, we aim to analyse Bugzilla bugs to gain insights into its key metrics and enable informed decision-making for bug prioritisation, resource allocation and quality improvement.

This whole section was designed and executed as the collaborative work of my colleague Soumya Ogoti and me. In particular, 2.1-2.3 were conducted by Soumya and 2.4-2.5 were conducted by me.

2.1 Bug analysis

Total number of bugs at product level – In Fig. 1, we present the total number of bugs associated with each product, to gain insight into relative bug occurrence among various products. The highest number of bugs are for the *Core* followed by *SeaMonkey* product.

Fig. 1. Number of bugs per product

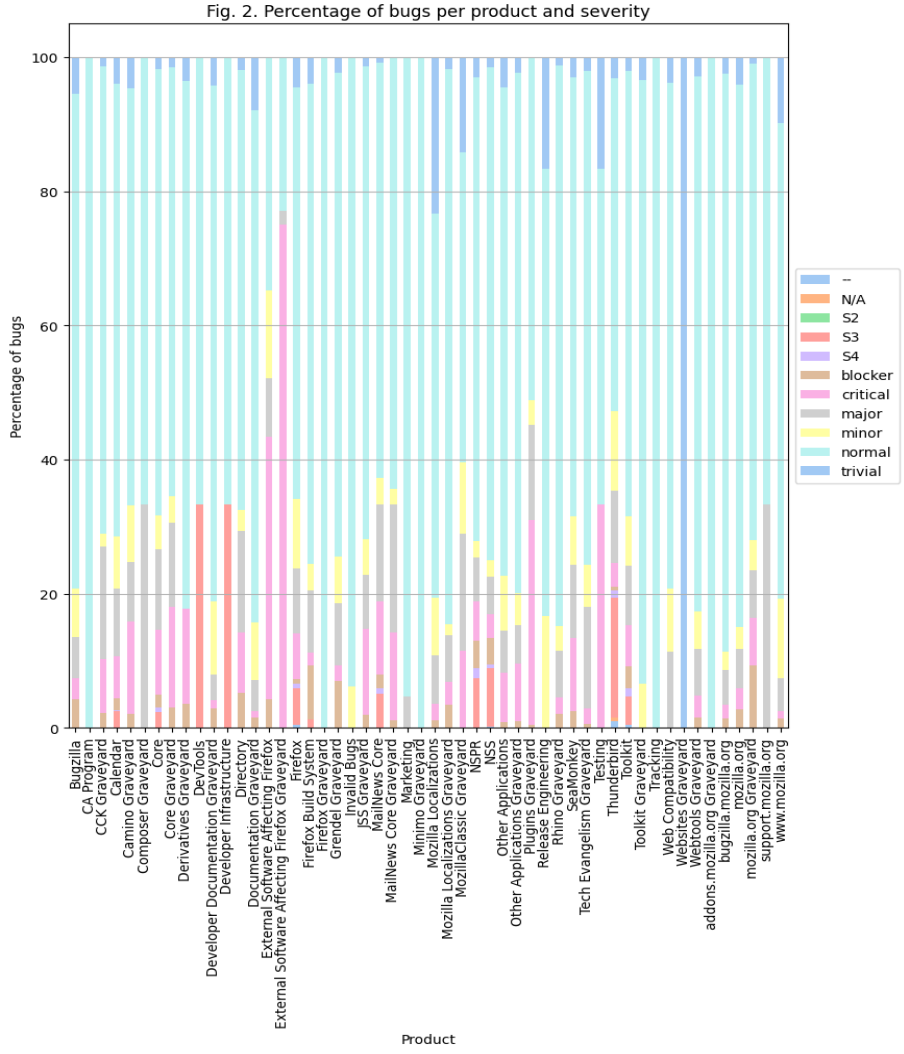


	product text	bug_count bigint
1	Core	76124
2	SeaMonkey	68769
3	Core Graveyard	20022
4	MailNews Core	17573
5	Tech Evangelism Graveyard	6073
6	Bugzilla	5282

Table 1. Top 5 products by bug count

Distribution of bugs by severity per product – We display the percentage distribution of bugs across various products (Fig. 2), with severity levels stacked to provide insights into bug composition within each product. We observe that most of the bugs are of the category *normal*. The product *Websites Graveyard* only has *trivial* bugs, and the product *tracking* only has *normal* bugs. The highest percentage of critical bugs is in the product *External Software Affecting Firefox Graveyard*.

