JITBot: An Explainable Just-In-Time Defect Prediction Bot

Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon
Jirayus Jiarpakdee, Chakkrit Tantithamthavorn
Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta

Faculty of Information and Communication Technology (ICT), Mahidol University, Bangkok, Thailand
Faculty of Information Technology (FIT), Monash University, Melbourne, Australia

ABSTRACT

Just-In-Time (JIT) defect prediction is a classification model that is trained using historical data to predict bug-introducing changes. However, recent studies raised concerns related to the explainability of the predictions of many software analytics applications (i.e., practitioners do not understand why commits are risky and how to improve them). In addition, the adoption of Just-In-Time defect prediction is still limited due to a lack of integration into CI/CD pipelines and modern software development platforms (e.g., GitHub). In this paper, we present an explainable Just-In-Time defect prediction framework to automatically generate feedback to developers by providing the riskiness of each commit, explaining why such commit is risky, and suggesting risk mitigation plans. The proposed framework is integrated into the GitHub CI/CD pipeline as a GitHub application to continuously monitor and analyse a stream of commits in many GitHub repositories. Finally, we discuss the usage scenarios and their implications to practitioners. The VDO demonstration is available at https://jitbot-tool.github.io/

ACM Reference Format:
Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta. 2020. JITBot: An Explainable Just-In-Time Defect Prediction Bot. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3324884.3415295

1 INTRODUCTION

Just-In-Time (JIT) defect prediction is a classification model that is trained using historical data to predict bug-introducing changes [5]. Prior studies argued that JIT defect models can provide earlier feedback for developers, while design decisions are still fresh in their minds [5]. Recently, CommitGuru [9] has been proposed to increase the adoption of JIT defect prediction. Yet, the actionability of CommitGuru is still limited due to a lack of explainability of their predictions and a lack of integration into modern software development platforms (e.g., GitHub).

Recently, researchers raised concerns that software analytics must be explainable and actionable [1, 3, 6]. In addition, the decision-making based on JIT defect prediction needs to be better justified and uphold privacy laws (i.e., developers could be laid off by committing poor code into the repositories). In particular, Article 22 of the European Union's General Data Protection Regulation (GDPR) states that the use of data in decision-making that affects an individual or group *requires an explanation for any decision made by an algorithm.* Also, Wan *et al.* [11] also raised concerns that the limited adoption of defect prediction has to do with a lack of integration into CI/CD pipelines and modern software development platforms. Jiarpakdee *et al.* [3] is among the first to use model-agnostic techniques to address the explanability challenges for defect prediction models. Yet, such techniques have never been fully integrated into a GitHub development workflow.

To address these challenges, we present an explainable Just-In-Time defect prediction bot in order to prioritize which pull requests that developers should review first by generating the riskiness of each commit, explain why such commit is risky, and suggest risk mitigation plans. To do so, for each Git repository, we first collect change-level software metrics from the repository. Then, we construct a random forest classification technique to generate the prediction of each commit (i.e., the riskiness of each commit to introducing bugs). We evaluated the models using an Area Under the Receiver Operating Characteristic curve (AUROC) and F-measure. We leveraged a model-agnostic technique to explain the predictions of bug-introducing commits. Finally, we integrated the proposed framework into the GitHub platform as a GitHub application to continuously monitor and analyse a stream of commits in many GitHub repositories. Our usage scenarios demonstrated that the explanations generated from the JITBot provide implications and benefits to developers to prioritize pull requests based on the riskiness of the latest commit and guide managers to develop risk mitigation plans.

