Predictability of Bug Resolution Time Using Machine Learning

Eduardo Duarte Bassani
Department of Computer Science
University of Victoria
Victoria BC Canada
eduardodbassani@gmail.com

Manraj Minhas
Department of Computer Science
University of Victoria
Victoria BC Canada
manraj.s.minhas@gmail.com

Julia Ishibashi
Department of Computer Science
University of Victoria
Victoria BC Canada
juliaishibashi@uvic.ca

ABSTRACT

Software bugs are inevitable in software development, and predicting their resolution time is critical for effective planning and resource management. This study investigates the feasibility of using machine learning models to predict bug resolution time based on structured issue tracker data. Using bug reports from three large Apache projects (Kafka, Cassandra, and Flink), we evaluate three prediction strategies: within-project modeling, cross-project generalization, and a hybrid transfer approach that combines local and external data. Our results show that while models such as XGBoost consistently outperform a simple median baseline, the improvements are modest, and prediction errors remain large compared to typical bug lifetimes. Among the strategies, the hybrid approach achieved the best performance, though still with limited practical utility. Analysis of feature importance highlights watch count and project maturity as consistently strong predictors across projects. Overall, our findings suggest that structured metadata provides useful signals but is insufficient for accurate forecasting. Future work should incorporate richer features, such as natural language from bug reports and social or organizational factors, to improve predictive power.

CCS CONCEPTS

Computing methodologies: Machine learning approaches

KEYWORDS

Bug resolution time, Software bugs, Machine learning, XGBoost

1 Introduction and Motivation

Software bugs are an unavoidable part of software development. While finding and fixing them is crucial, a key challenge for project managers and development teams is accurately estimating how long a bug will take to resolve. This metric, known as bug resolution time, is critical for effective project planning, resource allocation, and managing stakeholder expectations. Knowing whether a bug will take hours, days, or weeks to fix helps teams prioritize tasks, schedule releases, and communicate timelines more reliably.

The most common approach to this problem is to build a predictive model using a single project's own historical bug reports. The first step in our research, therefore, is to establish a baseline by evaluating the effectiveness of this within-project prediction strategy. This addresses our first research question (RQ1) and helps us understand how well a typical project can predict its own future outcomes.

However, this standard approach has limitations, especially for new projects that lack a rich history of bug reports. This raises our second question (RQ2): can we build a general model trained on data from multiple other projects to make predictions for a new project it has never seen before? Building on this, we then explore a hybrid approach (RQ3) to see if we can get the best of both worlds by augmenting a project's local data with data from other projects.

Finally, our final research question (RQ4) aims to identify which bug report features (like severity or priority) are the most powerful and consistent predictors of resolution time across different software projects. By addressing these four questions, we hope to provide a comprehensive picture of how to best predict bug resolution times.

2 Research Questions

To guide our investigation, we have defined the following four research questions (RQs):

- RQ1 - Within-project prediction: How accurately can we predict the resolution time of future bugs in a project using only that project's own historical bug reports?
- RQ2 - Cross-project generalization: To what extent can a model trained on historical bug reports from other projects predict the resolution time of bugs in a completely new, unseen project?
- RQ3 - Hybrid transfer: Does augmenting a project's local training data with bug reports from other projects improve its future-bug prediction accuracy compared with using local data alone?
- RQ4 - Feature consistency: Which features are consistently most predictive of bug resolution time across projects, and do their importances differ between projects?

3 Background and Related Work

The challenge of understanding and predicting how long it takes to fix bugs is well studied in software engineering. Early research by Zhang et al. applied simple regression to Eclipse bug data and

showed that factors like component, priority, and code churn strongly affect fix time [1]. Özkan et al. added a wide range of static code metrics (for example, cyclomatic complexity and code size) in their "Bug Analysis Towards Bug Resolution Time Prediction," and found these metrics predict resolution time just as well across network-softwarization projects [2]. Adekunle et al. then looked at machine-learning libraries on a large scale, examining report details such as reporter, component, and type, and discovered that those metadata alone account for much of the differences in how long bugs stay open [3].

Having rich, curated datasets has been key to these modeling efforts. Acharya and Ginde's BugsRepo provides over 119,000 fixed Bugzilla reports complete with comments, contributors, and timestamps, making it easy to reproduce studies of bug resolution workflows [4]. More recently, Patil's GitBugs dataset gathered detailed reports (including "created" and "resolved" timestamps) from nine major open-source projects, offering a modern, varied corpus for finding new predictors [5].

Beyond metadata, the text of bug reports has proven to be a powerful signal. Nastos et al. introduced an interpretable pipeline that combines topic modeling of descriptions with structured fields (priority, labels, and past assignee history) to both predict and explain resolution times [6]. Building on that, Chen et al. showed that using LLMs to break bug reports into sub-issues and action items surfaces features that correlate closely with fix latency [7].

Other parts of the bug lifecycle have been explored too. Li et al. demonstrated that changing a bug's priority after it's filed does more than update a label—it speeds up or slows down fixes across 32 Apache projects [8]. And Zhang et al.'s cross-language study found that different programming-language ecosystems have distinct median resolution times and patch sizes, highlighting the need to adjust models when moving between languages [9].

