PRECOG: Pull Request Prioritization and Visualization

Hugo Raskin
University of Namur
Namur, Belgium
hugo.raskin@student.unamur.be

Xavier Devroey
NADI, University of Namur
Namur, Belgium
xavier.devroey@unamur.be

Benoît Vanderose
NADI, University of Namur
Namur, Belgium
benoit.vanderose@unamur.be

***Abstract*—A Pull Request (PR) is a code change proposal submitted by a developer for review by other team members. For a reviewer, prioritizing and selecting a PR to review is not easy: this selection can depend on many factors, including the extent of the change, the time available, their state of mind, etc. Existing tools and platforms like GitHub offer support for visualizing code changes and recommending reviewers to assign a PR to. However, little research has been done on providing support to the reviewer to select a PR to review from a list of (assigned) PRs. In this paper, we propose PRECOG, a Visual Studio Code plugin to help reviewers select the appropriate PR by enabling more informed decisions. PRECOG offers a radar chart visualization of the PR's difficulty across various dimensions, and ranks PRs according to their difficulty. We evaluated PRECOG with 9 developers in a user experiment on the Java Spring Framework. Our results show that although participants differed in how they assess PRs' difficulty, all agreed that the synthetic view provided by PRECOG helps them make rapid, informed decisions about which PR to review first. PRECOG is available at https://github.com/snail-unamur/PRECOG. A demonstration video is available at https://youtu.be/qPlr6lmgyMM.**

***Index Terms*—pull request, code review, developer experience, measurement, tool**

I. INTRODUCTION

Pull Requests (PRs), also known as merge requests, are increasingly used in software development to separate development effort from integration into the codebase. The decision to include (*merge*) changes into the codebase results from a code review process carried out by a developer (a *reviewer*), usually different from the *author* of the changes. The benefits of code review include preventing potential defects, enhancing code readability, and sharing knowledge of the source code [1], [2]. Version control platforms like GitHub assist reviewers by providing tools to visualize changed code (e.g., diff views), suggest improvements, and leave comments. These platforms also present PRs as a list for reviewers to select from.

Previous studies have attempted to identify the socio-technical factors influencing the decision to merge PRs [3]–[6], which have driven research on developing ranking techniques for pull and code review requests [7], [8]. Building on our previous study [9] examining the triggers and effects of *confirmation bias* and *decision fatigue* during code review, we want to investigate how more informed decisions based on *objective factors* (i.e., software metrics that indicate how complex, large, critical, or potentially poorly designed the code in the PRs is) can help reviewers select an appropriate PR to review from a set of pre-assigned PRs. To that end, we introduce PRECOG, a Visual Studio Code extension that can serve as a basis for investigating different (socio-)technical metrics to enhance *empirical-based decisions* for reviewers. Building upon the original *Pull Request plugin* developed by Microsoft, PRECOG offers: (i) a visual representation of each PR's difficulty, and (ii) an automatically generated list ranking PRs based on their *difficulty*. This difficulty is defined by a set of metrics collected from various data sources, i.e., CodeQL and SonarCloud, in this version of the tool.

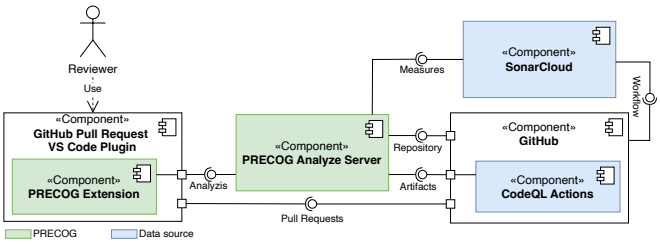

Fig. 1. PRECOG architecture and workflow

The remainder of this paper is structured as follows: Section II presents PRECOG's architecture and workflow. Section III presents PRECOG's user evaluation on an open-source project, Java Spring. Section IV discusses the applications of PRECOG, followed by Section V, which reviews related work, and Section VI concludes this paper.

II. APPROACH

Conceptually, the architecture of PRECOG, as shown in Figure 1, consists of various data sources (blue boxes in the schema) whose data are aggregated by a server to compute the different metrics, and then transmitted to a client to be displayed (green boxes in the schema) to a reviewer.

The implementation involves adding a client (PRECOG *Extension*) to the GitHub Pull Request plugin and an aggregator (PRECOG *Analyze Server*). Data are collected by the server

This research was funded by the CyberExcellence by DigitalWallonia project (No. 2110186), funded by the Public Service of Wallonia (SPW Recherche).