2 AN OVERVIEW OF OUR JITBOT FRAMEWORK

JITBot is an explainable Just-In-Time defect prediction framework which automatically generates feedback to developers by providing the riskiness of each commit, explaining why such commit is risky, and suggesting risk mitigation plans. JITBot is designed as a GitHub application that is tightly integrated into continuous software development pipelines (CI/CD). The primary inputs of JITBot are a pull request and related commit information, triggered by commits

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3415295

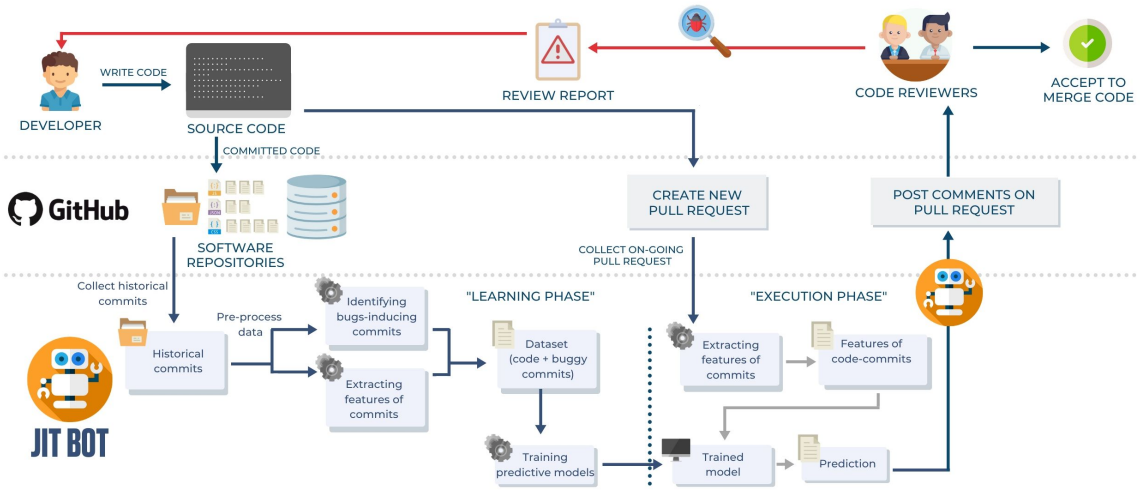

Figure 1: The architecture overview of JITBot.

made by developers. For a given pull request, the outputs of JITBot are three-fold: (1) the riskiness that a given commit will introduce defects in the future; (2) an explanation of the prediction; and (3) suggestion for improvement to mitigate the risk.

Figure 1 presents an overview of how JITBot works. First, one needs to install and add our JITBot to a GitHub repository. When one submits a pull request to the repository, GitHub will send a webhook to our JITBot to trigger the back-end application. Our back-end application will return results to the GitHub via a webhook again and the feedback will be presented at the comments of the pull request. Below, we describe the front-end and back-end architecture and the implementation details of our JITBot framework.

2.1 The Front-end Implementation

When a developer submitted a pull request to a GitHub repository, GitHub will send a webhook to the back-end of the JITBot app which has been linked during the installation. Then, the back-end of the JITBot app will use the trained models to generate predictions and explanations, and return results to the pull request. We developed the JITBot app using Node.js with the Probot framework.

2.2 The Back-end Implementation

The back-end implementation can be further divided into 2 phases: learning phase and execution phase. The learning phase occurs at the beginning after the JITBot app has been installed into a GitHub project. It is responsible for creating predictive models. The execution phase occurs after the predictive models are trained and ready to be used. The JITBot app will respond to each pull request and generate defect predictions with explanations.

2.2.1 The Learning Phase. This phase consists of two steps.

Step 1: Collect change-level metrics. Similar to Kamei *et al.* [4], we collected eight change-level metrics with respect to three dimensions (see Table 1).

Table 1: Summary of change-level metrics

Category	Name	Description
4*Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
3*Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix

Diffusion category measures how distributed a change is. A highly distributed change is more complex and more prone to defects, as shown in prior work [2]. We collected the number of modified subsystems (NS), the number of modified directories (ND), the number of modified files (NF), and the distribution of modified code across each file (Entropy), to measure the diffusion of a change. Similar to Hassan [2], we normalized the entropy by the maximum entropy $log_2 n$ to take the differences in the number of files n across changes into account. *Size category* measures the size of a change using the lines added (LA), lines deleted (LD), and lines total (LT). The intuition is that the size of a change is a strong indicator of the change's defect-proneness [7]. *Purpose category* measures whether a change fixes a defect. The intuition is that a change that fixes a defect is more likely to introduce another defect.