4 Data

This section details the dataset used for our analysis, covering its origin, structure, and the preparation steps required for modeling. We begin by describing the data source, the public JIRA issue trackers of three prominent Apache projects. We then outline the cleaning and feature engineering process that was necessary to transform the raw bug reports into the structured, analysis-ready datasets used in our experiments.

4.1 Data Sources

Our study uses bug report data sourced from the public JIRA issue tracker of the Apache Software Foundation (ASF). The ASF is an ideal source because it hosts hundreds of mature, widely used open-source projects, providing realistic and high-quality software engineering data that ensures the transparency and reproducibility of our work. For our analysis, we selected three prominent ASF projects known for being large, complex, and data-centric systems: Apache Kafka, a distributed event streaming platform; Apache Cassandra, a distributed NoSQL database system; and Apache Flink, a stream processing framework and batch processing system. Each bug report from these projects contains a rich set of fields that characterize the bug.

4.2 Data Description

The datasets were retrieved using the JIRA REST API. We specifically queried for issues of type "Bug" with a "Resolved" or "Closed" status, as these statuses contain the necessary resolution date information. The raw JSON data obtained from the API was loaded and flattened into data frames using the from JSON ("", flatten = TRUE) function. This process allowed for a preliminary analysis of the available columns and rows within each dataset.

The number of bug reports (records) for each project is as follows: Cassandra with 10,009 records, Flink with 12,119 records, and Kafka with 6,356 records. The initial flattening process yielded a substantial number of columns for each dataset, with 200 for Cassandra, 132 for Flink, and 165 for Kafka. It's important to note that some of these columns contained lists of objects, which were not fully expanded during the initial flattening. Consequently, the reported column counts do not represent every possible nested field within the raw data.

4.3 Feature Overview

Across all three projects, the fields presented in Table 1 are present in every dataset and look most promising as predictors of resolution_time (our target variable):

Field	Why it is useful	Data Type
fields.created	Timestamp when the bug was opened (used for calculating resolution_time).	character
fields.resolutiondate	Timestamp when the bug was marked resolved/closed (used for calculating resolution_time).	list
fields.priority.name	Higher-priority bugs often get fixed faster.	character
fields.reporter.name	Reporters may file easier-to-reproduce bugs the more often they do it (experience signal).	character

fields.comment.comments	More back-and-forth often means more triage time.	list
fields.attachment	Patches/logs attached may speed or slow diagnosis.	list
fields.watches.watchCount	Bugs with more watchers may be addressed more quickly.	integer
fields.votes.votes	Highly voted bugs may receive more attention.	integers
fields.description	Longer descriptions usually mean more context/detail. Presence of code snippets or embedded images can speed triage.	list

Table 1: Most Promising Fields as Predictors

4.4 Data Cleaning and Preparation

The raw data we collected from JIRA was in a complex JSON format and not immediately suitable for exploratory data analysis and modeling. This section describes the crucial steps we took to clean the data and perform feature engineering, the process of creating new variables from the raw information. This entire process was automated using an R script to ensure it was consistent and reproducible for all four projects.

First, we performed several cleaning operations. We started by filtering out any bug reports that were missing a creation or resolution date, as these are essential for calculating our target variable. We then removed any bugs that had missing values in key predictor fields. This ensures that our models are trained on complete data.

Next, we engineered a set of new features designed to capture different aspects of a bug report that might influence its resolution time. These features capture the bug's context, the community's engagement with it, and the reporter's experience. Table 2 details each newly created feature, its purpose, and its data type.

Feature	Description & Rationale	Data Type
resolution_time	The target variable. The total time in days from bug creation to resolution.	Double
watch_count, vote_count	The number of users watching or voting for a bug. A higher count may signal higher impact, potentially leading to faster resolution.	Integer
num_comments, num_attachments	The count of comments and attachments. High numbers can indicate a complex, ongoing discussion or a well-documented issue.	Integer
description_length	The number of characters in the bug's description. This can act as a simple proxy for the bug's initial complexity.	Integer
created_wday, created_month	The day of the week and month the bug was created. This helps capture weekly or seasonal patterns (e.g., weekend vs. weekday).	Integer
days_since_project_start	The number of days from the project's very first bug report. This captures the project's maturity at the time of the bug.	Integer
bugs_last_7d, bugs_last_30d	The number of bugs created in the 7- or 30-days prior. This measures the recent workload, which could strain resources.	Integer

has_code_block	A flag indicating if the description contains a formatted code block. A reproducible example could speed up diagnosis and resolution.	logical
has_inline_attachment	A flag indicating if the description includes an inline image or file. Visual aids might help clarify the issue.	logical
reporter_experience	The number of bugs a user had previously reported in the project. An experienced reporter may submit higher-quality reports.	Integer

Table 2: Set of Engineered Features

In addition to these engineered features, we also included the bug's original priority level in our final dataset. This categorical feature was carried over directly from the raw data because it represents a developer's explicit judgment of a bug's urgency and is expected to be a strong predictor of resolution time. In total, our final cleaned dataset contains 15 columns: our target variable (resolution_time) and 14 predictor features.