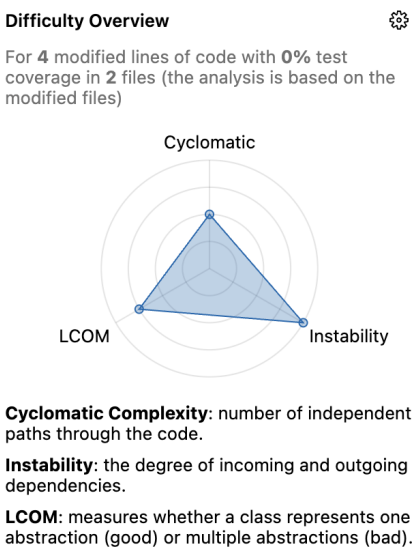

Fig. 2. PRECOG PR difficulty overview

from SonarCloud, GitHub, and artifact (i.e., query results) produced by CodeQL (a code analysis engine developed by GitHub to query code) GitHub actions. Once processed, the data can be retrieved by the PRECOG client via the GitHub plugin and displayed to the reviewer.

A. PRECOG *Extension*

In addition to the original features of the GitHub Pull Request plugin, PRECOG supports the *visualization of a PR difficulty* and *automatically ranks PRs based on their difficulty*.

1) PR difficulty: The visualization of a PR's difficulty, shown in Figure 2, consists of a radar chart displayed on each PR overview page. The radar axes define several metrics, each with configurable thresholds that normalize their values to a common scale from 0 to 4.

The radar uses CodeQL to measure three class-level metrics. The first, *cyclomatic complexity*, quantifies the number of independent paths through the code and indicates how difficult it is to understand and maintain. The *Chidamber–Kemerer lack of cohesion (LCOM)* assesses whether a class represents a single abstraction or multiple concerns: a high score indicates a "god class," which is harder to maintain. Finally, *instability* measures the degree of coupling between classes, computed from efferent coupling (C_e), the number of outgoing dependencies, and afferent coupling (C_a), the number of ingoing dependencies, obtained via CodeQL: Instability $= C_e/(C_e + C_a)$. Together, these three metrics capture different aspects of software quality: internal complexity (cyclomatic complexity, LCOM) and inter-class dependencies (instability). For each PR, a *difficulty score* is calculated based on the area of the radar. This score is then used to rank the PR list from easiest to most difficult automatically.

In addition to the radar metrics, the plugin retrieves contextual information from SonarCloud: the number of modified files, the number of modified lines, and the percentage of new test coverage. These data inform reviewers about the PR's scope. For instance, in Figure 2, the PR exhibits a cyclomatic complexity of 2 across two modified files.

2) Auto-ranked PRs list: The difficulty score is used to sort PRs in a drop-down list from least complex (relative to the configured metrics) to most complex. The rationale is that, for a reviewer to select a PR, they need a general understanding of the various PR difficulties. Manually sorting a list of PRs might involve reading the title and description to gauge difficulty, which, for many PRs, can take considerable effort and time. PRECOG seeks to simplify this process by automatically sorting the PRs, easing the reviewer's task. Depending on the context (time available, state of mind, etc.), the reviewer can select a PR from the top (easier PRs), the bottom (harder PRs), or the middle of the list.

B. PRECOG *analyze server*

A central component of the architecture is the PRECOG analyze server. Its main purpose is to collect data and metrics from different data sources to provide them to the PRECOG plugin extension. PRECOG is designed to be generic and configurable regardless of languages and metrics. Each analyzed repository can define a specific configuration file that the server will first retrieve. This file specifies metrics to compute, how to compute them (i.e., which data source to contact), and the threshold value for each metric to build the radar chart. The rationale is that each repository will require a specific configuration tailored to the development team and the programming languages used. The server is meant to be hosted by an organization or a team, as it uses credentials and tokens specific to their repositories and data sources.

C. *Data sources*

The tools used to analyze repositories and PRs are standard, as the plugin's goal is not to compute new metrics but to inform the reviewer of different measurements during the review. To this end, PRECOG mainly relies on *SonarCloud* and *CodeQL*, each triggered by GitHub Actions when a specific PR is created or new code is pushed to the PR branch. PRECOG itself can be easily extended to consider other metrics or data sources (e.g., new CodeQL queries) using configuration files.