Fig. 2. Percentage of bugs per product and severity

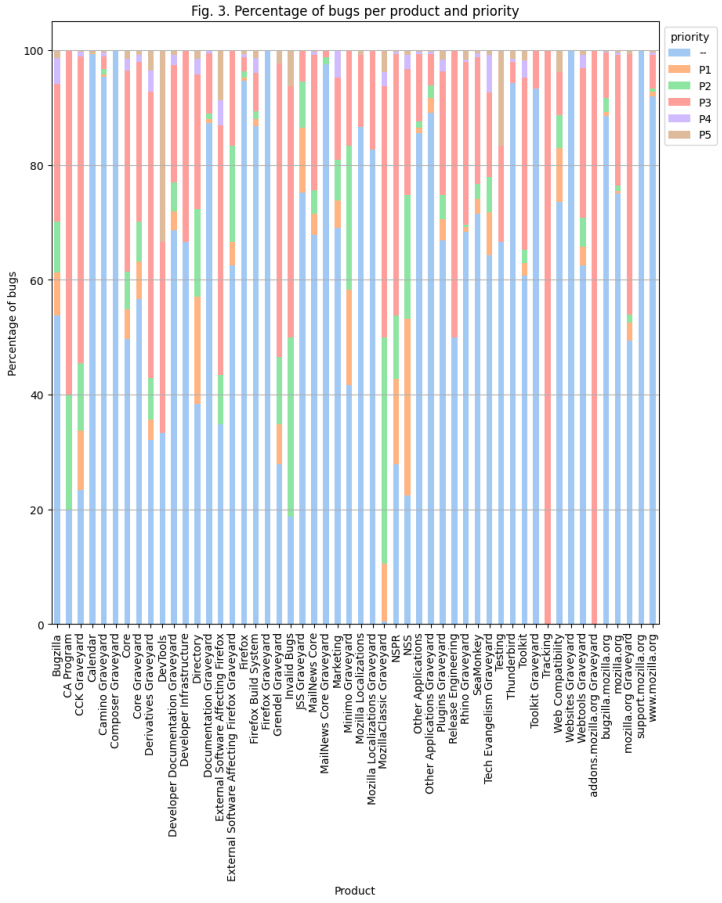


For the product *Core* which has the highest number of bugs, 66% of the bugs are *normal* followed by 12% *major* bugs.

	product text	severity text	bug_count bigint	log_bug_count double precision	percentage numeric
1	Core	normal	50685	4.704879450830577	66.5821554306132100
2	Core	major	9144	3.9611362173872253	12.0119804529451947
3	Core	critical	7388	3.868526886768204	9.7052178025327098
4	Core	minor	3850	3.5854607295085006	5.0575377016446850
5	Core	S3	1749	3.2427898094786767	2.2975671273185855
6	Core	blocker	1399	3.1458177144918276	1.8377909726236141
7	Core	trivial	1303	3.1149444157125847	1.7116809416215648
8	Core	S4	577	2.7611758131557314	0.75797383216856707477
9	Core	--	21	1.3222192947339193	0.02758656928169828175
10	Core	N/A	8	0.9030899869919435	0.01050916925017077400

Table 2. Distribution of bugs per severity level for the product *Core*

Distribution of bugs by priority per product – In a similar way, we visualise the bugs by priority per product in Fig. 3 and observe that most of the bugs are of priority level *P3*.



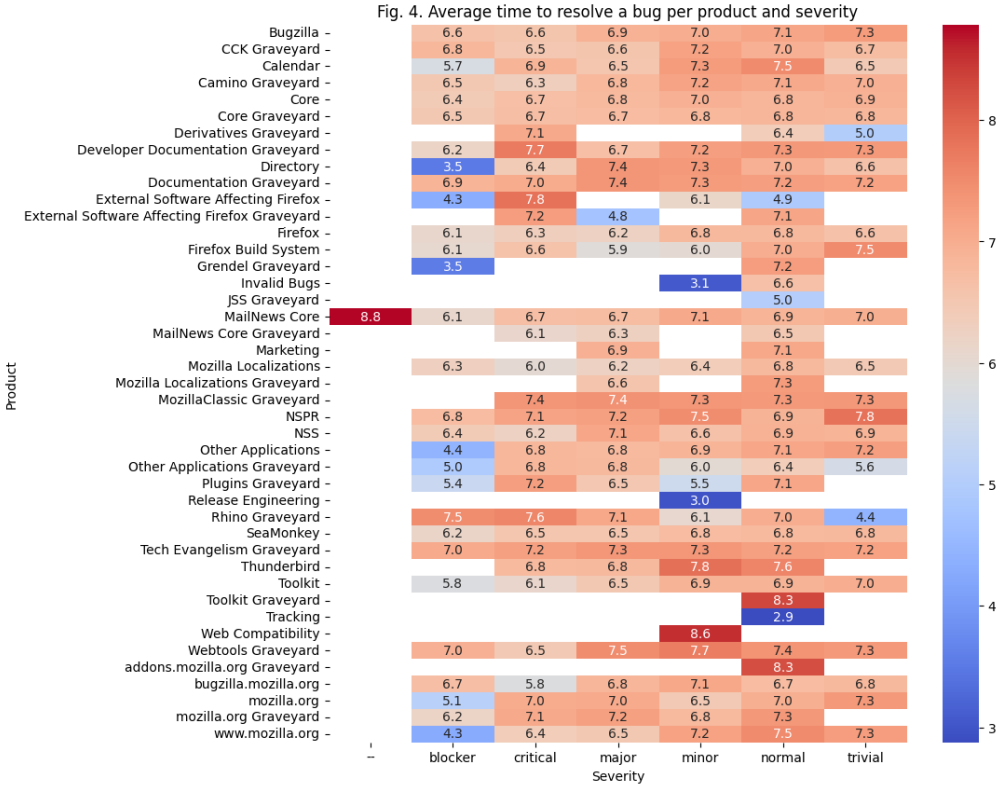
For the product *Core*, 49.76% of the bugs are *in '- -' (no decision)* followed by 34.98% *P3* bugs.

	product text	priority text	bug_count bigint	log_bug_count double precision	percentage numeric
1	Core	--	37880	4.5784099703312355	49.7609163995586149
2	Core	P3	26632	4.425403782148611	34.9850244338185066
3	Core	P2	5041	3.7025166974381505	6.6220902737638590
4	Core	P1	3892	3.5901728315963144	5.1127108402080816
5	Core	P4	1534	3.185825359612962	2.0151332037202459
6	Core	P5	1145	3.0588054866759067	1.5041248489306920