Step 2: Train JIT models. Since prior studies have demonstrated that random forest is the top-performing classifiers [10], we trained JIT models using a random forest classification technique.

2.2.2 The Execution Phase. After the predictive models are trained in the learning phase, they are put to use every time a new pull request is created. This phase consists of 3 steps as explained below.

Step 1: Generate predictions. When a new pull request comes in, the JITBot app extract a list of commits from the pull request

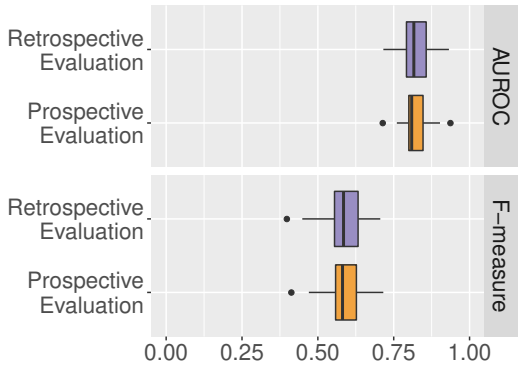

Figure 2: The distributions of the performance of our JITBot models for the retrospective scenario and the prospective scenario for AUC and F-measure.

and analyze each commit individually. The change-level metrics are collected from each commit, and then the tool generates the prediction of the commit (i.e., the riskiness that the commit will introduce bugs).

Step 2: Explain the predictions. We leveraged a model-agnostic technique to explain the predictions of bug-introducing commits, that is suggested by our recent work [3]. LIME constructs a local regression model from the instance of interest and its surrounding random samples to measure the contributions of metrics towards the final probability of the prediction of such instance. The implementation of LIME is provided by the `lime` python library.

Step 3: Return results. Finally, the back-end system will return the output from the models to the given pull request. In particular, the results aim to answer the following three questions:

(Q1) *What is the risk of each bug-introducing commit?*
(Q2) *Why is the commit predicted as a bug-introducing commit?*
(Q3) *How to mitigate the risk of bug-introducing commits?*

3 MODEL EVALUATION

To investigate how well our JITBot can accurately predict bug-introducing changes, we evaluated the predictive accuracy of the JIT models using 21 large open-source software systems. The studied datasets are collected through an API of the CommitGuru [9]. We performed 2 evaluation scenarios, i.e., retrospective evaluation and prospective evaluation. First, we split 70% of each dataset for retrospective evaluation, and 30% for prospective evaluation. For retrospective evaluation, we used the 10-fold cross-validation technique. For prospective evaluation, we used the retrospective samples for constructing JIT models and the prospective samples for estimating the accuracy of our JIT models on unseen data. We used the Area Under the Receiver Operating Characteristic curve (AUROC) and the F-measure (i.e., a harmonic mean ($\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$) of precision ($\frac{\text{TP}}{\text{TP}+\text{FP}}$) and recall ($\frac{\text{TP}}{\text{TP}+\text{FN}}$)) as the studied performance measures, which is calculated from a confusion matrix. True positive (TP) is the number of correctly classified as bug-introducing commits. False positive (FP) is the number of incorrectly classified bug-introducing commits. False positive (FN) is the number of incorrectly classified non-bug-introducing commits.

Our JITBot models consistently yield high model performance for both retrospective and prospective evaluation scenarios. Figure 2 shows the distributions of the performance of our JIT models for both retrospective scenario and prospective evaluation scenarios. For the retrospective scenario, the performance of our JIT models is, at the median, 0.82 for the AUROC and 0.58 for the F-measure. Similarly, for the prospective scenario, the performance of our JIT models is, at the median, 0.80 for the AUROC and 0.59 for the F-measure. The high model performance for both studied scenarios across the studied large open-source software systems indicates that our JIT models can accurately estimate and explain the risk of bug-introducing commits.