III. USER EVALUATION

This section presents our results from an empirical user evaluation of PRECOG on the open-source Java Spring framework. We chose Spring because it is well-maintained and receives continuous PRs. We randomly selected nine open PRs in the repository. The only criterion was that the selected PRs must not conflict with the main branch, as conflicting PRs would be trivially rejected with a comment asking to resolve the conflict. For PRECOG, such PRs in conflict would not trigger the necessary GitHub Actions. For the configuration, we used the data sources and metrics presented in Section II.

We relied on convenience sampling and a LinkedIn post to recruit participants. In total, the evaluation involved nine participants, which is similar to related work [10]. Two participants had less than 6 months of software development

experience, 4 had experience ranging from 1 to 5 years, and 2 had experience ranging from 5 to 10 years. On a scale from 1 to 5, three participants rated their expertise in code review as 4, one as 3, four as 2, and one as 1.

A. Data collection

Each session is recorded and begins by asking the participant for their agreement, and to install and configure the plugin on Visual Studio Code on their machine, following a given procedure. We then ask the participant to complete two tasks:

1) Ranking the PRs: First, each participant has 20 minutes to manually rank the 9 PRs from least complex to most complex. No guidelines are provided to participants regarding *difficulty*, and each participant must use their own criteria. Once the twenty minutes have ended, the participants are given the list of PRs ranked by the plugin. We then ask them to compare their list with the auto-ranked list and give their opinion and interpretation of the differences.

2) Evaluating the PR difficulty: Secondly, each participant is asked to read the radar chart of four PRs: the ones they rated as the least and most complex in the first use case, and the ones that PRECOG ranked as the least and most complex. We then ask participants to fill out a questionnaire to determine (i) whether the radar informs them about the PR's difficulty, and (ii) whether the radar is consistent with the PR's difficulty.

At the end of the evaluation, we asked each participant to complete a standard *User Experience Questionnaire (UEQ)* [11], [12], a heavily validated, state-of-the-art questionnaire that measures user experience based on established scales: Attractiveness (do users like or dislike the product?), Perspicuity (is it easy to familiarize oneself with the product?), Efficiency (can users complete their tasks without unnecessary effort?), Dependability (does the user feel in control of the interaction?), Stimulation (is it exciting and motivating to use the product?), and Novelty (is the product innovative and creative?). In addition to the UEQ, we ask participants to answer questions about their overall experience (see Figure 5), strengths and weaknesses of the plugin, and if they have any suggestions for metrics to add or remove.

The questionnaires,the use cases, collected data, and data analysis are available in our replication package: https://github.com/snail-unamur/precog-plugin-evaluation

B. Results

1) Ranking PR evaluation: The results of the PRs ranking evaluation are presented as a Sankey Diagram in Figure 3, where the left column contains the PRs automatically ranked by PRECOG from the least (ranked 1) to the most complex (ranked 9) based on the enabled metrics, and the right column shows where the participants placed the corresponding PR.

Two observations can be drawn from the diagram. Firstly, the rankings of PRECOG and the participants differ significantly. For instance, the PR identified by PRECOG as the most complex (PR #9) received highly divergent rankings from participants: only one rated it as most complex, while two placed it at the opposite end. Secondly, participants' rankings

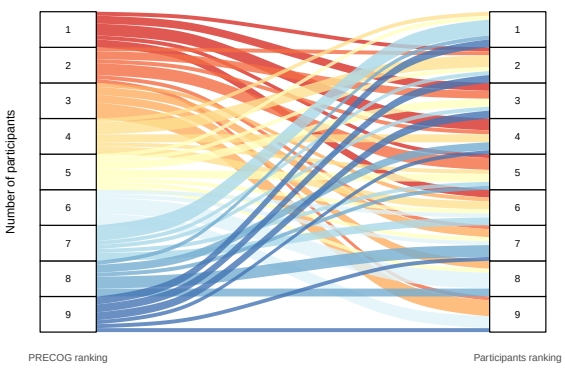

Fig. 3. Results of the ranking use case

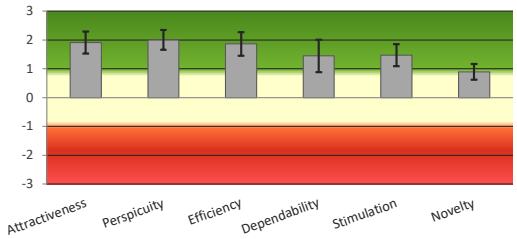

Fig. 4. UEQ scales (Mean and Error bar)

also vary significantly, with PRs ranked in different positions. This indicates that each participant uses different criteria to rank the PRs. For example, when a participant was asked which criteria they used, they indicated that one of the PRs used concurrency, which they ranked higher because they were less experienced with this aspect. When asked if the easier PRs were correctly ranked, four participants agreed. However, when asked about the difficult PRs, four participants disagreed, and one strongly disagreed, emphasizing the divergence in rankings between PRs perceived as easier or difficult.