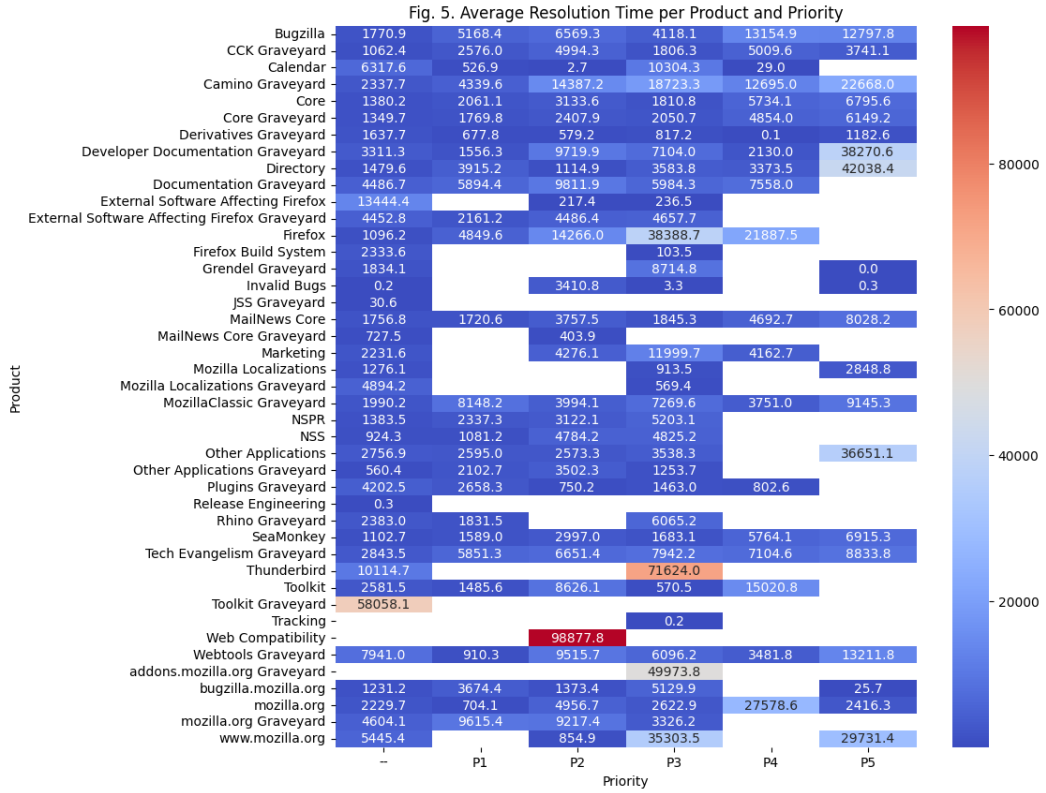
Table 3. Distribution of bugs per priority level for the product *Core*

Average time of resolution of bugs per severity level for each product – The heatmap below highlights the variations in resolution times per severity level for products to identify areas of improvement in the resolution process. The dark red shades represent longer resolution times (the highest is for *MailNew Core* – '- - 'severity level (default value for new bugs) while the bluer shades show shorter resolution periods (the lowest is for *Tracking*).

Fig. 4. Average time to resolve a bug per product and severity



Priority and average resolution time – In Fig. 5, a heatmap highlighting the variations in resolution times per priority level for products is shown, to identify areas of improvement in the resolution process. The highest resolution times (darker red) are for *Web compatibility – P2* priority and the lowest (darker blue) is for *Grendel graveyard – P5* priority.