After this cleaning and feature engineering process, we were left with three cleaned datasets. The final number of cleaned bug reports for each project is as follows: Cassandra with 10,007 records, Flink with 12,118 records, and Kafka with 6,332 records. Each dataset was saved as an .rds file to enable efficient loading and reuse in later exploratory and modeling stages.

5 Exploratory Data Analysis

This section presents the exploratory data analysis (EDA) conducted to understand the characteristics of the cleaned bug report data before modeling. The primary goals of this analysis are to investigate the distribution of our target variable, explore the relationships between the predictor features and the resolution time, and identify any interactions between the predictors themselves. The insights gained from this process are essential for informing our final feature selection and overall modeling strategy.

5.1 Univariate Analysis: Target Variable

Our first step in the exploratory data analysis was to understand the distribution of our target variable, resolution_time. The initial analysis revealed that the data is highly right-skewed, which is common for time-based metrics.

Table 3 below compares the summary statistics of the resolution time before and after applying a log transformation. In the original data, the mean (109.1 days) is over 13 times larger than the median (8.1 days). This large gap confirms that a small number of outlier bugs, which take an exceptionally long time to fix, are significantly inflating the average. After the transformation, the mean (2.60) and median (2.21) are much closer, indicating a more symmetric distribution.

Statistic	resolution_time	log_resolution_time
Min.	0	0
1st Qu.	1.1	0.74
Median	8.14	2.21
Mean	109.15	2.6
3rd Qu.	58.27	4.08
Max.	4233.28	8.35

Table 3: Summary Statistics of the Resolution Time

This skew is clearly visible in the histogram below (Figure 1), which shows a massive concentration of bugs resolved quickly on the left and a long, flat tail extending to the right.

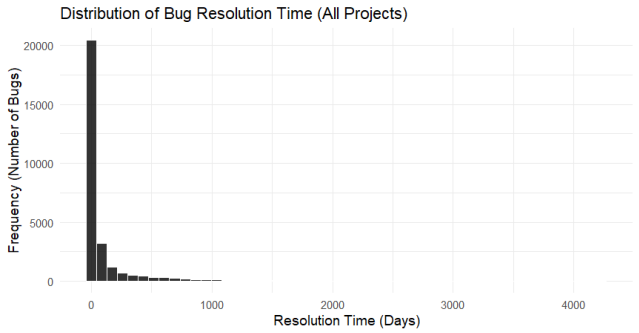

Figure 1: Distribution of Original Bug Resolution Time

A heavily skewed target variable can impair the performance of predictive models. To address this, we applied a logarithmic transformation by calculating log (resolution_time + 1). This technique compresses the range of the data, pulling in the extreme values. The resulting distribution, shown in Figure 2, is much more balanced and suitable for modeling.

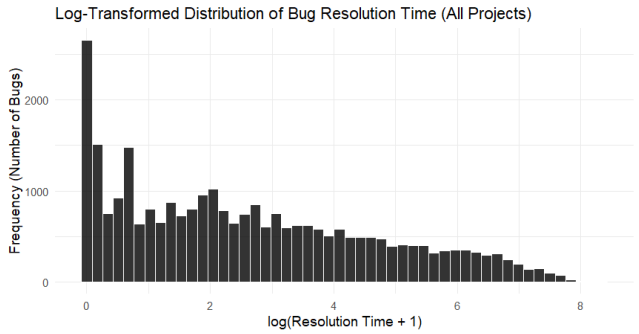

Figure 2: Distribution of Log-Transformed Bug Resolution Time

This transformed variable, log_resolution_time, will serve as the target for our predictive models going forward.

5.2 Bivariate Analysis: Predictors vs. Resolution Time

In this section, we investigate the relationship between each predictor feature and our target variable, log_resolution_time, to identify which factors are most associated with how long it takes to resolve a bug.

5.2.1 Numerical Predictors vs. Resolution Time

To explore the relationship between the numerical features and bug resolution time, we employed a two-step approach. First, we generated scatter plots for a visual inspection of the patterns. Second, we calculated Spearman's rank correlation coefficient to numerically quantify the strength and direction of these relationships. Spearman's was chosen over the more common Pearson's correlation because it can detect monotonic (consistently increasing or decreasing) relationships, even if they are not linear.

The scatter plots for each numerical predictor against the log-transformed resolution time are shown in Figure 3.

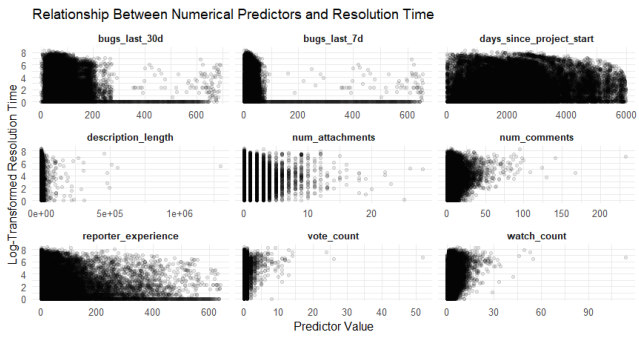

Figure 3: Scatter Plots of Numerical Predictors vs. Log-Transformed Resolution Time

From the plots, it is difficult to draw definitive conclusions by eye. In nearly every plot, the data points are heavily concentrated in the bottom-left corner, a phenomenon known as overplotting. This density of points obscures any underlying trends, making visual interpretation unreliable.

