Evaluating Learning-to-Rank Models for Prioritizing Code Review Requests using Process Simulation

Lanxin Yang*, Bohan Liu*, Junyu Jia*, Junming Xue*, Jinwei Xu*, Alberto Bacchelli†, He Zhang*

*State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University, China
†Department of Informatics, University of Zurich, Switzerland
yang931001@outlook.com, bohanliu@nju.edu.cn, {jjyu_keji, xjm990917, jinwei_xu}@163.com,
bacchelli@ifi.uzh.ch, hezhang@nju.edu.cn

***Abstract*—In large-scale, active software projects, one of the main challenges with code review is prioritizing the many Code Review Requests (CRRs) these projects receive. Prior studies have developed many Learning-to-Rank (LtR) models in support of prioritizing CRRs and adopted rich evaluation metrics to compare their performances. However, the evaluation was performed before observing the complex interactions between CRRs and reviewers, activities and activities in real-world code reviews. Such a `pre-review` evaluation provides few indications about how effective LtR models contribute to code reviews. This study aims to perform a `post-review` evaluation on LtR models for prioritizing CRRs. To establish the evaluation environment, we employ Discrete-Event Simulation (DES) paradigm-based Software Process Simulation Modeling (SPSM) to simulate real-world code review processes, together with three customized evaluation metrics. We develop seven LtR models and use the historical review orders of CRRs as baselines for evaluation. The results indicate that employing LtR can effectively help to accelerate the completion of reviewing CRRs and the delivery of qualified code changes. Among the seven LtR models, LambdaMART and AdaRank are particularly beneficial for accelerating completion and delivery, respectively. This study empirically demonstrates the effectiveness of using DES-based SPSM for simulating code review processes, the benefits of using LtR for prioritizing CRRs, and the specific advantages of several LtR models. This study provides new ideas for software organizations that seek to evaluate LtR models and other artificial intelligence-powered software techniques.**
Data&materials: https://figshare.com/s/a033e99cd2a61e64c8bc.

***Index Terms*—Modern code review, code review request, learning to rank, software process simulation modeling, discrete event simulation**

I. Introduction

Code review is a widespread process in which peer reviewers manually examine code changes to detect quality issues [1, 2]. Code review was originally conducted in a synchronous and heavyweight manner, known as Fagan inspections. Nowadays, the Pull-based Development (PbD) model [3] has become dominant and allows code authors and reviewers to participate in code reviews in an asynchronous and lightweight manner [4]. Such a form of code review is known as Modern Code Review (MCR) [5].

Despite its benefits, conducting proper MCR is challenging [6, 7, 8, 9]. In large-scale, active software projects, one of the main challenges with code review is prioritizing the many Code Review Requests (CRRs) these projects receive [10]. CRRs sometimes fail to get a timely response because their number exceeds the workload of reviewers, thus leading to delayed *post-review* software activities such as build, testing, and deployment; a long wait may result in code authors experiencing negative feelings, especially if their code changes are eventually rejected [11]. Such long delays and rejections are especially unpleasant to newcomers in Open-Source Software (OSS) communities, who could be discouraged from making further contributions [12, 13]. A solution to this problem is to *prioritize CRRs*. In fact, a good prioritization mechanism can help detect important and urgent code changes (*e.g.*, fixes for high-impact bugs), which can be merged more quickly and their delivery to *post-review* software activities accelerated. Also, such a mechanism can help code reviewers better schedule their time and pay attention to the high-priority CRRs, and help code authors receive more timely feedback to improve their changes [14, 15].

As shown in Figure 1, so far most software platforms (*e.g.*, GitHub and GitLab) provide only basic rules for ranking CRRs, *e.g.*, *newest* and *recently updated*. There is a wider range of complex considerations besides recency in practice, *e.g.*, *outcome* and *urgency* [14]. Studies [14, 15, 16, 17, 18] have employed learning-based prioritizers to address this problem automatically. These prioritizers are based on a model (henceforth: Learning-to-Rank or LtR model) that decides on an optimal ordering of an entire list of CRRs. Selecting an effective LtR model is critical to prioritize CRRs.

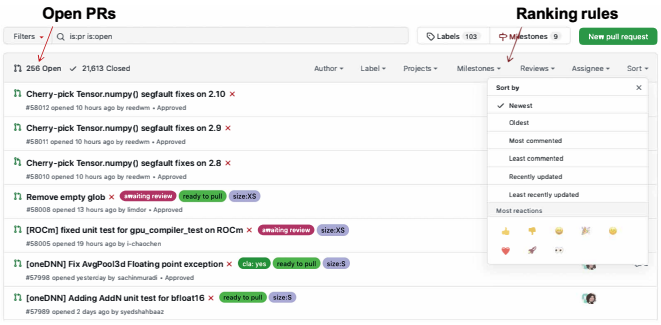

Fig. 1. A motivating example of prioritizing CRRs (TensorFlow project)

Prior studies [14, 15, 16, 17, 18] have developed a range of LtR models based on different techniques, such as Bayesian Network [19], Random Forests [20], RankBoost [21], and

ListNet [22]. As LtR models apply supervised machine learning (ML) for ranking [23], researchers have followed classic ML procedures for evaluating LtR models. Generally, the evaluation consists in measuring the similarity between two ordered lists of CRRs: one representing the ground truth (*i.e.*, the "ideal" review orders predefined by reviewers, rather than the historical review orders) and the other representing the output of a LtR model. The similarity can be measured with a variety of metrics such as Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG) [23]. The more similar the two lists are, the better we consider the LtR model for prioritizing CRRs.

This kind of evaluation is performed *before* the complex interactions between CRRs and events (*e.g.*, commenting on code changes), artifacts (*e.g.*, review comments), and reviewers take place. Such a *pre-review* evaluation, together with using abstract evaluation metrics (*e.g.*, NDCG), provides limited indications on how LtR models contribute to code review. Specifically, the state of a CRR (especially its priority) changes as it crosses events in code reviews, and consequently impacts the review events, artifacts, and reviewers' behaviors in a dynamic manner. Therefore, the *pre-review* evaluation provides only an *instantaneous evaluation* of LtR models. Moreover, existing evaluation metrics mainly work for measuring the distance between the LtR models' outputs and the "ideal" orders predefined by reviewers. However, the "ideal" orders are based on reviewers' experience that may or not match the current state of code review. As a result, a *pre-review* approach may provide an *imprecise evaluation* of LtR models.

This study aims to perform a *post-review* evaluation of LtR models for prioritizing CRRs. Toward this goal, we used Discrete-Event Simulation (DES)-based Software Process Simulation Modeling (SPSM) [24] to customize a dynamic, iterative environment (*i.e.*, allowing CRRs to correlate with activities, artifacts, and reviewers in code reviews) for evaluation. Moreover, we customized three *post-review* evaluation metrics, two for measuring the completion of reviewing CRRs and one for measuring the delivery of qualified code changes at the stage of code review, for providing realistic quantitative measures of the impact of LtR models on code review.

We developed seven LtR models and compared the orders resulting from LtR models with the historical review orders (baseline) on ten OSS projects with more than 117K CRRs for experiments. The results indicate that using LtR models can effectively help to accelerate the completion of reviewing CRRs and the delivery of qualified code changes at the stage of code review. Among the seven LtR models, LambdaMART contributes more to completion, while AdaRank contributes more to delivery, in general.

With this study we make the following main contributions:

- **Empirical evidence** showing the effectiveness of using DES-based SPSM for simulating code review processes, the benefits of using LtR for prioritizing CRRs, and the specific advantages of several LtR models.
- **A novel simulation model** that provides a dynamic, iterative experimental environment for *post-review* evaluation.
- **Three new evaluation metrics** that provide realistic quantitative values for understanding how effective LtR models impact code review. Together, the ideas behind SPSM and evaluation metrics can help software organizations to evaluate a wider range of artificial intelligence-powered software techniques besides LtR models.

II. Background and Related Work

This section briefly reviews work that is related to ours.

Code review is one of the critical quality assurance processes in software development and maintenance [2, 25, 26]. However, sometimes it becomes wasteful [27]. Software researchers have worked out many intelligent techniques to improve the efficiency and effectiveness of code review, *e.g.*, recommending suitable reviewers [28, 29], and automating code review [30, 31]. Besides, prioritizing CRRs[1] as a technique to support code review at an early stage, has received increasing concern.