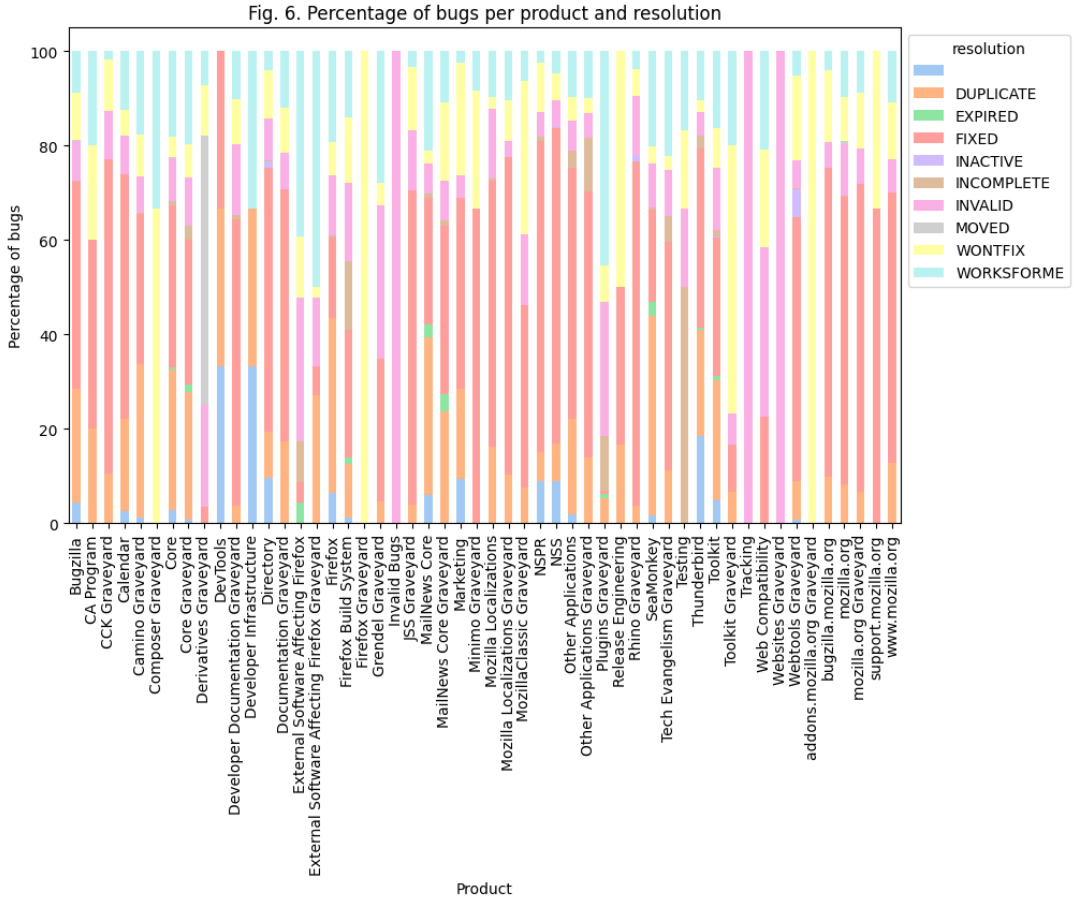
The table below shows the average resolution time in hours for each priority level for the product *Core*. We see that the highest average resolution time is for the *P5* category.

	product text	priority text	avg_res_time_hr numeric
1	Core	--	1380.23471577005846768958
2	Core	P1	2061.12162586246620054142
3	Core	P2	3133.55188338713279234390
4	Core	P3	1810.78927503618155136907
5	Core	P4	5734.12124317891559775831
6	Core	P5	6795.58952697262481674884

Table 4. Resolution times for the product *Core* by priority

Distribution of bugs by resolution per product – Here, we show the percentage of bugs for each resolution status of all products for the bugs that have been resolved (either verified or closed by a QA) to show how being are being resolved. It shows that the resolution status for most of the bugs is Fixed and we see quite a few bugs that are either in the status *Worksforme* or of the status *Wontfix*.

Fig. 6. Percentage of bugs per product and resolution



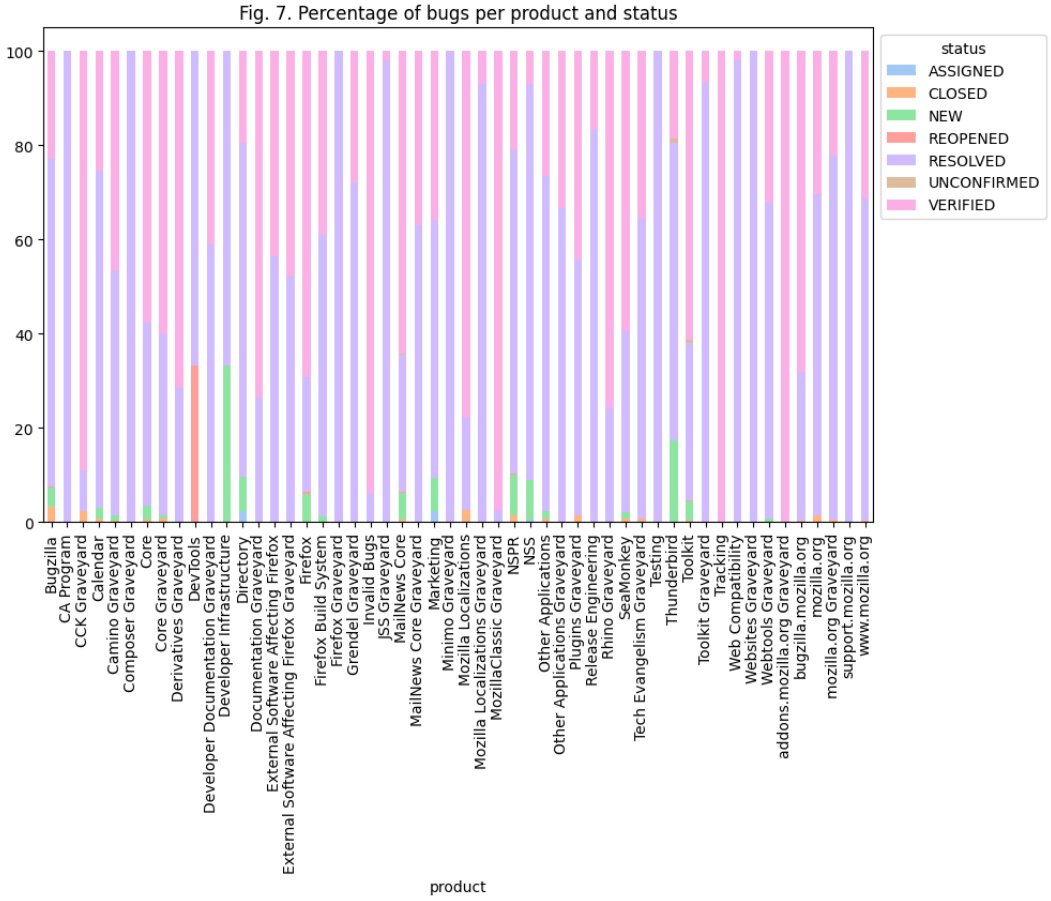
The bug counts for each resolution status level for the product *Core* show that 34% of the bugs have been *Fixed* and an almost equivalent number, 29% of the bugs are *Duplicate* bugs.