As shown by the results, ranking a list of PRs is not easy, as subjective factors influence participants' rankings. This divergence highlights the difficulty of capturing human perception of *difficulty* through a single metric-based ranking. Future work will explore how subjective factors, such as developer experience or familiarity with specific technologies, could be systematically integrated into our approach.

2) User experience evaluation: Participant responses to the UEQ were analyzed using the *UEQ Data Analysis Tool* [12]. The tool calculates scores for six user experience scales. Responses, originally on a 1-to-7 scale, are converted to a range from -3 (negative) to +3 (positive). These values are then used to compute the mean and error bar for each scale. Results for the six scales are reported in Figure 4.

The results show that, among the six scales, three stand out: *Attractiveness*, *Perspicuity*, and *Efficiency*, with *Perspicuity* receiving the highest score. This suggests that PRECOG provides clear, understandable information, thereby demonstrating potential to enhance developer efficiency. In addition, PRECOG is perceived as an attractive tool.

Conversely, the *Novelty* scale received the lowest score. Some participants explained that *"*PRECOG *uses data that are already available in different places; its strength is to make*

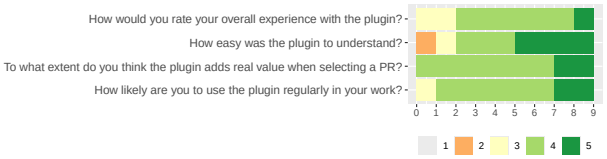

Fig. 5. Results of the questions asked after performing the use cases

them available in the same place." Others found it difficult to evaluate the novelty and level of stimulation, as it is designed for use in a professional working context, which requires seamless integration, rather than disrupting a given practice.

3) Questionnaire: Figure 5 presents the results of the questions asked after using PRECOG. Overall, despite the ranking mismatch, the participants were positive about their experience with the plugin. One participant indicated that the different metrics should be better explained to better understand the visualization. All participants were positive about the added value of PRECOG for selecting PRs to review, and the majority would be willing to integrate the plugin into their work.

Regarding PRECOG's strengths, participants pointed out the readability and usefulness for quickly having an idea of the PRs' complexity. Regarding weaknesses, the participants pointed out difficulties understanding some metrics, a lack of other dimensions not directly related to code in the metrics (e.g., number of changes), and a lack of accuracy in handling subtle difficulties (e.g., concurrency in the changed code).

Regarding the radar metrics, all participants agree that the three are useful. Two suggested adding metrics on the number of modifications (lines and files) involved in the PR directly in the radar, rather than in a separate text. Finally, one participant suggested a measure of "context switching" (i.e., usually denoting the presence of shotgun surgery code smell) required to review the PR, and another suggested a measure of "specification coverage" to gauge the impact of the changes on the application's features.

IV. DISCUSSION

A. PRECOG *evaluation*

The main takeaway from our evaluation is that, despite obtaining different rankings from participants and PRECOG (as shown in Figure 3), participants are positive about the potential impact of PRECOG on PR reviewing. This is confirmed by the UEQ results, where all dimensions (except Novelty) are well ranked, and the answers to our final questionnaire. Those results encourage us to continue our efforts to develop PRECOG and explore new dimensions for the radar and ranking. Similar to our previous work [9], we intend to continue using a developer-centered approach to gain a deeper knowledge of how developers select PR for reviewing.

For our future work, although the number of participants remains limited, our evaluation enabled us to gather several perspectives and identify potential directions. The participants have different levels of seniority and varying levels of expertise in code review. Depending on their experience, they paid attention to different aspects of the pull requests to sort. For instance, one junior participant was looking for documentation in the pull request, or if it was linked to an issue. Another more senior participant looked for tests and ranked all PRs without tests as easy, as those would likely be rejected with a comment asking for testing the changes. A third senior participant searched code changes for clues, such as *multithreading*, to understand what the code does and its technical implications before ranking the PRs. Those observations are consistent with the different rankings presented in Figure 3.