4 USE-CASE SCENARIO

Problem Motivation. Let's imagine that developers have a stream of pull requests to be reviewed. Exhaustively reviewing all pull requests are infeasible due to limited resources. Thus, our JITBot is designed to help developers prioritize their code review effort when reviewing a stream of pull requests and provide guidance to help developers make data-informed decisions.

Demonstration. To demonstrate a use-case scenario of our JITBot, we selected the pull request ID 3349 from the Apache Camel project as a subject of this use-case scenario. Installation of our JITBot typically needs authorization from the core developers who have an admin permission of GitHub projects. Thus, simulating the real-world pull requests remains challenging. To address this challenge, we simulated the given pull request by replicating the same pull-request on the repository that we fork from the original repository. This pull request consists of 2 commits, i.e., the correctly predicted bug-introducing commit (`e21c9a8`) and the correctly predicted non bug-introducing commit (`1a8a375`). Figure 3 shows the generated feedback (i.e., the riskiness, the explanation, and the risk mitigation plans) from our JITBot for each commit of the studied pull request.

4.1 Explain a moderate-risk estimation of a bug-introducing commit (`e21c9a8`)

Our JITBot suggested that the commit `e21c9a8` has a moderate risk of 63% to be a bug-introducing commit. Based on the supporting scores (towards class Defect), our JITBot explains the top-3 factors that increase the risk as follows: (1) the low experiences of developers in the subsystem *increases* the risk by 10.53%, (2) the low experiences of recent developers in the subsystem *increases* the risk by 4.67%, and (3) the high number of deleted lines of code in this commit *increases* the risk by 2.85%. With respect to the risk of being a bug-introducing commits, our JITBot offers 2 risk mitigation plans as follows: the author of this commit (1) should ask developers who have experience in the subsystem to review this commit; and (2) may consider reducing the number of deleted lines of code.

4.2 Explain a low-risk estimation of a bug-introducing commit (`1a8a375`)

Our JITBot suggested that the commit `1a8a375` has a low risk of 34% to be a bug-introducing commit. Based on the contrasting scores (towards class Clean), our JITBot explains the top-3 factors that increase the risk as follows: (1) the high number of prior changes to the modified files in this commit *increases* the risk by 7.3%; (2) the