	product text	resolution text	bug_count bigint	log_bug_count double precision	percentage numeric
1	Core		2224	3.34713478291002	2.9215490515474752
2	Core	DUPLICATE	22586	4.353839323804543	29.6700120855446377
3	Core	EXPIRED	390	2.591064607026499	0.51232200094582523252
4	Core	FIXED	26007	4.415090257671115	34.1639955861489149
5	Core	INACTIVE	109	2.037426497940624	0.14318743103357679575
6	Core	INCOMPLETE	643	2.808210972924222	0.84467447848247596028
7	Core	INVALID	7031	3.847017097935354	9.2362461247438390
8	Core	MOVED	8	0.9030899869919435	0.01050916925017077400
9	Core	WONTFIX	3437	3.536179532137225	4.5150018391046188
10	Core	WORKSFORME	13689	4.136371723492323	17.9825022331984657

Table 5. Resolution statuses of bugs for the product *Core*

Distribution of bugs by status per product – Below is the visualisation to show the percentage of bugs in each status category to understand the distribution for effective resolution efforts. It shows that most of them are in the status Resolved or Verfied. The higher percentage of *Resolved* but not *Verified* bugs indicates that there are quite a few bugs that are pending QA verification.

Fig. 7. Percentage of bugs per product and status



The bug counts for each status level for the product *Core* also follow a similar trend. 57% of the bugs have been *Verified* by QAs and 38.8% have been resolved which is a good resolution rate.

	product text	status text	bug_count bigint	log_bug_count double precision	percentage numeric
1	Core	ASSIGNED	7	0.8450980400142568	0.00919552309389942725
2	Core	CLOSED	498	2.6972293427597176	0.65419578582313068152
3	Core	NEW	2117	3.325720858019412	2.7809889128264411
4	Core	REOPENED	76	1.8808135922807914	0.09983710787662235300
5	Core	RESOLVED	29575	4.470924753299968	38.8510850717250801
6	Core	UNCONFIRMED	24	1.380211241711606	0.03152750775051232200
7	Core	VERIFIED	43827	4.641741743799373	57.5731700909043140

Table 6. Statuses of bugs for the product category *Core*

2.2 Comment analysis

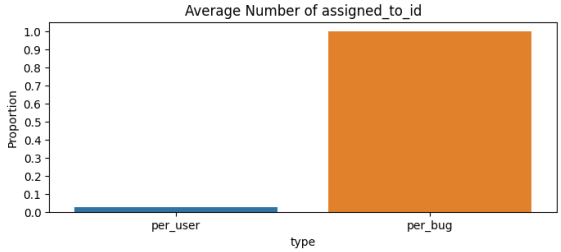
2.3 Dependency Analysis:

2.4 User Analysis

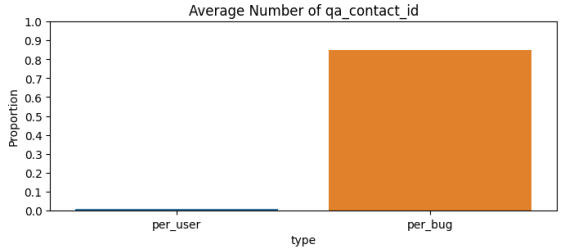
Bugs are allocated to different users (solver, QA, etc.) once being reported. The User Analysis section executes an in-depth analysis regarding the relationships between bugs and different types of users.

User roles and related bugs – According to the official document from Bugzilla, each bug will only have one developer in charge (defined by "assigned_to_id"), one QA (defined by "qa_contact_id") and multiple CC (defined by "cc_id). Each plot in Figure 12 below shows the proportions of a specific type of users who are assigned with bugs (among all users), and the proportion of bugs that have been assigned to a correspondent type of users. According to the first 2 charts, it can be implied that each bug will have one developer in charge to solve this bug and 85% of bugs have one QA, but the solvers only make up of a minor group of users among the whole population (less than 3%). The last plot indicates that approximately 70% of users will receive a CC from at least one bug report and around 68% of bugs would be CC to at least one user.

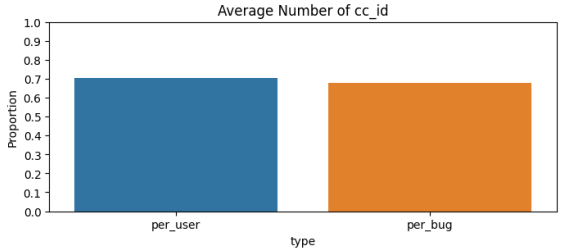
Fig. 12. User roles and bugs



	per_user numeric	per_bug numeric
1	0.026	1.000



	per_user numeric	per_bug numeric
1	0.008	0.848



	per_user numeric	per_bug numeric
1	0.705	0.679

The ratio of developer in charge to bugs, by category

This ratio reveals how many solvers are working on a same certain type of bug. The value is between 0 to 1. If the ratio is close to 1, it's indicated that for each bug within this category, almost one unique solver would be assigned with one bug. On the contrary, if the value is close