To select the best ranking model for prioritizing CRRs, Van Der Veen et al. [16] developed three LtR models: Random Forests, Logistic Regression, and Naïve Bayes, and evaluated them with `Precision` and `Accuracy` metrics. The results indicated that Random Forests performed best.

Zhao et al. [14] developed six LtR models: RankNet, RankBoost, Coordinate Ascent, MART, ListNet, and Random Forests and two heuristic rule-based models: First-In-First-Out and Small-Size-First. The authors used the `NDCG` metric to compare eight ranking models. The results indicated that Random Forests win the best model.

Azeem et al. [18] developed seven LtR models to support prioritizing CRRs, including Logistic Regression, Support Vector Machine, Random Forests, Decision Trees, Naïve Bayes, k-Nearest Neighbors, and XGBoost. Among them, XGBoost outperformed the others in terms of `F1-score`, `Accuracy`, `MAP`, and `Average Recall` metrics.

Saini and Britto [15] constructed five LtR models to select the most appropriate one for constructing a CRR prioritizer, including Bayesian Network, Random Forests, Gradient Boosting, Logistic Regression, and k-Nearest Neighbors. Among them, Bayesian Network worked best on `Root Mean Square Error` and `Mean Absolute Error` metrics.

Fan et al. [17] used Random Forests to build the LtR model and reinforced it with cost-sensitive learning strategies. The results indicated this model outperformed random guess as well as other baselines (Bayesian Network in Jeong et al. [32], Random Forests in Gousios et al. [3]) in terms of `AUC`, `Precision`, `Recall`, and `F1-score` metrics. These four metrics together with `Cost-effectiveness` were adopted by Islam et al. [33]. The authors developed six LtR models, including LightGBM, DNN, Random Forests, GradientBoosting (GBT), ExtraTrees, and Logistic Regression. The results indicated that LightGBM occupied the leading position.

Overall, prior studies have developed many LtR models to prioritize CRRs and employed very rich evaluation metrics.

[1]As most studies do not specify the stage of PRs and the beneficiaries (reviewer or integrator), this section does not distinguish CRRs and PRs.

However, there are no commonly accepted evaluation metrics or best LtR models so far. Moreover, they rarely considered the dynamic, iterative code review processes. The evaluation was performed in a *pre-review* style, *i.e.*, LtR models were not correlated with the complex activities, artifacts, and reviewers in code reviews. Therefore, the evaluation metrics can be immediately calculated before performing code reviews. Due to this, developers are not clear exactly how effective LtR models contribute to code review.

III. Preliminaries

This section outlines (1) *learning-to-rank*, the core technique for developing evaluation subjects; and (2) *software process simulation modeling*, the core technique for developing the environment for *post-review* evaluation.

A. Learning-to-Rank

Learning-to-Rank (LtR) refers to applying supervised machine learning techniques for training models in a ranking task [23], which has been widely employed in the information retrieval field. In LtR, a system maintains a collection of examples, and when a query is submitted, the system retrieves examples associated with the query from its collection, ranks examples, and returns top examples. When developing datasets for training LtR models, each query is associated with several examples and their relevance (*i.e.*, label) to the query. Given the context of prioritizing CRRs, the query could be "*which CRRs are most likely to be merged?*", "*which CRRs are easiest to review?*", etc. Once the query is clear, we can focus on training LtR models and testing/evaluating them.

There are three main learning strategies in LtR: pointwise, pairwise, and listwise. Pointwise LtR models transform the ranking problem into classification, regression, or ordinal regression. Pairwise LtR models take example pairs as instances in learning and formalize the ranking problem as that of classification or regression. Finally, listwise LtR models determine the optimal ordering of an entire list of examples [22, 23]. Besides constructing the LtR models investigated in the prior related studies, we develop several classic LtR models to ensure that each of the three learning strategies is investigated in this study.

In this study, the goal of LtR models is to assign priority to each CRR and output the optimal ordering of the entire list of CRRs. We are particularly interested in (1) whether there is a difference between prioritization and non-prioritization, and, (2) if differences exist, whether there are differences among multiple LtR models.

B. Software Process Simulation Modeling

A software process is defined as a set of activities, methods, practices, and transformations that people use to develop and maintain software and associated artifacts [34]. A model is an abstraction of a real or conceptual complex system [35]. Software Process Simulation Modeling (SPSM) is a set of modeling techniques that imitate behaviors of real-world software processes [24]. Kellner et al. [35] summarized six main purposes of using SPSM: (1) strategic management, (2) planning, (3) control & operational management, (4) process improvement & technology adoption, (5) understanding, and (6) training & learning. SPSM has now been employed to address various issues in software development [36, 37, 38, 39].

There are three main simulation paradigms in SPSM: Discrete-Event Simulation (DES), System Dynamics (SD), and Agent-Based Simulation (ABS). DES simulates a system/process as a sequence of discrete events that occur over time. SD focuses on simulating the nonlinear behavior of a dynamic system rather than the fine details. SD simulates a dynamic system at a high level of abstraction and in continuous time. While ABS simulates a system by its individual active agents that interact with each other [35, 38].

A code review process consists of a series of time-varying sub-processes (events) in which CRRs pass through each subprocess sequentially and interact with various events (*e.g.*, commenting), artifacts (*e.g.*, comments), and reviewers simultaneously [4]. DES is a good fit for simulating such a process as it assumes few changes between events (the events in a code review process are clear). In addition, DES has the capability of modeling events in a process at a relatively low or medium level of abstraction, providing rich quantitative and qualitative feedback for observing simulation details. More importantly, DES can distinguish each simulated object (CRR in this study) with extensive attributes, while the other two paradigms cannot do so [35, 38]. Therefore, we use DES-based SPSM to simulate code review processes.

IV. The Process Simulation Model for Evaluation

This section presents the research methods and procedures for developing the process simulation model, which comprises three steps: (1) review process analysis; (2) descriptive model construction; and (3) simulation model construction.

A. Descriptive Model

As a first step, we investigated code review processes by analyzing GitHub documents and then interviewing five developers with code review experience from large software companies. Specifically, we drafted a GitHub-style code review flowchart and presented it to each developer individually. We revised the flowchart based on their feedback to output a widely accepted descriptive model.

We found there are two main roles in code reviews: code authors and reviewers. Code authors fork the main branch of a project and make local changes such as adding new features, fixing bugs, and refactoring; then commit and submit CRRs to the project. Code reviewers pick up CRRs from the waiting list and comment on the CRRs when defects are found. Code authors fix defects according to review comments and resubmit CRRs. When no further defects are found, the CRRs are qualified to be merged into the main branch.

Then we customized the prioritization-specific code review processes as the focus of this study is to evaluate LtR models in the context of prioritizing CRRs. Specifically, we divided the prioritization-specific processes into eight main

events, which together constitute the descriptive model. The descriptive model is an abstraction of real-world situations that can be used to explain prioritizing CRRs and to construct the simulation model. Figure 2 shows an overview of the descriptive model, where the eight main events are labeled E1-E8, respectively. Compared with general code review processes in the real-world situation, we particularly customized a "prioritization" part (E2) and its implementation—"LtR" (E3).

The customized code review processes are as follows: Code authors create and submit CRRs (E1); code reviewers decide whether to automatically prioritize CRRs (E2); if prioritized, reviewers enable LtR models to obtain an ordered list of CRRs (E3); otherwise, reviewers personally decide the review order of CRRs (E4); reviewers check their daily workload to decide whether to continue reviews (E5); if continue, reviewers examine a CRR and leave comments if needed (E6); after then, reviewers check whether the CRR needs further review (E7); finally, integrators decide whether to merge the CRR and then close it (E8).

B. Simulation Model

1) Basic Assumptions: To better simulate real-world situations while developing a customized experimental environment for evaluating LtR models in the context of prioritizing CRRs, we propose three main assumptions behind the simulation model. The first assumption is that prioritizing CRRs has no impact on the outcomes (*e.g.*, *number of review times* and *acceptance*) of a CRR. According to our interviews, such outcomes depend heavily on the code changes and the authorship of a CRR. Therefore, it is possible to evaluate the impact of LtR models from a process perspective. The second assumption is that a review team stops reviewing CRRs once it reaches its daily workload (so there are open CRRs in the project repository) and the team's daily workload can be measured by its total number of review times for CRRs. Therefore, the *number of open CRRs* could be used to calibrate the parameters and variables of our simulation model as well as to verify it. The third assumption is that the distribution of the datasets rarely changes [40] when adjusting the order of adjacent CRRs; otherwise, the performance of LtR models is easily affected. Therefore, we could re-train LtR models after a certain amount of time (*e.g.*, 24 hours) because real-time training of LtR models within the simulation model is highly time- and cost-consuming.