To overcome this visual limitation, we calculated the Spearman's correlation coefficients, presented in Table 4. This provides a clear numerical summary of the relationships.

Feature	Spearman Correlation
watch_count	0.42
num_comments	0.29
vote_count	0.18
description_length	0.17
days_since_project_start	0.1
num_attachments	-0.02
bugs_last_30d	-0.1
reporter_experience	-0.11
bugs_last_7d	-0.12

Table 4: Spearman's Correlation with log_resolution_time

The correlation coefficients reveal several trends. The strongest relationship is a moderate positive correlation for watch_count (0.42), indicating that bugs with more watchers tend to take longer to resolve. Weaker positive correlations for num_comments (0.29), vote_count (0.18), and description_length (0.17) suggest a similar pattern where bugs involving more discussion or detail are also associated with longer resolution times. Conversely, the strongest negative correlations are for bugs_last_7d (-0.12) and reporter_experience (-0.11), suggesting that a higher volume of recent bugs and bugs filed by more experienced reporters are both associated with slightly faster resolutions. Finally, num_attachments (-0.02) has a correlation near zero, indicating it has no meaningful relationship with bug resolution time.

5.2.2 Categorical Predictors vs. Resolution Time

For categorical features, we used box plots to visually compare the distribution of log_resolution_time across the different groups. This method allows us to easily see differences in the median and spread of resolution times for categories like bug priority or the day a bug was created.

Of all the categorical predictors, bug priority has the most significant impact on resolution time. However, as shown in Figure 4, this relationship is not straightforward and is inconsistent across the different projects.

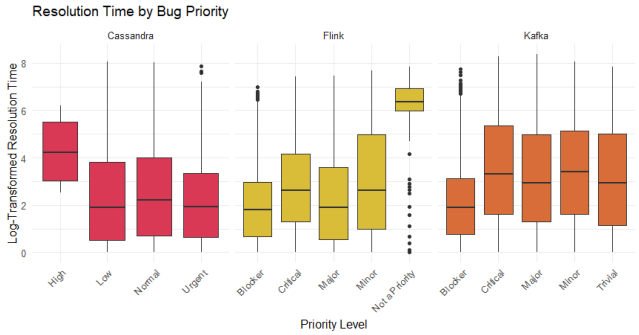

Figure 4: Resolution Time by Bug Priority

The plots reveal that while priority is a key factor, it does not have a simple linear effect where increasing priority guarantees a faster resolution. For example, Cassandra's priority levels do not follow a logical order, with "High" priority bugs taking the longest to resolve. This inconsistency, including the use of different labels across projects, highlights a major challenge for cross-project modeling.

We also investigated whether the time a bug was created had any impact on its resolution time. As seen in Figure 5, the day of the week has a weak relationship with resolution time. There is a slight tendency for bugs filed on Monday to be resolved the fastest, while those filed on Friday and Saturday take slightly longer, likely because they are heading into the weekend. However, the distributions for all seven days overlap significantly, suggesting this is not a strong predictor.

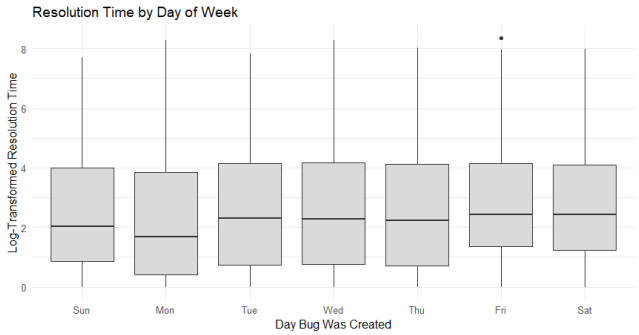

Figure 5: Resolution Time by Day of the Week

Similarly, the month a bug was created shows little predictive power (Figure 6). There are no strong seasonal trends, although there is a noticeable dip in the median resolution time for bugs created in June. Aside from this anomaly, the distributions for the other months are very similar.

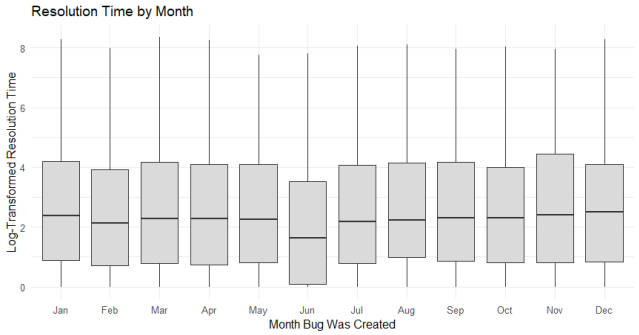

Figure 6: Resolution Time by Month

5.2.3 Logical Predictors vs. Resolution Time

Finally, we analyzed the logical predictors, has_code_block and has_inline_attachment, to see if the presence of these elements in a bug report affects resolution time. We used box plots to compare the resolution times for bugs that included these features versus those that did not.