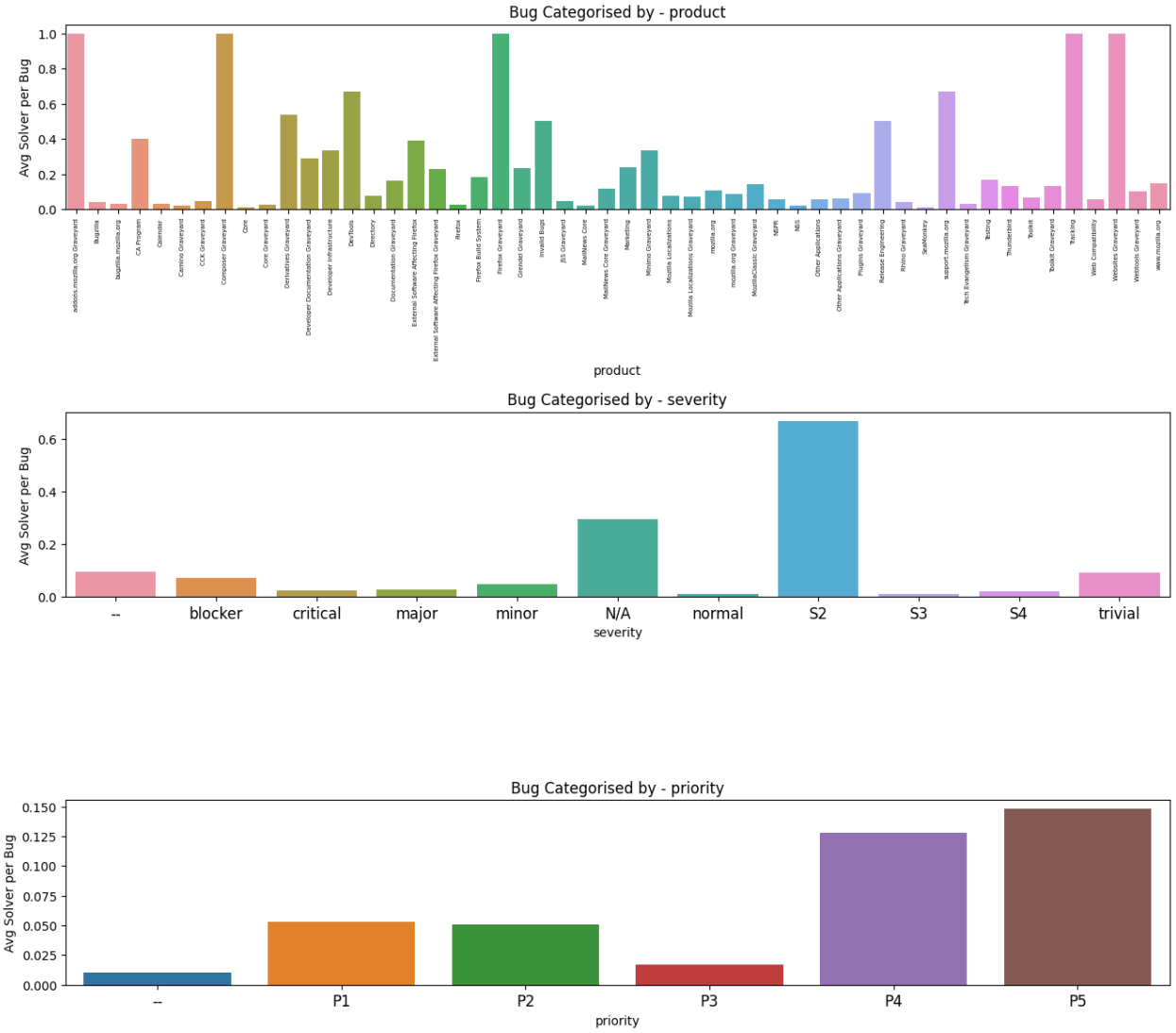
to 0, it's inferred that there are relatively fewer number of solvers are working on these bugs (one solver can be assigned with many bugs from the same category)

The relationship between developer in charge (assignee) and bug is a 1-to-many relationship. In the first chart of Figure 13, a few types of products are solved 1-to-1 by developers, but the average number of developers per product is around 0.246 and the median is 0.113.

In the second graph, two sets of severity scales can be recognised. There are relatively more solvers working on bugs of “S2” and “N/A” within the Firefox severity system (Appendix 2), and more “blocker” and “trivial” in the default system. Considering that “S2” and “blocker” are the highest severity level, it can be implied that bugs with higher severity may have more developers working on them respectively, instead of being assigned to only a few same group of solvers. However, according to Figure 2 there are very few bugs of severity “S2”, “blocker”, therefore the alternative interpretation is that with the fewer number of these types of bugs, relatively more solvers would be able to work on them.

In the third graph, “P4” and “P5” are the priority levels that have more developers on average working on, but the ratios are overall low within a narrow range from 0.01 to 0.15. According to Figure 3, there are fewer bugs of priority “P4” and “P5”, which indicates that the lower number of bugs may result in the higher ratios. Nevertheless, the overall ratios of all priorities are close, implying that bugs with all priorities are equally assigned to a relatively same scale of solvers. To further prove the insights summarised above, ratios should be evened out on the basis of the number of bugs.

Fig. 13. Average number of solvers per bug by categories



2.5 Timeseries Analysis

Analysis regarding datetime is implemented within this section. The time intervals between each status of a bug are analysed.