2) Model Construction: We employed DES-based SPSM with AnyLogic [41] which is a popular simulation modeling software (specifically, its process modeling library), to transfer the descriptive model into an executable simulation model. The process modeling library provides rich components, which can be used to describe a series of discrete events over time. Figure 3 shows an overview of the simulation model. For descriptions of the main components used, please refer to the Library Reference Guides [42]. Note that some components are set for implementation convenience in AnyLogic, *e.g.*, "Ordered_List2" is set to cache "Ordered_List1".

We present the major events in the customized code review processes by elaborating on the mappings between the descriptive model (cf. Figure 2) and the simulation model (cf. Figure 3).

E1: Start. The E1 simulates the event of creating and submitting CRRs. The internal system time of the model is set to the start of the historical time span (01/06/2021 – 15/06/2022), and then the model starts to run. When each CRR is submitted, a delay time is set with it for simulating the time spent on creating and submitting. The simulation model assigns each CRR's attributes with a set of parameters and variables (cf. Table I) based on historical data such as the number of review times, merge outcome, and so on. When E1 occurs for a while, the model has accumulated some ordered (based on the submission time) CRRs that are awaiting prioritization (if E3 triggers) and review. For the sake of practical application (*e.g.*, re-training LtR models once a day), the simulation unit is set to one day; and for the sake of succinctness (*e.g.*, ignoring discrete changes within a CRR), the review unit is set to one CRR.

E2: Switching to Prioritize CRRs. The E2 simulates the event of the switching to prioritize CRRs. If the prioritization switch is turned on, then the E3 (Learning to Rank) is triggered; otherwise, the E4 (Human Decision) is triggered.

E3: Learning to Rank CRRs. The E3 simulates the event of re-ranking the list of CRRs by employing LtR. Note that the priority of each CRR is definite in one round of review, referring to leaving a review comment or closing without any review comments, and the priority of a CRR can be adjusted in the next round of review (if needed). The E3 outputs an "ideal" optimal ordering of the entire list of CRRs, *i.e.*, the most promising and urgent CRRs have the highest priority (labeled with "Ordered List" in Figure 2). In the E3, the simulation model directly leverages the outputs of LtR models to accelerate the process of prioritizing CRRs. Note that the processes of training and testing LtR models are not described in the descriptive and simulation model. The role of LtR models, in this case, is merely to provide an ordered list. Since LtR is a supervised learning task, we used labeled CRRs to train LtR models. For details on the datasets and training settings, please refer to Section V-B.

E4: Human Decision on CRRs. The E4 simulates the event of re-ranking the list of CRRs based on human decisions. In reality, the waiting list of CRRs is sent directly to code reviewers if the prioritization switch is off. While in the simulation model, the waiting list of CRRs (ordered by the historical submission time) is re-ranked by querying the historical review time of each CRR (labeled "Ordinary List" in Figure 2). Technically, the model simulates human decisions on prioritization by querying historical time.

E5: Checking Review Team's Workload. The E5 simulates the event of whether to continue to review CRRs. Before reviewing CRRs, the simulation model checks whether the review team's workload remains. If the workload remains, then the E6 (Code Review) is triggered; otherwise, the E2 (Switching to prioritize CRRs) is triggered. The daily workload of a

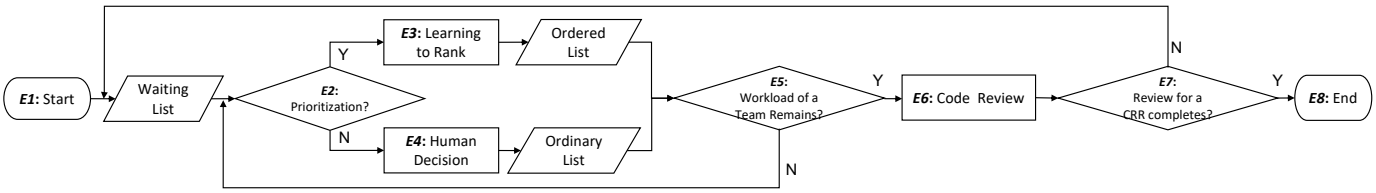

Fig. 2. An overview of the descriptive model

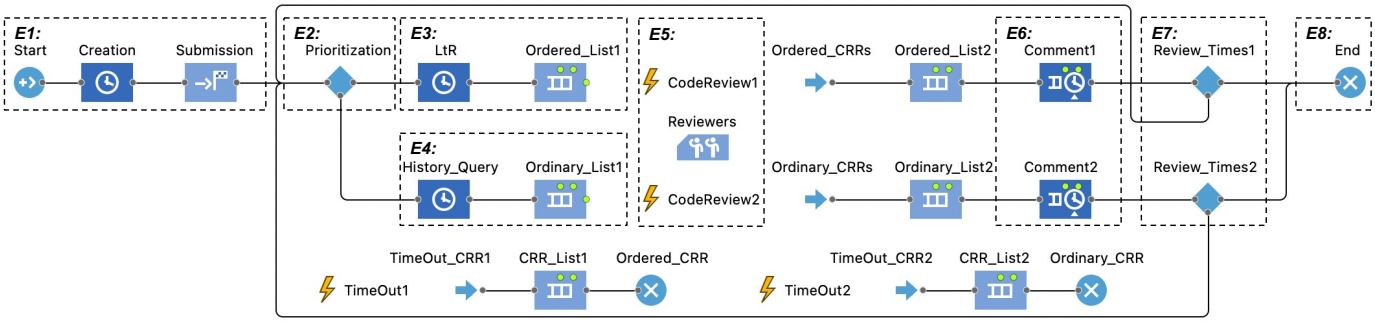

Fig. 3. An overview of the simulation model (implemented with AnyLogic 8.8.0)

review team is measured with the "number of review times", which can be accessed from the historical data. Please refer to E7 for the measure of "number of review times". The review team stops working when it has completed its daily workload, and the team continues to work the next day.

E6: Code Review. The E6 simulates the event of picking up and reviewing CRRs. In this event, code reviewers regularly pick up a CRR from the top of the list of CRRs ("Ordered List" or "Ordinary List"). When there are defects or issues that are unclear, reviewers leave comments for specific lines of code changes or the whole file. For different types of CRRs (*e.g.*, *adding features* and *fixing bugs*), the time spent on reviewing is unequal. We followed the code review experience at one of the global Information and Communication Technology (ICT) enterprises, and set six types of CRRs in the simulation model based on the keywords occurring in the descriptions or function names of a CRR, specifically, "feature", "bug/test", "refactor/improve", "documents". The time spent on reviewing CRRs is set as "feature"–10 minutes, "bug/test"–5 minutes, "refactor/improve"–15 minutes, "documents"–5 minutes, none (*i.e.*, without occurring any predefined keywords)–10 minutes, mix (*i.e.*, occurring multiple predefined keywords)–15 minutes. Moreover, the simulation model does not personalize each code reviewer, but rather considers them as a whole team.

E7: Checking CRR's Completion. In some cases, a CRR receives several rounds of feedback (review comments) or receives no feedback at all before closing. The E7 simulates the event of re-commenting on a CRR (if needed). The number of review times is defined for a CRR, and it is measured with the "number of review comments plus 1". Despite being checked by reviewers, the integrator still makes the final check on a CRR. The '1' accordingly is set to simulate this case. On the other hand, the '1' works for the case in which a CRR receives no feedback but is closed, *i.e.*, merged or rejected. The number of review times for a CRR can be accessed from the historical data. In the simulation model, when a CRR receives a comment, its number of review times is reduced by 1. Therefore, the judging condition is set as "number of review times > 0". If the judging condition is true, the CRR is sent back to the waiting list to be prioritized (if needed) and reviewed; otherwise, the E8 is triggered.

E8: End. The E8 simulates the event of closing (*i.e.*, merging or rejecting) a CRR. This study does not consider the "reopen" situation as it rarely appears [43]. Therefore, a CRR has reached the end of its life cycle in the simulation model after E8. In the E8, those CRRs that have not yet been closed (*i.e.*, remaining in the simulation model) on the end day (15/06/2022) of simulation are either sent to the "Ordered List" or the "Ordinary List" according to their previous events, to ensure that the number of exit CRRs is equal to the number of entry CRRs (as shown at the bottom of Figure 3).