The results show a subtle but interesting pattern. Contrary to our initial hypothesis that a code block would speed up a fix, Figure 7 shows that bugs containing a code block (TRUE) have a slightly higher median resolution time than those without one. This suggests that the presence of a code block is often an indicator of a more complex, technical bug that requires more time to diagnose and resolve.

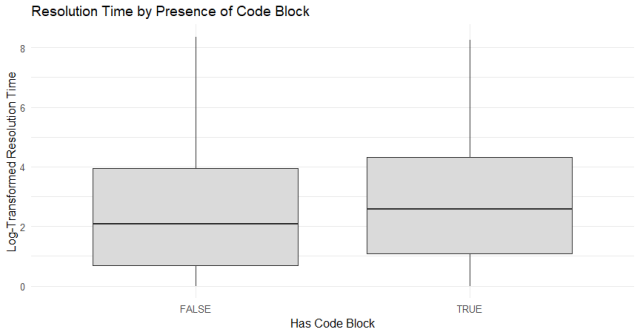

Figure 7: Resolution Time by Presence of Code Block

For the has_inline_attachment feature, Figure 8 shows there is virtually no difference in the median resolution time between bugs with and without an inline attachment. This indicates that the presence of an inline image or file is not a meaningful predictor of how long a bug will take to be fixed.

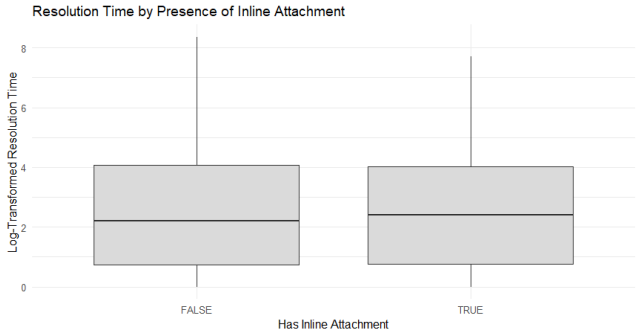

Figure 8: Resolution Time by Presence of Inline Attachment

5.3 Multivariate Analysis: Interactions Between Predictors

The final step of our exploratory analysis was to examine the relationships between the numerical predictor variables themselves. The primary goal of this multivariate analysis is to identify multicollinearity, which occurs when two or more predictors are highly correlated and carry redundant information. This can be problematic for certain predictive models.

To do this, we calculated the Spearman's correlation coefficient for every pair of numerical predictors and visualized the results as a heatmap, shown in Figure 9.

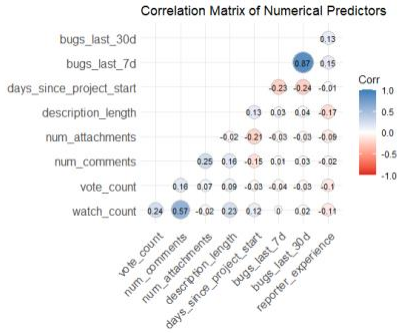

Figure 9: Correlation Matrix of Numerical Predictors

The heatmap reveals one clear case of high multicollinearity. There is a very strong positive correlation of 0.87 between bugs_last_7d and bugs_last_30d. This is expected, as these two features measure a very similar concept of recent bug activity. Because they are highly redundant, including both in a model could make it unstable. Therefore, one of these features should be removed before the modeling stage.

We also observe a moderate positive correlation of 0.57 between watch_count and num_comments, which makes sense as both relate to community engagement. The remaining predictor variables show weak to negligible correlations with each other, indicating that they capture unique information and can be safely used together in a model.

5.4 Feature Selection

Based on the insights gathered during the exploratory data analysis, we have selected a final set of features for the modeling phase. The goal was to create a simpler and more robust model by removing features that were redundant, inconsistent, or had no predictive power.

Several features were removed from the dataset. The priority feature was removed because its labels and effects were highly inconsistent across projects, making it an unreliable predictor for a generalizable model. We also removed bugs_last_30d due to its high multicollinearity with bugs_last_7d; the latter was retained because its correlation with the resolution time was slightly stronger. The num_attachments and has_inline_attachment features were removed for having no predictive value, as confirmed by their near-zero correlation and indistinct box plots, respectively. Finally, created_month was dropped as the analysis showed no clear seasonal pattern, rendering it uninformative.

The remaining features were kept for modeling. While several of these have weak individual correlations with resolution time, we decided to retain them because, in combination, they may provide enough signal to build a predictive model that is more accurate than a trivial baseline. For example, created_wday has a weak overall relationship, but the box plot revealed a subtle, logical pattern where bugs filed on Mondays are resolved slightly faster and those filed heading into the weekend take longer. This small insight, combined with others, could contribute to a more accurate model. The final set of features selected for modeling is: watch_count, num_comments, vote_count, description_length, reporter_experience, bugs_last_7d, days_since_project_start, has_code_block, and created_wday.