Temporal analysis of status changes (intervals)

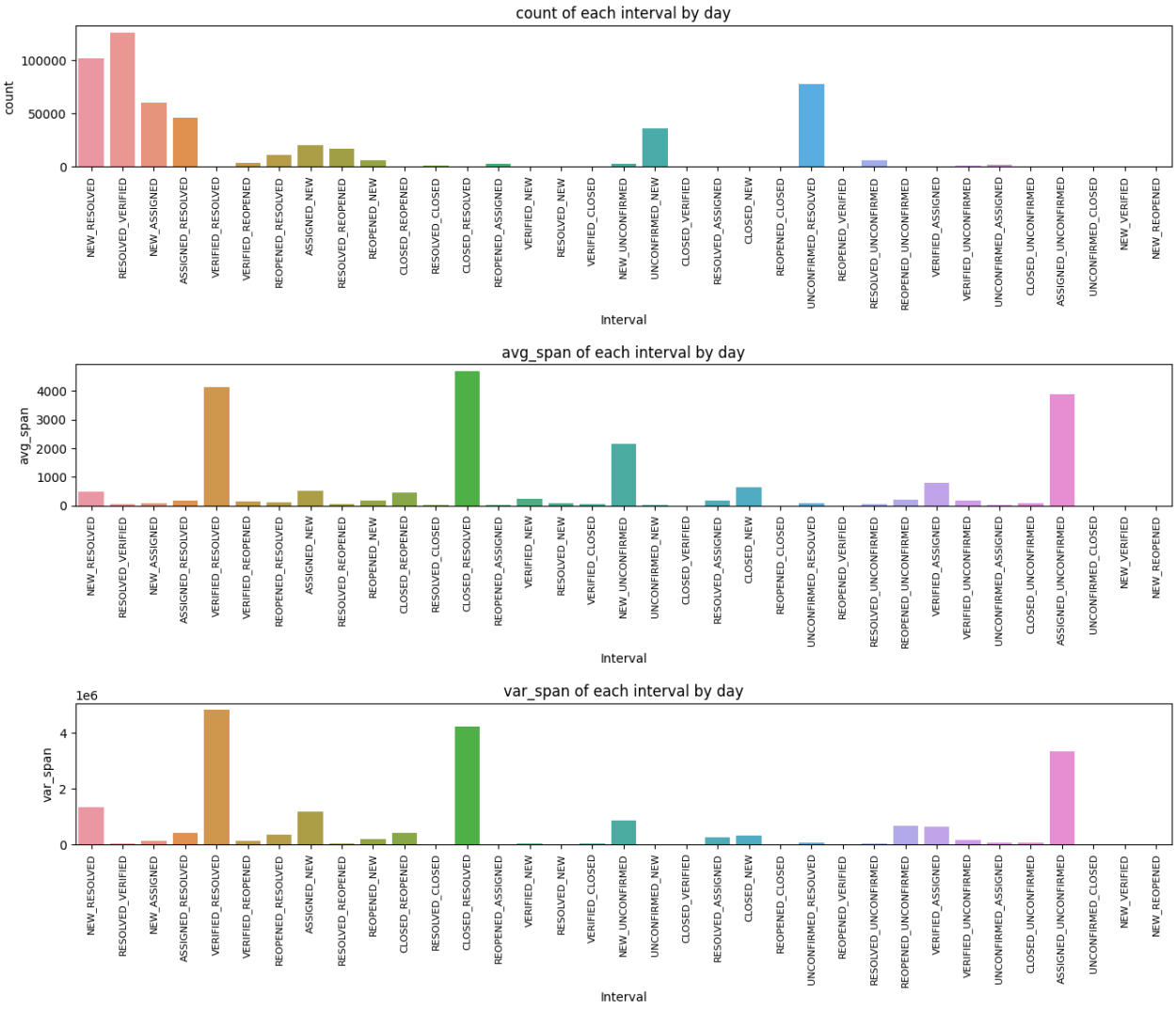
In Figure 14, the first bar chart indicates the count of different intervals. The average interval is around 14,974 days and only those major procedures of solving a bug have higher statistics. The most frequently recorded intervals are “UNCONFIRMED_ RESOLVED”, “NEW_RESOLVED” and “RESOLVED_VERIFIED” which indicates that most bugs were resolved directly without going through the resolution process. By contrast, there are around 25 types of intervals that could not be found in the official document which are less frequently recorded. It cannot be inferred whether these bugs were recorded by mistake or they were just not recorded by the official document (Appendix 3).

The second graph shows the average span by days of each interval. The average days of all intervals is 566, and the median is 83, which indicates that the data may be influenced by

extreme values or outliers and is skewed distributed. According to the graph, “VERIFIED_RESOLVED”, “CLOSED_RESOLVED” and “ASSIGNED_UNCONFIRMED” are the extreme interval types, whose average spans are higher than 4,000 days. Comparing the 2nd graph to the 1st graph, all anomaly types only have a few records and are not among the major resolving process according to the official document. The explanation to the extreme intervals is that only these few types of bugs exist in the system for a long time. These bugs can be hard to solved or harmless to the system so were left in the system.

The third graph supports the previous argument, where the intervals that have extremely high variance are those types that have high mean values. The high variance indicates that the mean values of certain types of intervals may be influenced by a few extreme values among a smaller amount of data. The variance of “NEW_RESOLVED” is relatively higher than other common intervals, implying that this process might be diversified and less predictable compared to other common intervals.