3) Model Calibration: It is critical to assume that parameters and variables follow probability distributions in order to simulate real-world processes. In this study, however, we did not rigorously require such an assumption and corresponding calibration procedures since we utilized the historical data to develop a realistic evaluation environment. The workload of a review team is one of the most important variables of the simulation model. The lack of recording makes modeling this part the most challenging. We assumed using a given number of CRRs or a given duration of time spent at first. By employing such settings, we found very high accumulations of CRRs, which was clearly contrary to the real-world situation. After interviewing developers with this issue, we used the historical "number of review times" to measure a review team's workload as it dynamically reflects the real-world situation on a daily basis.

Table I presents the main parameters and variables of the simulation model. Parameters are mainly used to describe the static characteristics of the modeled objects; while variables are mainly used to describe the dynamic state of the modeled objects or store results.

TABLE I
DESCRIPTIONS OF MAIN PARAMETERS AND VARIABLES

Parameter	Description
id	The identifier of a CRR.
type	The type of a CRR. It is characterized by the keywords occurring in the description.
submission_time	The time of submitting a CRR.
review_time	The time spent reviewing a specific type of CRR.
comments_num1	The number of historical review comments of a CRR.
comment_time1	The date list of commenting on a CRR. It is only used to re-rank the historical waiting list of CRRs (to model E4).
merge_time	The time of merging a CRR. It is only used to re-rank the historical waiting list of CRRs (to model E4).
close_time1	The time of closing a CRR. It is only used to re-rank the historical waiting list of CRRs (to model E4).
is_merged	Whether a CRR is merged into the main branch?
switchOn	Whether enabling prioritizing CRRs?
submission_delay	The delay time of submitting a CRR.
prioritization_delay	The delay time of prioritizing CRRs.
transfer_delay	The delay time of transferring CRRs to the waiting list.
Variable	**Description**
priority	The priority score of a CRR. It is assigned by the LtR model.
comments_num2	The number of review comments of a CRR in the simulation model.
comment_time2	The time of commenting on a CRR in the simulation model.
close_time2	The time of closing a CRR in the simulation model.
day_num	The time span of the simulation. It is measured by the number of days.
workload	The number of review times that a review team undertakes in a day.
merge_num	The number of CRRs merged in a day.
review_num	The number of CRRs reviewed in a day.
review_time_spent	The time spent on reviewing CRRs of a day.
open_num	The number of open CRRs when starting to perform code reviews in a day.
open_time_spent	The time spent on reviewing all the open CRRs in a day.

4) Model Verification and Validation: Verification and validation are critical procedures in SPSM to secure the quality and credibility of a model [44]. Following Lindland et al.'s [45] and Ahmed et al.'s [46] framework for understanding quality in conceptual modeling, we verified and validated the simulation model by considering three major aspects of quality.

Syntactic quality concerns the syntactic correctness of the model corresponding to modeling grammars. Toward this goal, two researchers independently checked the parameters and variables of each component, links between components (*e.g.*, condition branches and loops), and the model meets the syntax defined in the simulation software—AnyLogic.

Semantic quality concerns the *feasible validity* and *feasible completeness* of the model. Four researchers first confirmed the structure of the DES-based simulation model is consistent with the descriptive model. Specifically, for *feasible validity*, four researchers checked the model's settings that may be invalid but should not be eliminated (*e.g.*, CRR types and time spent). For *feasible completeness*, four researchers checked the completeness of the model and removed duplicate settings.

Pragmatic quality concerns the *comprehension* and *understandability* of the model. The former is related to audience groups and the latter is related to models. This aspect requires a model to be easily understood and manipulated by users. Especially for the former, we first invited dozens of postgraduate students who have little knowledge of SPSM and three doctoral students who have at least five years of experience in SPSM to check and correct our descriptive model and simulation model. Then, we invited four developers with knowledge and experience in SPSM from the software development division at an ICT enterprise to further verify and validate our models.

It is worth reporting on one of the measures we have taken to test the validity of our simulation model. As one of the objectives of this study is to investigate the differences resulting from prioritization; therefore, the historical review orders ("Ordinary List") rather than the historical submission orders were expected to be baselines. Figure 4 shows an example of measuring the simulation biases on the TensorFlow project. The dependent/observed variable is the number of open CRRs. Specifically, "Real-world" is the number of open CRRs accessed from history. While "Simulated" is the number of open CRRs resulting from "Ordinary List", which is subject to many settings of the simulation model, *e.g.*, the "workload" of a review team. Intuitively, although there are some fluctuations, the trends of the two curves are generally similar. We used the Mean Magnitude of Relative Error (MMRE) [47] to measure the distance between the two sets of variables. The MMRE for simulating the TensorFlow project is 0.177 (the complete data items are available in the online replication package [48]), which quantitatively validates the reasonableness of our settings. Accordingly, the historical review orders are adopted as baselines.

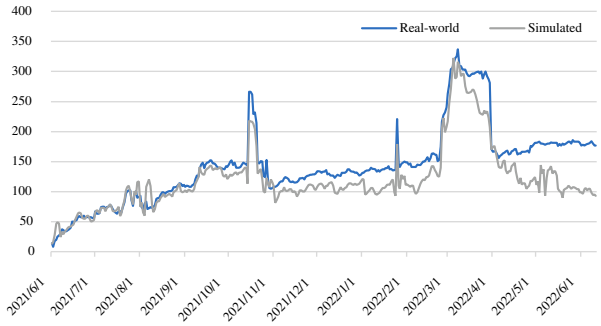

Fig. 4. A comparison of the number of open CRRs from the real-world situation and simulation model for the TensorFlow project

V. EXPERIMENTS

This section reports on the experimental designs and results.

A. Research Questions

Given a definite workload of a review team that is measured by the number of daily review times, two Research Questions (RQs) are proposed to guide experiments.

- **RQ1:** How effective are LtR models in accelerating the completion of reviewing CRRs?
- **RQ2:** How effective are LtR models in accelerating the delivery of qualified code changes?

B. Experimental Designs

1) Project Selection: Table II outlines the demographics of ten OSS projects selected for experiments. The selection criteria are as follows: (1) has been investigated in the prior related studies (*i.e.*, [14, 16, 17, 18]); (2) has more than 40% merge rate (to exclude projects adopting non-PbD model) [14]; (3) has accumulated more than 2000 CRRs (to exclude small-scale, inactive projects) as we found the numbers of CRRs

of the majority of projects exceed 2000 after meeting the previous two criteria. Moreover, the projects should be diverse in terms of project domains, programming languages, etc. With these measures, we expect to ensure the projects' suitability for experiments.

TABLE II
DEMOGRAPHICS OF THE SELECTED PROJECTS

Project	Time span	#CRR	#Merge	#Author	#Reviewer
Katello (P1)	04/2012–06/2022	9852	8845	199	127
Kuma (P2)	02/2011–06/2022	6687	5739	373	116
Moby (P3)	02/2013–06/2022	21795	17057	2362	1168
OpenCV (P4)	07/2012–06/2022	13108	10580	1898	665
React (P5)	05/2013–06/2022	12289	8329	1725	958
scikit-learn (P6)	09/2010–06/2022	13626	9139	2624	1614
TensorFlow (P7)	11/2015–06/2022	21164	14650	4016	1547
Terraform (P8)	07/2014–06/2022	12258	10011	1992	575
NetBeans (P9)	09/2017–06/2022	3875	3224	215	139
Phoenix (P10)	01/2014–06/2022	2661	2065	1129	267
Total	–	117315	89639	16533	7176

* Data collection ends on 15/06/2022.

2) Priority Setting: After experimental projects are selected, the next step is assigning each CRR a priority score to build ground truth datasets for training LtR models. Unfortunately, so far there is no commonly agreed definition of CRRs' priority. In this study, we measured the priority of a CRR from two perspectives: *outcome* and *urgency*. An urgent CRR that also has a positive outcome should be given high priority. Specifically, we characterized the *outcome* of a CRR with its acceptance on merging into the main branch, *i.e.*, *accept* or *reject* [49, 50, 51]; we characterized the *urgency* of a CRR with the time waiting for the first round of review feedback that was inspired by a large number of prior related studies [52, 53, 54, 55, 56]. The sub-label *outcome* is quantified with two values: 4–*accept*, and 0–*reject*; the sub-label *urgency* is quantified with five values: 0–*very slow*, 1–*slow*, 2–*medium*, 3–*fast*, and 4–*very fast* through equally dividing the time span for receiving the first feedback in historical data for a project. The ultimate label (*i.e.*, relevance with respect to the unique query) of a CRR is the sum of two scores, which was inspired by prior studies [18].