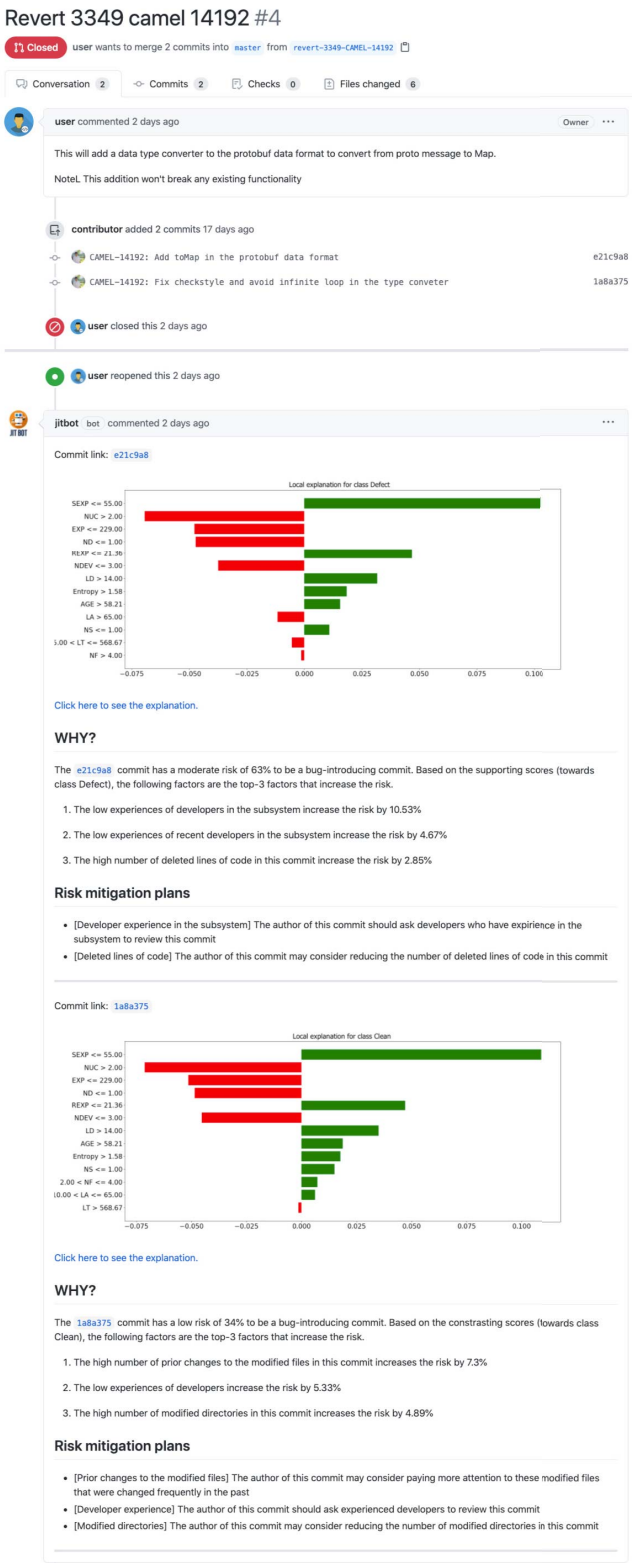

Figure 3: An example screenshot of our JITBot and the generated feedback for the pull request ID 3349 of the Apache Camel project.

low experiences of developers *increases* the risk by 5.33%; and (3) the high number of modified directories in this commit *increases* the risk by 4.89%. With respect to the risk of being a bug-introducing commits, our JITBot offers 3 risk mitigation plans as follows: the author of this commit (1) may consider paying more attention to these modified files that were changed frequently in the past; (2) should ask experienced developers to review this commit; and (3) may consider reducing the number of modified directories.

5 CONCLUSION

This paper presents an explainable Just-In-Time defect prediction bot (JITBot) in order to prioritize which pull requests that reviewers should review first by generating the riskiness of each commit, explain why such commit is risky, and suggest risk mitigation plans. Through an illustration of a use-case scenario, our JITBot offers many implications to reviewers, developers, and managers. First, the riskiness of being bug-introducing commits from our JITBot could help reviewers to prioritize their limited resources on the most risky pull requests so reviewers could save a huge amount of effort on code review, especially, in a rapid release setting. Second, the explanation of the predictions from our JITBot could help developers to better understand why the JIT models make that decisions. Such explanations could help reviewers and developers understand how to mitigate the risk of bug-introducing commits and help managers develop risk mitigation plans to avoid such mistakes in the future. Nevertheless, we noted that our explanations must be used as a guidance for developing QA improvement plans, instead of a causal effect. Our future work will focus on explaining the finer-level granularity of JIT models [8].

Acknowledgement. We thank anonymous reviewers for their constructive feedback. CT was supported by ARC DECRA (DE200100941) and a Monash-FIT Early Career Researcher Seed Grant.

REFERENCES

[1] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2018. Explainable Software Analytics. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 53–56.
[2] Ahmed E. Hassan. 2009. Predicting Faults using the Complexity of Code Changes. In *ICSE '09*. 78–88.
[3] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2020. An Empirical Study of Model-Agnostics Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)* (2020).
[4] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.
[5] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *IEEE Transactions on Software Engineering (TSE)* 39, 6 (2013), 757–773.
[6] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. 2013. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *ICSE '13*. 372–381.
[7] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *ICSE '05*. 284–292.
[8] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software (JSS)* 150 (2019), 22–36.
[9] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit Guru: Analytics and Risk Prediction of Software Commits. In *FSE '15*. 966–969.
[10] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *ICSE '16*. 321–332.
[11] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering (TSE)* (2018).