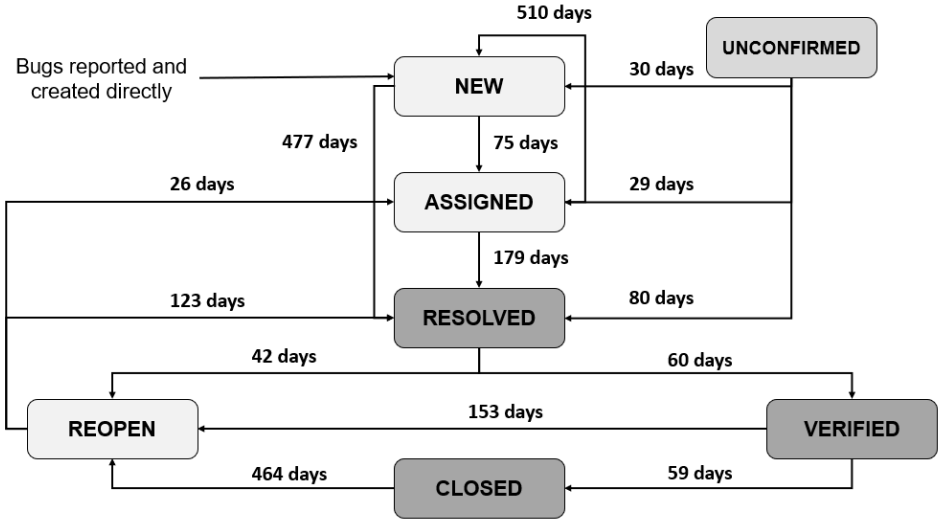
Fig. 14. Count, average interval, variance of intervals (days) by status type



Average time between status changes

The average spans (by day) of the most regular statuses are visualised below in the form of a flow chart (Figure 15).

Fig. 15. The average time between status changes



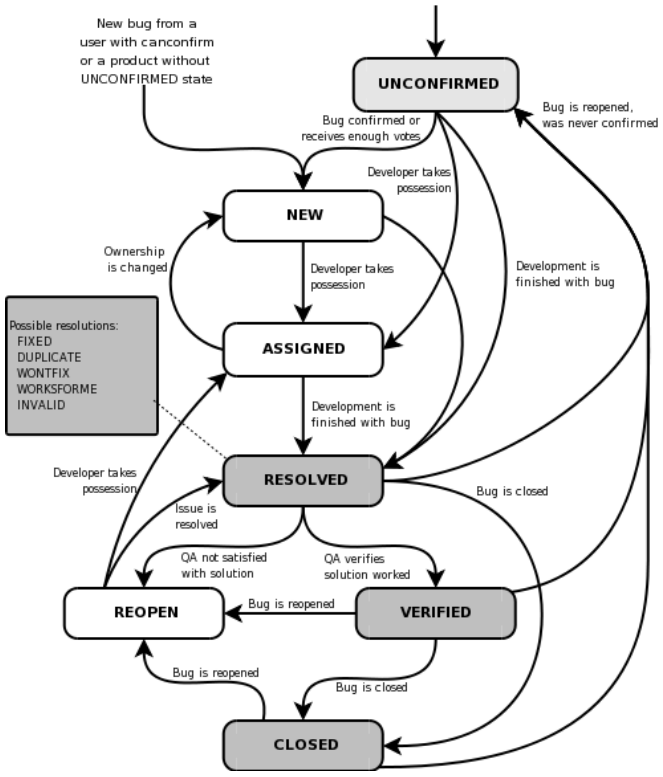
Appendix

Priority	Description
--	No decision
P1	Fix in the current release cycle
P2	Fix in the next release cycle or the following (nightly + 1 or nightly + 2)
P3	Backlog
P4	Do not use. This priority is for the Web Platform Test bot.
P5	Will not fix, but will accept a patch

Appendix 1. Description for Priority levels in bug reports (MozillaWiki, 2022)

Severity	Description
--	Default value for new bugs; bug triagers for components (ie engineers and other core project folks) are expected to update the bug's severity from this value. To avoid them missing new bugs for triage, do not alter this default when filing bugs.
S1	(Catastrophic) Blocks development/testing, may impact more than 25% of users, causes data loss, likely dot release driver, and no workaround available
S2	(Serious) Major functionality/product severely impaired or a high impact issue and a satisfactory workaround does not exist
S3	(Normal) Blocks non-critical functionality and a work around exists
S4	(Small/Trivial) minor significance, cosmetic issues, low or no impact to users
N/A	(Not Applicable) The above definitions do not apply to this bug; this value is reserved for bugs of type Task or Enhancement

Appendix 2. Description for Severity levels in bug reports (MozillaWiki, 2022)



Appendix 3. The life cycle of a Bugzilla bug (Lauhakangas, 2023)

References

Lauhakangas, I. (2023) *QA/bugzilla/fields/status*, *QA/Bugzilla/Fields/Status - The Document Foundation Wiki*. Available at: https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Status (Accessed: 19 July 2023).

MozillaWiki (2022) *BMO/UserGuide/Bugfields*, *BMO/UserGuide/BugFields - MozillaWiki*. Available at: https://wiki.mozilla.org/BMO/UserGuide/BugFields#bug_severity (Accessed: 19 July 2023).