3) LtR Models and Features: In the simulation model, each CRR is assigned a priority score by calling the pre-trained LtR models. Specifically, we trained the following seven LtR models. Two of them (*i.e.*, Random Forests and Bayesian Network) have been intensively studied [14, 16, 17, 18], others are popular in various tasks in the field of information retrieval, thus serving as experimental subjects in this study.

- **Bayesian Network (BN)** [19] is a probabilistic graphical model that leverages Bayesian inference on a directed acyclic graph to model causal relationships among a set of variables. It is implemented in a pointwise style.
- **Random Forests (RF)** [20] is an ensemble of multiple decision tree models, each of which is independent of the others. It is implemented in a pointwise style.
- **RankBoost** [21] is a pairwise LtR model. It uses the boost strategy to iteratively create and aggregate a group of "weak rankers" to build a ranking model.
- **AdaRank** [57] is a listwise LtR model. It also iteratively constructs "weak rankers" as RankBoost, but its loss function is directly defined on performance measures.
- **Coordinate Ascent (CA)** [58] is a listwise LtR model. It differs from other LtR models by employing "coordinate ascent" which iteratively optimizes the objective function by solving a series of one-dimensional searches.
- **ListNet** [22] is a listwise LtR model. It learns the joint probability distribution via neural network and gradient descent for a ranking list rather than the point or pairwise probability of examples.
- **LambdaMART** [59] is a listwise LtR model. It is a combination of LambdaRank [60] and MART (Multiple Additive Regression Tree) [61].

Table III presents the extracted features for training LtR models. Considering LtR models assign a CRR with a priority score when it remains on the waiting list and has not been reviewed in the simulation model. Therefore, we did not consider features related to review processes when training LtR models. Moreover, we did not consider features related to the quality of code changes due to the lack of evidence indicating a correlation between code quality and acceptance [62]. Despite this, Table III aggregates a wide range of features, such as source code, project information, and author experience.

We collected the experimental datasets through the GitHub REST API [63] and developed LtR models with the RankLib library [64]. Then, we performed the min-max normalization to mitigate the impact of data scales. Finally, we trained the LtR model for each project. The time span of the training dataset is from each project's start time to 31/05/2021, and the time span of the testing dataset is 01/06/2021 – 15/06/2022.

TABLE III
FEATURES EXTRACTED FROM CODE REVIEW REQUESTS

Feature	Description	Ref.
is_merged	Whether the patch is merged?	[18]
assignees	The number of assignees (*i.e.*, "@xxx").	[18]
has_test	Whether the function name contains the word "test"?	[16]
has_bug	Whether the description contains a "bug"?	[17]
has_feature	Whether the description contains a "feature"?	[17]
has_improve	Whether the description contains a "improve"?	[17]
has_document	Whether the description contains a "document"?	[17]
has_refactor	Whether the description contains a "refactor"?	[17]
directories	The number of directories modified.	[17]
subsystems	The number of subsystems modified.	[17]
language_types	The number of programming language types used.	[17]
file_types	The number of file types in the CRR.	[17]
lines_added	The number of lines of code added.	[17]
lines_deleted	The number of lines of code deleted.	[17]
segs_added	The number of segments of code added.	[17]
segs_deleted	The number of segments of code deleted.	[17]
segs_changed	The number of segments of code changed.	[17]
files_added	The number of files added.	[17]
files_deleted	The number of files deleted.	[17]
files_changed	The number of files changed.	[17]
file_developer	The number of developers who changed files.	[17]
change_num	The average number of commits of a CRR submitted by the author.	[17]
files_modified	The number of times files were modified before.	[17]
is_core_member	Whether the author is a core member of the project?	[65]
commits	The number of commits.	[65]
labels	The number of labels affiliated.	[65]
prev_CRRs	The number of CRRs created by the author.	[65]
title_words	The number of words in the title.	[65]
body_words	The number of words in the body.	[65]
commits_average	The average number of commits per CRR.	[65]

4) Evaluation Metrics: As the *post-review* evaluation aims to explain to developers how effectively LtR models accelerate reviewing CRRs and delivering qualified code changes, the commonly used metrics such as MAP and NDCG are no longer appropriate due to ignoring review outcomes. Therefore, we customized the following evaluation metrics.

- **Completion** (RQ1)

$$E_{c1} = \frac{1}{K}\sum_{i=1}^{K}\frac{N_{ir}}{N_{io}} \tag{1}$$

where, K is the number of days simulated, N_{ir} and N_{io} are the number of reviewed CRRs (is recorded after the simulation ends) and the number of open CRRs (is recorded before the simulation starts) on the i^{th} day, respectively.

$$E_{c2} = \frac{1}{K}\sum_{i=1}^{K}\frac{T_{ir}}{T_{io}} \tag{2}$$

where, T_{ir} and T_{io} are the time spent on the reviewed CRRs and the time spent (should) on the open CRRs on the i^{th} day, respectively.

- **Delivery** (RQ2)

$$E_{d} = \frac{1}{K}\sum_{i=1}^{K}\frac{N_{im}}{N_{ir}} \tag{3}$$

where, N_{im} and N_{ir} are the number of merged CRRs and the number of reviewed CRRs on the i^{th} day, respectively. E_{c1}, E_{c2}, and E_d are percentages from 0% to 100%.

C. Results and Analysis

Table IV summarizes the performance of each LtR model for prioritizing CRRs (the best results are highlighted in bold).

1) Answer to RQ1: We first focus on each LtR model's contribution to accelerating the completion of reviewing CRRs. When it comes to the completion-specific metric E_{c1}, *i.e.*, the ratio of the number of reviewed CRRs to the number of open CRRs, LambdaMART has an average E_{c1} of 0.132 on ten projects, which occupies the first among seven models as well as significantly higher than the real-world situation (0.075). Followed by BN and RankBoost, their performances in terms of E_{c1} are 0.120 and 0.115, respectively. RF and ListNet performed the worst whose E_{c1} are 0.071 and 0.074, respectively, which is slightly lower than the real-world situation. Such a revelation occurs in the second completion-specific metric E_{c2}, *i.e.*, the ratio of the time spent on the reviewed CRRs to the time spent on the open CRRs, the top-3 well-perform LtR models are LambdaMART (0.127), BN (0.120), and RankBoost (0.114), the worst-perform LtR model is RF whose E_{c2} is merely 0.069. RF is the only LtR model with E_{c2} lower than the real-world situation (0.073). In summary, LambdaMART leads both metrics across four projects; BN dominates in two projects; ListNet and RankBoost each work best in one project.

2) Answer to RQ2: AdaRank achieves the leading average performance (0.472) with regard to the delivery-metric E_d (*i.e.*, the ratio of the number of merged CRRs to the number of reviewed CRRs) among seven LtR models and the real-world situation on ten projects. Followed by CA and RF, their average performances on ten projects in terms of E_d are 0.453 and 0.438, respectively. ListNet and RankBoost are in the last two places whose performances in terms of E_d are 0.424 and 0.431, respectively, falling behind the real-world situation (0.432). AdaRank holds a leading position on seven projects, and RF and CA occupy one project, respectively. In particular, none of the LtR models perform better than the real-world situation (0.545) on the project 'Katello' (P1). The only close performance results from RF (0.544), while the others are far behind it. Though AdaRank performs best in most projects, the E_d of AdaRank is merely 0.479 in this case. In a nutshell, among the seven LtR models, AdaRank has the highest effectiveness in accelerating code delivery on ten projects, while the others differ little in terms of the delivery-specific metric E_d.

By answering two RQs, we find that (1) applying LtR effectively helps to accelerate the completion of reviewing CRRs and the delivery of qualified code changes at the stage of code review; (2) the performances vary among different LtR models. However, it is worth noting that either for the completion-specific metrics (E_{c1} and E_{c2}) or the delivery-specific metric (E_d), there are a few cases in which the performances of LtR models are inferior to that of the real-world situation. Although LtR can facilitate code review, it still needs to reinforce LtR models carefully.

VI. Discussion

This section discusses the considerations and implications of the process simulation model.