6 Methodology

This section provides a detailed account of the experimental methodology employed to address our research questions. Following the exploratory data analysis and feature selection, we established a robust process for preparing the data, selecting and training the predictive models, splitting the data according to each research question, and evaluating the final performance.

6.1 Model Selection

The task of predicting bug resolution time is a supervised regression problem, as we are predicting a continuous numerical value. Based on the insights from our exploratory data analysis, we selected two models for our experiments to ensure a rigorous evaluation.

First, a Median Baseline model was established. Given that the distribution of resolution_time is heavily right-skewed, the mean is a poor measure of central tendency. The median, being robust to outliers, provides a much more representative baseline. This model's function is to predict the median resolution time of the training data for every bug, regardless of its features. This serves as a "no-information" benchmark; any sophisticated model must demonstrate a significant improvement over this simple heuristic to be considered useful.

Second, a Gradient Boosting Machine, specifically the XGBoost implementation, was chosen as our primary predictive model. The EDA revealed that the relationships between many features and the target variable are complex and non-linear. XGBoost is a state-of-the-art ensemble algorithm that excels in such scenarios. It builds a sequence of decision trees, where each new tree is trained to correct the errors of the previous ones. This sequential, error-correcting process allows it to effectively capture the intricate patterns and feature interactions present in the data, making it a powerful and highly suitable choice for this problem.

6.2 Data Preprocessing for Modeling

Before the data could be used for training, a final preprocessing pipeline was necessary to convert all selected features into a purely numerical format that the XGBoost model can accept. This involved transforming the remaining non-numerical features.

The categorical created_wday feature was converted using one-hot encoding. This process transforms the single column of weekday labels into seven new binary (0 or 1) columns, one for each day of the week. This allows the model to learn the individual impact of a bug being created on a specific day without imposing a false ordinal relationship between the days (e.g., assuming Wednesday is numerically "greater" than Tuesday).

Additionally, the logical has_code_block feature, which contained TRUE and FALSE values, was converted into a binary numerical format. TRUE was mapped to 1 and FALSE was mapped to 0, allowing the model to interpret this feature as a numerical input.

6.3 Experimental Design and Data Splitting

To rigorously answer our three distinct research questions, we designed a specific data splitting strategy for each experimental scenario, using the data from the three projects: Cassandra, Flink, and Kafka.

For RQ1 (Within-Project Prediction), the goal was to simulate a real-world scenario where a project team wants to predict future outcomes based on their own historical data. To achieve this, we performed a chronological split for each of the three projects. For each project, the bug reports were ordered by their creation date, and the oldest 80% of the data was used as the training set, while the most recent 20% was reserved as the test set. This resulted in three independent experiments, one for each project.

For RQ2 (Cross-Project Generalization), we aimed to test a model's ability to generalize to a completely new project it has never seen before. To evaluate this, we used Leave-One-Project-Out Cross-Validation. This procedure involves three experimental folds. In each fold, we trained a single model on the combined data from two of the projects and tested its performance on the third, entirely held-out project. This process was repeated until each project had served as the test set once.

For RQ3 (Hybrid Transfer), we investigated whether augmenting a project's local data with data from other projects could improve its predictive accuracy. This involved another set of three experiments. For each "target" project, the training set was constructed by combining all the data from the other two projects with the oldest 80% of the target project's own data. The model was then tested on the newest 20% of the target project's data.

6.4 Evaluation Metrics

A crucial step in our evaluation process was to ensure the results are interpretable in practical, real-world terms. Since the model was trained to predict the log-transformed resolution time, all predictions were first inverse-transformed back to the original scale of days using the formula days = exp(log_prediction) - 1.

The performance of the XGBoost model and the median baseline was then measured on this original scale using two standard regression metrics. We selected Mean Absolute Error (MAE), which measures the average absolute difference between the predicted and actual resolution time in days. MAE is easily interpretable and provides a clear sense of the average prediction error. We also used Root Mean Squared Error (RMSE), which also measures the average error in days but gives a higher weight to larger errors. This is particularly useful for understanding the impact of significant mispredictions, which are common in skewed datasets.

For each of the three research questions, the final performance was calculated by averaging the MAE and RMSE scores obtained from the three independent experimental runs. This averaging provides a more robust and generalizable measure of each prediction strategy's effectiveness.

7 Findings

This section presents the results from our modeling experiments, directly addressing the four research questions. We begin by discussing the initial model's performance and the subsequent tuning process, then present the final results for each experimental scenario using our optimized XGBoost model.

7.1 Model Tuning

Our initial experiments were conducted using an untuned XGBoost model with standard default hyperparameters. The results from this model were unsatisfactory. While it showed a marginal improvement over the median baseline in terms of Root Mean Squared Error (RMSE), its Mean Absolute Error (MAE) was often worse. This indicated that the untuned model was not effectively learning from the data and failed to provide a meaningful improvement over a simple heuristic.