The different strategies identified by our participants could be incorporated into future versions of PRECOG to add more dimensions to the difficulty scoring, e.g., by leveraging static analysis on code changes and natural language processing on PR and issue comments. Several participants were sensitive to change dispersion, a phenomenon related to the *shotgun surgery* code smell. PRECOG relies on class-level metrics, which are averaged across changes that affect multiple classes. Additionally, CodeQL does not support querying changes directly. It only supports querying the classes they affect. These reduce the precision of the metrics used to estimate *difficulty*. They provide an approximation that we will further validate in our future work.

Finally, the evaluation was done using an open-source project for which none of our participants is a maintainer. This means we could not investigate how business-oriented metrics (e.g., deadlines, customer impact, roadmap) can influence developers' PR selection strategies. This requires further investigation using dedicated research approaches.

B. PRECOG *implementation*

The main challenge in implementing PRECOG was collecting data from various sources to present to the reviewer. For the front-end, we relied on the widely used GitHub Pull Request plugin for Visual Studio Code. Besides reducing our implementation time and seamlessly integrating with GitHub, it offers the advantage of using a tool familiar to the reviewers (as evidenced by the UEQ's lower Novelty rating). For data collection, we developed the PRECOG Analysis Server. In earlier versions, the server relied solely on the SonarCloud and GitHub APIs to collect metrics, but this approach was too limited in terms of (i) metric diversity and (ii) extensibility. To address that, we considered performing measurements on the server itself, but this would have required setting up yet another build environment with appropriate access tokens. Instead, we chose to rely on GitHub Actions for CodeQL, which leverages the existing continuous integration environment and gives the development team full control over the collected data. PRECOG supports adding new metrics to the radar and computing rankings via configuration files, enabling customization without modifying the source code.

For our future work, we will focus on two implementation aspects: (i) defining dedicated CodeQL queries to collect metrics capturing the dimensions identified during our evaluation (e.g., the number of context switches required to perform the review and the number of features impacted). And (ii) allowing developers to customize the ranking and dimensions in the radar, based on a locally defined profile.

V. Related work

Previous research has investigated the criteria that influence whether a PR is accepted or rejected. These studies examine various dimensions, including meta-information (e.g., number of modified files, age of the project) [13]–[15], code-related aspects such as the presence of code smells [5], and social factors related to the contributors [16]. Our work complements this line of research. We aim to make explicit and visible measurable aspects of PRs to help developers make an informed choice when selecting a PR to review. Some of this information, like the number of modified files, is already available in PRECOG. In our future work, we intend to extend this list to other data and enhance the radar and ranking.

Additional research has addressed PR prioritization using algorithms or machine learning techniques [7], [17], [18]. Some approaches have also focused on recommending PRs to specific reviewers based on expertise or past interactions [19], [20]. Such approaches could be envisioned for PRECOG to rank the different PRs. However, ML-based and other recommendation approaches usually require large amounts of data, which are not always available. In our case, PRECOG can be used by various developers regardless of the project history. Yet, personal expertise and past interactions with the codebase are an interesting path to explore for our future work.

Finally, studies have aimed to enhance the review and recommendation process by improving how PR characteristics are visualized or presented to reviewers. Rahman et al. [21] investigated how a graphical visualization of developers' contributions, expertise, collaborations, and current workloads can help managers assign reviews. In our case, we assume a developer has already received a batch of PRs to review and must select which to start with. In their recent work, Göçmen et al. [10] explored how metrics and an overall risk score derived from a change impact analysis combining call graph-based dependency analysis and history mining techniques can help ranking and reviewing PRs. They validated their approach through two focus groups with 7 participants, emphasizing the potential benefits of change-impact analysis for code review. This was also suggested by some participants during our empirical evaluation. We intend to explore how lightweight change impact analysis can be integrated into PRECOG.

VI. Conclusion And Future Work

In this paper, we presented PRECOG, a Visual Studio Code plugin that visualizes a PR's difficulty and ranks PRs using a defined difficulty score. PRECOG is based on the GitHub Pull Request plugin. It aims to help reviewers make more informed decisions when selecting a PR to review and increase reviewers' awareness of PR characteristics during review.

We conducted an empirical user evaluation of PRECOG on the open-source Java Spring framework. In total, nine participants were asked to complete two tasks: ranking a list of PRs and assessing their difficulty. Additionally, participants had to answer a UEQ and specific questions about the experiment. Our results show that for PR ranking, subjective factors influence PR positions, leading to divergence in participants' rankings. Nevertheless, the UEQ and subsequent questions show that all participants see PRECOG's potential benefits.