A. Considerations for Historical Data

1) Historical Data for Simulating Review Processes: We considered the following aspects when leveraging historical data for simulating code review processes. First, though simulation software (*e.g.*, AnyLogic) can generate CRRs and other types of data, we made full use of the majority of historical data because it reflects the most realistic situations such as the submission time, the waiting time, the type of CRRs, and the workload of review teams We carefully minimized the number of assumptions and estimations to reflect real-world code review processes, in order to provide a realistic evaluation environment. Second, some historical data may not be recorded or exist in project repositories (*e.g.*, types of CRRs, time spent reviewing CRRs, and review expertise). In this case, we have to make several assumptions and estimations to ensure the integrity of the code review processes and the rigor of the simulation model. Finally, there are several historical data that reflect reality but are not ideal as ground truth for training models, and therefore require adjustment. For instance, the code reviewers assigned for CRRs are not appropriate [66, 67]. Therefore, we treated the reviewers as a team, without differentiating between individuals.

2) Historical Data for Training LtR Models: The historical orders of CRRs that had been reviewed may not be an appropriate benchmark for training LtR models. For example, code reviewers randomly pick up CRRs or/and carefully select

TABLE IV
PERFORMANCE SUMMARY OF EACH LTR MODEL (*post-review* EVALUATION)

	Non-Prioritization			**Prioritization**																			
	Reality			**BN**			**RF**			**CA**			**RankBoost**			**ListNet**			**AdaRank**			**LambdaMART**		
	E_{c1}	E_{c2}	E_d	E_{c1}	E_{c2}	E_d	E_{c1}	E_{c2}	E_d	E_{c1}	E_{c2}	E_d	E_{c1}	E_{c2}	E_d	E_{c1}	E_{c2}	E_d	E_{c1}	E_{c2}	E_d	E_{c1}	E_{c2}	E_d
P1	.116	.105	**.545**	.145	.144	.525	.095	.082	.544	.089	.081	.507	.136	.135	.520	.066	.058	.492	.133	.137	.479	**.176**	**.171**	.517
P2	.022	.021	.097	.030	.030	.118	.016	.014	**.124**	.034	.037	.120	.020	.020	.103	**.045**	**.047**	.113	.020	.021	.113	.022	.022	.094
P3	.044	.046	.376	**.078**	.079	.403	.030	.029	.367	.076	**.080**	.455	.077	.077	.380	.051	.050	.392	.064	.073	**.463**	.071	.066	.409
P4	.071	.064	.546	.129	.121	.528	.131	.128	.580	.095	.086	.577	**.168**	**.156**	.526	.055	.050	.466	.042	.037	**.606**	.100	.091	.509
P5	.084	.085	.373	**.152**	**.160**	.398	.074	.071	.381	.100	.088	.359	.131	.132	.379	.131	.116	.367	.103	.094	**.405**	.148	.149	.386
P6	.074	.072	.679	.118	.115	.660	.061	.062	.676	.132	.132	.687	.112	.110	.685	.071	.063	.660	.118	.117	**.714**	**.222**	**.199**	.694
P7	.076	.076	.490	.195	**.196**	.490	.061	.061	.479	.039	.038	.502	**.197**	.194	.482	.068	.066	.492	.093	.094	**.587**	.190	.184	.498
P8	.057	.059	.380	.070	.070	.399	.044	.043	.401	.053	.053	.424	.069	.071	.417	.056	.054	.404	.053	.054	**.449**	**.097**	**.097**	.388
P9	.113	.107	.578	**.165**	**.165**	.601	.078	.082	.580	.159	.150	.629	.151	.154	.582	.102	.102	.598	.121	.116	**.634**	**.165**	**.165**	.601
P10	.095	.090	.251	.116	.118	.240	.124	**.122**	.249	.121	.116	**.274**	.094	.093	.239	.090	.095	.252	.074	.075	.266	**.124**	.120	.252
Avg	.075	.073	.432	.120	.120	.436	.071	.069	.438	.090	.086	.453	.115	.114	.431	.074	.070	.424	.082	.082	**.472**	**.132**	**.127**	.435
Std	.028	.025	.163	.047	.047	.155	.036	.035	.159	.039	.036	.161	.049	.048	.160	**.026**	**.023**	**.151**	.036	.035	.171	.057	.054	.163

* The higher the E_{c1}, E_{c2}, and E_d of the model, the better of the LtR model is.

CRRs by following specific rules (*e.g.*, First-In-First-Out and Small-Size-First). As a result, the promising and urgent CRRs are delayed. In this study, we re-rank the historical list of CRRs according to the premise that "the most promising and urgent CRRs should be placed at the top of the list" to train LtR models. It is worth noting that the testing/evaluation of LtR models was performed with the customized DES-based SPSM environment (*post-review* evaluation) rather than using the refined datasets (*pre-review* evaluation).

B. Considerations for Process Simulation

Leveraging LtR models for prioritizing CRRs is an important measure to secure the efficiency and effectiveness of code review at an early stage. While prior studies have primarily employed *pre-review* evaluation to select appropriate models, this study employs *post-review* evaluation using SPSM. The main reasons are that although LtR models can provide an ordered list of CRRs that should be close to the pre-defined "ideal" list, they cannot test the extent to which the "ideal" output influences code review. Code review is a dynamic, iterative software process, *e.g.*, a CRR may be commented on more than one time by more than one reviewer. The state of a CRR is subject not only to its original priority assigned by LtR models, but also to the specific code review process in which it is positioned. We have conducted a *pre-review* evaluation on seven LtR models with three classic evaluation metrics (*i.e.*, NDCG, MRR, and Kendall's Tau [23]). The detailed results are shown in Table V. Random Forests outperformed the other models among all the three evaluation metrics in the *pre-review* evaluation; while LambdaMART and AdaRank performed well in the *post-review* evaluation but performed poorly in the *pre-review* evaluation. These differences confirm the necessity of using SPSM to evaluate LtR models in the context of prioritizing CRRs.

Moreover, since the actions and decisions made at one point in the process impact others in complex or indirect ways and must be accounted for, SPSM provides feedback mechanisms for investigating "what if/whether"-style and unexpected questions. With SPSM, software organizations can better understand prioritizing CRRs and test a setting in a more realistic scenario, thereby reducing costs and risks. Specifically, many problems remain to be explored in this study, such as (1) what if changing the settings of types of CRRs and time spent? (2) what if considering the experience and expertise of code reviewers (*i.e.*, distinguishing between each reviewer)?

We would never recommend replacing the *pre-review* evaluation with a *post-review* evaluation. Instead, the *post-review* evaluation could be used as a measure to assist the *pre-review* evaluation. For instance, the *post-review* evaluation metrics could be used to guide the priority settings of CRR examples (*e.g.*, giving higher weights to merged CRRs) and the training of LtR models (*e.g.*, using a cost-sensitive learning strategy). The ideal LtR models are expected to perform well in two types of evaluation.

C. Beyond Evaluating LtR Models

Artificial Intelligence (AI) has rapidly evolved over the past few years, along with an increasing number of its applications in software engineering [68]. However, AI-powered software techniques are at risk of being untrustworthy [69]. One of the major reasons is the weak explainability and interpretability of their core component—AI algorithms [70]. Researchers have developed methods for explaining and interpreting the "black box" AI algorithms [71]; however, these methods are difficult to understand for non-AI professionals. In addition to confusion regarding the mechanism of AI algorithms, there is uncertainty in software organizations about what impact they will have (*e.g.*, prioritizing CRRs with LtR on code review). Therefore, AI-powered techniques are applied in reality with great caution. This paper reports a case study of adopting SPSM to evaluate LtR models in the context of prioritizing CRRs. Compared with existing evaluation manners that focus only on the ranking task itself, SPSM allows AI-powered techniques (*e.g.*, LtR) to correlate with complex activities (*e.g.*, prioritization and commenting), artifacts (*e.g.*, review comments), and professionals (*e.g.*, code reviewers) in dynamic, iterative processes (*e.g.*, code review). With SPSM, software organizations are able to elaborately test an AI-powered technique and its settings from a broad range of feedback (*e.g.*, the acceleration of the completion of reviewing CRRs and the delivery of qualified code changes). Finally, an AI-powered technique can be safely implemented in reality.