To address this, we undertook a methodical, incremental tuning process. We first tested a slower learning rate (eta = 0.05) with more trees, which yielded a slight improvement. Next, we

experimented with tree complexity, finding that simpler trees (max_depth = 4) performed better, suggesting the initial model was overfitting. Finally, we introduced regularization (subsample = 0.8, colsample_bytree = 0.8) to further combat overfitting. This final combination produced our best-performing model, which demonstrated an improvement over the initial, untuned version.

7.2 Within-Project Prediction (RQ1)

The first experiment tested how accurately a model can predict a project's future resolution time using only its own historical data. The results, shown in Table 5, show that the tuned XGBoost model outperforms the median baseline on both MAE and RMSE. This confirms that a project's historical data contains a learnable signal. However, the improvement is small. An average error of 19.4 days is substantial, especially considering the median resolution time for a bug is only 8.14 days. This level of error is too high for the model to be considered satisfactory for reliable planning.

Within-Project	Avg_MAE (Days)	Avg_RMSE (Days)
Median Baseline	19.8	37.7
Tuned XGBoost	19.4	34.5

Table 5: Evaluation Metrics for Within-Project Prediction

7.3 Cross-Project Generalization (RQ2)

The second experiment evaluated a model trained on data from other projects to predict on a new, unseen project. This cross-project model also managed to beat the baseline, as shown in Table 6, which indicates that some generalizable patterns exist across projects. Nevertheless, this was the weakest-performing of the three strategies, and the average error of 19.8 days remains unsatisfactorily high.

Within-Project	Avg_MAE (Days)	Avg_RMSE (Days)
Median Baseline	20.8	39.9
Tuned XGBoost	19.8	37.6

Table 6: Evaluation Metrics for Cross-Project Generalization

7.4 Hybrid Transfer (RQ3)

The third experiment tested a hybrid approach, where a project's local data was augmented with external data. This strategy yielded the best performance, as shown in Table 7, with an Avg_MAE of 19.2 days. While this confirms that a hybrid approach is the most effective strategy, the overall conclusion remains the same. Although the model learns from the data and outperforms a simple heuristic, its predictive power is limited. The final error is still too large for the model to be practically useful for generating precise estimates.

Within-Project	Avg_MAE (Days)	Avg_RMSE (Days)
Median Baseline	19.9	37.8
Tuned XGBoost	19.2	35.8

Table 7: Evaluation Metrics for Hybrid Transfer

7.5 Feature Consistency (RQ4)

To answer our final research question, we analyzed the feature importance scores from the three within-project models. The results, shown in Figure 10, reveal both consistent patterns and notable differences. The most striking finding is the consistent importance of a few key features. watch_count and days_since_project_start appear in the top three most predictive features for all three projects, indicating that community interest and project maturity are consistently strong signals. However, there are also clear differences. For Cassandra and Flink, watch_count is the most important feature, while for Kafka, days_since_project_start is the dominant predictor. This suggests that while some features are universally important, the specific dynamics of each project lead to different factors having more or less influence.

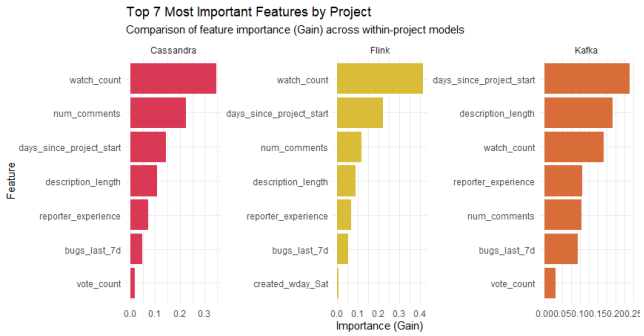

Figure 10: Top 7 Most Important Features by Project

8 Discussion

Our study demonstrates that predicting bug resolution time from structured issue tracker features is possible but remains highly limited in practice. We learned that machine learning models, particularly XGBoost, consistently outperform a simple median baseline, meaning there are genuine patterns in features such as watch count, project maturity, and number of comments. However, the size of these improvements was modest, and error margins were still much larger than the typical resolution time of most bugs. This shows that while historical metadata contains some predictive signal, it is not sufficient for reliable planning or fine-grained forecasting.

One of the most important takeaways is that bug resolution is shaped by far more than surface-level metadata. Social and organizational factors such as team workload, shifting project priorities, developer expertise, and communication bottlenecks often play decisive roles in how long an issue remains open. These factors are not reflected in structured features, which helps explain why our models plateaued in performance. This connects to prior work that emphasizes richer signals like bug text descriptions, code churn, or team processes [6,7,9]. Our results add to that body of

evidence by confirming that structured features alone cannot capture the full complexity of resolution dynamics.

We can also speculate about why hybrid transfer outperformed the other approaches, even if modestly. Adding external data likely exposed the model to a wider variety of resolution cases, helping it generalizes. However, the fact that improvements were small suggests that the main bottleneck is not data quantity but feature quality. What we cannot yet prove, but strongly suspect, is that incorporating richer, unstructured data, such as natural language from bug reports, social network measures of developer collaboration, would yield more meaningful gains than simply expanding the size of the training set.