In our future work, we will extend PRECOG by including additional dimensions. To that end, we will explore how to define new CodeQL queries and leverage additional data sources. We will also enhance customization for specific developer profiles to account for other factors, such as experience and familiarity with technologies, to produce a tailored ranking and adjust the visualization of PRs. Finally, we will evaluate PRECOG in a full-scale experiment (e.g., using focus groups) to gain a deeper knowledge of how developers select PR for reviewing.

References

[1] D. Badampudi, M. Unterkalmsteiner, and R. Britto, "Modern Code Reviews—Survey of Literature and Practice," *ACM Trans. on Soft. Eng. and Meth.*, vol. 32, no. 4, pp. 1–61, Oct. 2023.

[2] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *ICSE '18*. ACM, May 2018, pp. 181–190.

[3] G. Gousios, A. Zaidman, M.-A. Storey, and A. v. Deursen, "Work Practices and Challenges in Pull-Based Development," in *ICSE '15*, May 2015, pp. 358–368.

[4] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub," in *ICSE '14*, May 2014, pp. 356–366.

[5] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? An empirical study," *JSS*, vol. 171, p. 110806, Jan. 2021.

[6] O. Kuhejda and B. Rossi, "Pull Requests Acceptance: A Study Across Programming Languages," in *SEAA '23*, Sep. 2023, pp. 378–385.

[7] G. Zhao, D. A. da Costa, and Y. Zou, "Improving the pull requests review process using learning-to-rank algorithms," *Emp. Soft. Eng.*, vol. 24, no. 4, pp. 2140–2170, Aug. 2019.

[8] L. Yang, B. Liu, J. Jia, J. Xu, J. Xue, H. Zhang, and A. Bacchelli, "Prioritizing code review requests to improve review efficiency: a simulation study," *Emp. Soft? Eng.*, vol. 30, no. 1, p. 24, Jan. 2025.

[9] T. Jetzen, X. Devroey, N. Matton, and B. Vanderose, "Towards Debiasing Code Review Support," in *CHASE '25'*. IEEE, Apr. 2025, pp. 143–148.

[10] I. S. Göçmen, A. S. Cezayir, and E. Tüzün, "Enhanced code reviews using pull request based change impact analysis," *Emp. Soft. Eng.*, vol. 30, no. 3, pp. 1–44, May 2025.

[11] B. Laugwitz, T. Held, and M. Schrepp, "Construction and Evaluation of a User Experience Questionnaire," in *USAB '08*. Springer, 2008, pp. 63–76.

[12] M. Schrepp, A. Hinderks, and J. Thomaschewski, "Applying the User Experience Questionnaire (UEQ) in Different Evaluation Scenarios," in *DUXU '14*. Springer, 2014, vol. LNCS 8517, pp. 383–392.

[13] D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, "Acceptance factors of pull requests in open-source projects," in *SAC '15*. ACM, 2015, pp. 1541–1546.

[14] M. M. Rahman and C. K. Roy, "An insight into the pull requests of GitHub," in *MSR '14*. ACM, May 2014, pp. 364–367.

[15] X. Zhang, Y. Yu, G. Gousios, and A. Rastogi, "Pull request decisions explained: An empirical overview," *IEEE Trans. on Soft. Eng.*, vol. 49, no. 2, pp. 849–871, 2022.

[16] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," in *WCRE '13*, Oct. 2013, pp. 122–131.

[17] E. Van Der Veen, G. Gousios, and A. Zaidman, "Automatically prioritizing pull requests," in *MSR '15*. IEEE, 2015, pp. 357–361.

[18] M. I. Azeem, S. Panichella, A. Di Sorbo, A. Serebrenik, and Q. Wang, "Action-based Recommendation in Pull-request Development," in *ICSSP '20*. ACM, Sep. 2020, pp. 115–124.

[19] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review," in *SANER '15*, Mar. 2015, pp. 141–150.

[20] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Who Should Review this Pull-Request: Reviewer Recommendation to Expedite Crowd Collaboration," in *APSEC '14*, Dec. 2014, pp. 335–342.

[21] M. S. Rahman, D. Mondal, Z. Codabux, and C. K. Roy, "Integrating Visual Aids to Enhance the Code Reviewer Selection Process," in *ICSME '23*, Oct. 2023, pp. 293–305.