TABLE V
PERFORMANCE SUMMARY OF EACH LTR MODEL (*pre-review* EVALUATION)

	BN			RF			CA			RankBoost			ListNet			AdaRank			LambdaMART		
	NDCG*	MRR	KT	NDCG	MRR	KT	NDCG	MRR	KT	NDCG	MRR	KT	NDCG	MRR	KT	NDCG	MRR	KT	NDCG	MRR	KT
P1	.670	.122	.392	.771	.164	**.295**	**.766**	**.172**	.344	.697	.098	.447	.755	.153	.315	.627	.083	.513	.674	.123	.444
P2	.691	.279	.334	**.751**	**.393**	.319	.620	.213	.352	.599	.182	.386	.742	.331	**.303**	.665	.266	.379	.617	.204	.366
P3	.683	.093	.421	**.753**	.129	**.329**	.709	**.130**	.376	.664	.072	.424	.708	.117	.349	.620	.073	.490	.732	.107	.360
P4	.698	.157	.278	**.848**	**.357**	**.124**	.643	.082	.307	.796	.278	.144	.793	.313	.250	.515	.056	.547	.549	.059	.512
P5	.548	.123	.410	**.742**	**.286**	.332	.666	.212	.374	.644	.201	.409	.677	.208	**.323**	.459	.084	.590	.650	.195	.352
P6	.585	.046	.489	.664	.100	**.408**	**.691**	**.107**	.424	.575	.034	.493	.591	.063	.494	.647	.040	.447	.618	.034	.485
P7	.671	.040	.375	**.764**	**.125**	**.324**	.625	.025	.462	.650	.068	.465	.714	.066	.326	.623	.025	.458	.627	.023	.427
P8	.583	.093	.324	**.675**	.153	.258	.639	**.239**	.471	.586	.159	.462	.657	.117	**.230**	.574	.095	.386	.569	.125	.415
P9	.572	.058	.490	**.741**	**.201**	**.272**	.695	.165	.369	.686	.158	.386	.687	.132	.356	.570	.079	.521	.708	.164	.360
P10	.799	.298	.287	**.881**	**.458**	**.187**	.820	.367	.278	.778	.341	.452	.803	.353	.256	.678	.284	.546	.769	.328	.460
Avg	.650	.131	.380	**.759**	**.237**	**.285**	.687	.171	.376	.667	.159	.407	.713	.185	.320	.598	.108	.488	.651	.136	.418
Std	.073	**.086**	.071	.063	.121	.076	**.061**	.090	.059	.071	.092	.093	**.061**	.104	.071	.066	**.086**	.066	.066	.087	**.054**

* The higher the NDCG and MRR of the LtR model, the better the model is; the lower the KT of the LtR model, the better the model is.

VII. THREATS TO VALIDITY

This section describes possible threats to the validity of this study and our efforts to mitigate their impact.

Internal Validity. One of the possible threats to internal validity may result from the settings of CRR types. Since the effort (measured with time spent) put into different CRRs varies in practice, they may have an impact on the outcomes of prioritizing CRRs. After analyzing hundreds of CRRs and interviewing a dozen code reviewers at a global ICT enterprise, we set six types of CRRs based on keywords that appear in descriptions or function names of CRRs. Another threat concerns the setting of time spent reviewing different types of CRRs. Such information varies among code reviewers, and they were not recorded. Thus, the times spent were set as constants by interviewing code reviewers, rather than randomly generated by setting specific probability distributions in the simulation model. Finally, though our pilot experiments show that the selected features were slightly different in importance for each project, we did not customize the project-aware feature set; otherwise, it would have been difficult to distinguish the effects generated by the features or LtR models.

Construct Validity. There is no consensus on the measure of the workload of a review team. In this study, we used the historical review times for CRRs as a measure of the workload because it is a realistic record and has been quantitatively confirmed to be valid in most cases (cf. Section IV-B4). Another threat to the construct validity may result from the setting of the priority of CRRs. In this study, we followed the premise that "the most promising and urgent CRRs should be reviewed first" and used "acceptance" and "wait" to measure CRRs' outcome and urgency, respectively. These settings are reasonable as merged CRRs indicate that they are important to the project; otherwise, they may be rejected. While the longer a CRR waits for review, the lower its urgency is likely to be. On the other hand, although there are several metrics for evaluating LtR models, they are generally *pre-review* metrics. Therefore, we customized three *post-review* metrics to measure the impact of LtR models on code review. The new evaluation metrics have been confirmed valid by many software researchers and practitioners.

External Validity. We have only followed the code review processes at GitHub projects to construct the simulation model. The processes may slightly differ from those in the industrial context. However, we have only considered the main processes of code review. Therefore, the simulation model can be easily adapted to other contexts with minor customization. Moreover, some parameters and variables in the model cannot be accessed by fitting historical data (*e.g.*, time spent on reviewing, the workload of a review team) that were set by following the experience at one global ICT enterprise. When employing the simulation model, especially for the setting of the workload of a review team, should be tailed according to the specific contexts of code reviews.

VIII. CONCLUSIONS

Prioritizing CRRs in large-scale, active software projects nowadays has become one of the major challenges in code review. Even though a number of LtR models have been developed to support the prioritization of CRRs, their impact on code reviews is unclear because existing evaluations of LtR models are still conducted in a *pre-review* style. This study uses DES-based SPSM to simulate code review processes and proposes three evaluation metrics for *post-review* evaluation, *i.e.*, allowing CRRs to correlate with activities, artifacts, and reviewers in code reviews first, then analyzing LtR models' impact on the completion of reviewing CRRs and the delivery of code changes at the stage of code review. The experimental results indicate that using LtR has a positive impact on accelerating the completion of reviewing CRRs and the delivery of qualified code changes. Especially, LambdaMART is beneficial for accelerating completion; while AdaRank is beneficial for accelerating delivery. This study provides new ideas and techniques for software organizations that seek to evaluate LtR models and other AI-powered software techniques.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.62072227, No.62202219), the National Key Research and Development Program of China (No.2019YFE0105500) jointly with the Research Council of Norway (No.309494), as well as the Key Research and Development Program of Jiangsu Province (No.BE2021002-2). Alberto Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project 200021_197227.

REFERENCES

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 712–721.

[3] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 345–355.

[4] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 21th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2013, pp. 202–212.

[5] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 181–190.

[6] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2017.

[7] L. Dong, H. Zhang, L. Yang, Z. Weng, X. Yang, X. Zhou, and Z. Pan, "Survey on pains and best practices of code review," in *Proceedings of the 28th Asia-Pacific Software Engineering Conference*. IEEE, 2021, pp. 482–491.

[8] E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, pp. 106 737:1–24, 2022.

[9] L. Yang, H. Zhang, F. Zhang, X. Zhang, and G. Rong, "An industrial experience report on retro-inspection," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2022, pp. 43–52.

[10] Q. Chen, D. Kong, L. Bao, C. Sun, X. Xia, and S. Li, "Code reviewer recommendation in tencent: Practice, challenge, and direction," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2022, pp. 115–124.

[11] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 285–296.

[12] I. Steinmacher, I. Wiese, A. P. Chaves, and M. A. Gerosa, "Why do newcomers abandon open source software projects?" in *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 2013, pp. 25–32.

[13] S. Balali, I. Steinmacher, U. Annamalai, A. Sarma, and M. A. Gerosa, "Newcomers' barriers... is that all? an analysis of mentors' and newcomers' barriers in oss projects," *Computer Supported Cooperative Work*, vol. 27, no. 3, pp. 679–714, 2018.

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

[15] N. Saini and R. Britto, "Using machine intelligence to prioritise code review requests," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2021, pp. 11–20.

[16] E. Van Der Veen, G. Gousios, and A. Zaidman, "Automatically prioritizing pull requests," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 357–361.

[17] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3346–3393, 2018.

[18] M. I. Azeem, S. Panichella, A. Di Sorbo, A. Serebrenik, and Q. Wang, "Action-based recommendation in pull-request development," in *Proceedings of the 2020 International Conference on Software and System Processes*. ACM, 2020, pp. 115–124.

[19] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine Learning*, vol. 29, no. 2, pp. 131–163, 1997.

[20] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[21] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," *Journal of Machine Learning Research*, vol. 4, pp. 933–969, 2003.

[22] Z. Cao, T. Qin, T. Liu, M.-F. Tsai, and H. Li, "Learning to rank: From pairwise approach to listwise approach," in *Proceedings of the 24th International Conference on Machine Learning*. ACM, 2007, pp. 129–136.