Our study has several limitations. Methodologically, we relied on gradient boosting as our primary predictive model. While XGBoost is a strong baseline for structured data, other approaches, such as neural networks with textual embeddings or ensembles combining structured and unstructured features, could potentially capture deeper patterns. Our evaluation also used standard regression metrics like MAE and RMSE, which are sensitive to the highly skewed distribution of resolution times. This meant that a few extreme outliers had a disproportionate effect on error measures, making it difficult to evaluate performance on “typical” bugs.

In terms of data, we only worked with three Apache projects (Cassandra, Flink and Kafka). While these are large and mature systems, they may not be representative of all open-source software, particularly smaller or less formalized projects. Furthermore, our features were limited to structured fields available in JIRA trackers. Important variables such as the technical complexity of a fix, developer workload, or midstream changes in project priorities, were unavailable, leaving a significant gap in explanatory power. Finally, our preprocessing choices such as dropping inconsistent features like priority or collapsing logical fields may have limited the signal available to the model.

9 Conclusion

This research set out to test whether bug resolution times can be predicted using structured issue tracker data alone, and across different project contexts. Our findings suggest that while models like XGBoost can consistently beat a simple baseline, their improvements are small and not practically useful for planning or release management. Cross-project generalization was particularly weak, highlighting the importance of local project dynamics, while hybrid transfer showed modest gains, reinforcing the idea that more data helps but does not solve the core problem. The consistent importance of features like watch count and project maturity suggests some generalizable signals exist, but many drivers of resolution time are project specific.

Looking ahead, our work is relevant in two keyways. For practitioners, it highlights that current structured-feature models may provide coarse insights or benchmarking tools but should not be relied upon for precise forecasts. For practical use, it points toward clear directions for future work: richer feature sets that include textual semantics from bug descriptions, social and organizational signals, and developer workload data. More advanced modeling approaches, such as neural architectures or hybrid ensembles, could help capture both the skewed distribution of resolution times and the complex socio-technical dynamics underlying bug resolution.

Overall, predicting bug resolution time is a problem that matters greatly for project planning but is harder than it first appears. Our study shows there is signal to be found in structured tracker features, but the road to practical, actionable prediction will require broader data, deeper models, and closer integration of technical, social, and organizational perspectives.

REFERENCES

[1] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan, "An Empirical Study on Factors Impacting Bug Fixing Time," in Reverse Engineering (WCRE), 2012 19th Working Conference on, 2012. https://doi.org/10.1109/WCRE.2012.32

[2] Hasan Yagiz Ozkan, Poul Einer Heegaard, Wolfgang Kellerer, and Carmen Mas-Machuca. 2024. Bug Analysis Towards Bug Resolution Time Prediction. arXiv:2407.21241. Retrieved from https://arxiv.org/abs/2407.21241

[3] Adekunle Ajibode, Dong Yunwei, and Yang Hongji. 2023. Software issues report for bug fixing process: An empirical study of machine-learning libraries. arXiv:2312.06005. Retrieved from https://arxiv.org/abs/2312.06005

[4] Jagrit Acharya and Gouri Ginde. 2025. Bugsrepo: A comprehensive curated dataset of bug reports, comments and contributors information from Bugzilla. arXiv: 2504.18806. Retrieved from https://arxiv.org/abs/2504.18806

[5] Avinash Patil. 2025. Gitbugs: Bug reports for duplicate detection, retrieval augmented generation, triage, and more. arXiv:2504.09651. Retrieved from https://arxiv.org/abs/2504.09651

[6] Dimitrios-Nikitas Nastos, Themistoklis Diamantopoulos, Davide Tosi, Martina Tropeano, and Andreas L. Symeonidis. 2025. Towards an interpretable analysis for estimating the resolution time of software issues. arXiv:2505.01108. Retrieved from https://arxiv.org/abs/2505.01108

[7] Zhiyuan Chen, Vanessa Nava-Camal, Ahmad Suleiman, Yiming Tang, Daqing Hou, and Weiyi Shang. 2025. An empirical study on the capability of llms in decomposing bug reports. arXiv: 2504.20911. Retrieved from https://arxiv.org/abs/2504.20911

[8] Zengyang Li, Guangzong Cai, Qinyi Yu, Peng Liang, Ran Mo, and Hui Liu. 2024. Bug priority change: An empirical study on Apache projects. arXiv: 2403.05059. Retrieved from https://arxiv.org/abs/2403.05059

[9] Jie M. Zhang et al. 2020. A study of bug resolution characteristics in popular programming languages. arXiv: 1801.01025. Retrieved from https://arxiv.org/abs/1801.01025

Appendix A

All data, R scripts, and supporting documentation used in this study are publicly available in a GitHub repository to ensure the transparency and reproducibility of our findings.

The repository can be found at the following URL: https://github.com/UVic-Data-Science-for-SE/404-504-project-team-jem

The README.md file located in the root of the repository provides detailed, step-by-step instructions on how to run the scripts to fetch the raw data, perform the data cleaning and preparation, and execute the full modeling pipeline to reproduce the results presented in this report.