[23] H. Li, "A short introduction to learning to rank," *IEICE Transactions on Information and Systems*, vol. 94, no. 10, pp. 1854–1862, 2011.

[24] T. Abdel-Hamid and S. E. Madnick, *Software project dynamics: An integrated approach*. Prentice-Hall, Inc., 1991.

[25] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[26] F. Zampetti, G. Bavota, G. Canfora, and M. Di Penta, "A study on the interplay between pull request review and continuous integration builds," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019, pp. 38–48.

[27] N. Fatima, S. Nazir, and S. Chuprat, "Software engineering wastes-a perspective of modern code review," in *Proceedings of the 3rd International Conference on Software Engineering and Information Management*. ACM, 2020, pp. 93–99.

[28] G. Rong, Y. Zhang, L. Yang, F. Zhang, H. Kuang, and H. Zhang, "Modeling review history for reviewer recommendation: A hypergraph approach," in *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022, pp. 1381–1392.

[29] P. Pandya and S. Tiwari, "Corms: A github and gerrit based hybrid code reviewer recommendation approach for modern code review," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022, pp. 546–557.

[30] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022, pp. 1035–1047.

[31] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: Automated code transformation to support modern code review process," pp. 237–248, 2022.

[32] G. Jeong, S. Kim, T. Zimmermann, and K. Yi, "Improving code review by predicting reviewers and acceptance of patches," *Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006)*, pp. 1–18, 2009.

[33] K. Islam, T. Ahmed, R. Shahriyar, A. Iqbal, and G. Uddin, "Early prediction for merged vs abandoned code changes in modern code reviews," *Information and Software Technology*, vol. 142, pp. 106 756:1–16, 2022.

[34] M. C. Paulk, C. V. Weber, S. M. Garcia, M. B. Chrissis, and M. Bush, "Key practices of the capability maturity model, version 1.1," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1993.

[35] M. I. Kellner, R. J. Madachy, and D. M. Raffo, "Software process simulation modeling: Why? what? how?" *Journal of Systems and Software*, vol. 46, no. 2-3, pp. 91–105, 1999.

[36] C. Gao, H. Zhang, and S. Jiang, "Constructing hybrid software process simulation models," in *Proceedings of the 2015 International Conference on Software and System Process*. ACM, 2015, pp. 157–166.

[37] T. Baum, F. Kortum, K. Schneider, A. Brack, and J. Schauder, "Comparing pre commit reviews and post commit reviews using process simulation," in *Proceedings of the International Conference on Software and Systems Process*, vol. 29, no. 11. Wiley, 2017.

[38] J. A. García-García, J. G. Enríquez, M. Ruiz, C. Arévalo, and A. Jiménez-Ramírez, "Software process simulation modeling: Systematic literature review," *Computer Standards and Interfaces*, vol. 70, pp. 103 425:1–18, 2020.

[39] Y. Li, H. Zhang, L. Dong, B. Liu, and J. Ma, "Constructing a hybrid software process simulation model in practice: An exemplar from industry," in *Proceedings of the 2020 International Conference on Software and System Processes*. ACM, 2020, pp. 135–144.

[40] Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. Dillon, B. Lakshminarayanan, and J. Snoek, "Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift," in *Proceedings of the 33rd Conference on Neural Information Processing Systems*. Curran Associates, Inc., 2019, pp. 13 991–14 002.

[41] "Anylogic simulation software," https://www.anylogic.com/.

[42] "Process modeling library," https://anylogic.help/library-reference-guides/process-modeling-library/pml-blocks.html.

[43] A. Mohamed, L. Zhang, J. Jiang, and A. Ktob, "Predicting which pull requests will get reopened in github," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. IEEE, 2018, pp. 375–385.

[44] H. Gong, H. Zhang, D. Yu, and B. Liu, "A systematic map on verifying and validating software process simulation models," in *Proceedings of the 2017 International Conference on Software and System Process*. ACM, 2017, pp. 50–59.

[45] O. I. Lindland, G. Sindre, and A. Solvberg, "Understanding quality in conceptual modeling," *IEEE software*, vol. 11, no. 2, pp. 42–49, 1994.

[46] R. Ahmed, T. Hall, and P. Wernick, "A proposed framework for evaluating software process simulation models," in *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling*, 2003.

[47] Y. Mahmood, N. Kama, and A. Azmi, "A systematic review of studies on use case points and expert-based estimation of software development effort," *Journal of Software: Evolution and Process*, vol. 32, no. 7, pp. e2245:1–20, 2020.

[48] "Data and materials," https://figshare.com/s/a033e99cd2a61e64c8bc.

[49] A. Alami, M. L. Cohn, and A. Wąisowski, "How do foss communities decide to accept pull requests?" in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2020, pp. 220–229.

[50] R. N. Iyer, S. A. Yun, M. Nagappan, and J. Hoey, "Effects of personality traits on pull request acceptance," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2632–2643, 2021.

[51] D. Wang, Q. Wang, J. Wang, and L. Shi, "Accept or not? an empirical study on analyzing the factors that affect the outcomes of modern code review?" in *Proceedings of the 21st International Conference on Software Quality, Reliability and Security*. IEEE, 2021, pp. 946–955.

[52] C. Maddila, C. Bansal, and N. Nagappan, "Predicting pull request completion time: A case study on large scale cloud services," in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 874–882.

[53] D. Moreira Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, "What factors influence the lifetime of pull requests?" *Software: Practice and Experience*, vol. 51, no. 6, pp. 1173–1193, 2021.

[54] S. Wang, C. Bansal, and N. Nagappan, "Large-scale intent analysis for identifying large-review-effort code changes," *Information and Software Technology*, vol. 130, pp. 106 408:1–15, 2021.

[55] G. Kudrjavets, A. Kumar, N. Nagappan, and A. Rastogi, "Mining code review data to understand waiting times between acceptance and merging: An empirical analysis," in *Proceedings of the 19th International Conference on Mining Software Repositories*. IEEE, 2022, pp. 579–590.

[56] X. Zhang, Y. Yu, T. Wang, A. Rastogi, and H. Wang, "Pull request latency explained: An empirical overview," *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–38, 2022.

[57] J. Xu and H. Li, "Adarank: A boosting algorithm for information retrieval," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2007, pp. 391–398.

[58] D. Metzler and W. Bruce Croft, "Linear feature-based models for information retrieval," *Information Retrieval*, vol. 10, no. 3, pp. 257–274, 2007.

[59] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao, "Adapting boosting for information retrieval measures," *Information Retrieval*, vol. 13, no. 3, pp. 254–270, 2010.

[60] C. Burges, R. Ragno, and Q. Le, "Learning to rank with nonsmooth cost functions," *Advances in Neural Information Processing Systems*, vol. 19, pp. 395–402, 2006.

[61] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[62] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? an empirical study," *Journal of Systems and Software*, vol. 171, pp. 110 806:1–14, 2021.

[63] "Rest api," https://docs.github.com/en/rest.

[64] "Ranklib," https://sourceforge.net/p/lemur/wiki/RankLib/.

[65] M. I. Azeem, Q. Peng, and Q. Wang, "Pull request prioritization algorithm based on acceptance and response probability," in *Proceedings of the 20th International Conference on Software Quality, Reliability and Security*. IEEE, 2020, pp. 231–242.

[66] I. X. Gauthier, M. Lamothe, G. Mussbacher, and S. McIntosh, "Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community," in *Proceedings of the 36th International Conference on Automated Software Engineering*. IEEE, 2021, pp. 30–41.

[67] K. A. Tecimer, E. Tüzün, H. Dibeklioglu, and H. Erdogmus, "Detection and elimination of systematic labeling bias in code reviewer recommendation systems," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2021, pp. 181–190.

[68] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng, "Machine/deep learning for software engineering: A systematic literature review," *IEEE Transactions on Software Engineering*, 2022.

[69] D. Kaur, S. Uslu, K. J. Rittichier, and A. Durresi, "Trustworthy artificial intelligence: A review," *ACM Computing Surveys*, vol. 55, no. 2, pp. 1–38, 2023.

[70] M. Krishnan, "Against interpretability: A critical examination of the interpretability problem in machine learning," *Philosophy & Technology*, vol. 33, no. 3, pp. 487–502, 2020.

[71] P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis, "Explainable ai: A review of machine learning interpretability methods," *Entropy*, vol. 23, no. 1, pp. 18:1–45, 2020.