CORRECT: Code Reviewer Recommendation at GitHub for Vendasta Technologies

Mohammad Masudur Rahman   Chanchal K. Roy   Jesse Redl§   Jason A. Collins†
University of Saskatchewan, Vendasta Technologies§, Canada, Google Inc.†, USA
{masud.rahman, chanchal.roy}@usask.ca, jredl@vendasta.com§,
jasonco@google.com†

ABSTRACT

Peer code review locates common coding standard violations and simple logical errors in the early phases of software development, and thus, reduces overall cost. Unfortunately, at GitHub, identifying an appropriate code reviewer for a pull request is challenging given that reliable information for reviewer identification is often not readily available. In this paper, we propose a code reviewer recommendation tool–CORRECT–that considers not only the relevant cross-project work experience (e.g., external library experience) of a developer but also her experience in certain specialized technologies (e.g., Google App Engine) associated with a pull request for determining her expertise as a potential code reviewer. We design our tool using client-server architecture, and then package the solution as a Google Chrome plug-in. Once the developer initiates a new pull request at GitHub, our tool automatically analyzes the request, mines two relevant histories, and then returns a ranked list of appropriate code reviewers for the request within the browser's context. Demo: https://www.youtube.com/watch?v=rXU1wTD6QQ0

CCS Concepts

•Software and its engineering → Software notations and tools; *Code Review;* Recommendation; **•Collaboration in software development →** Programming teams;

Keywords

Code reviewer recommendation, cross-project experience, specialized technology experience, GitHub, pull request

1. INTRODUCTION

Peer code review is reported to be highly effective for locating coding standard violations or for performing simple logical verifications [1, 4, 12]. It also helps identify source code issues (e.g., vulnerabilities) in the early phases of development, and thus, reduces overall cost for the software project [1, 4]. GitHub promotes a distributed and collaborative software development through pull requests and code reviews respectively. In GitHub, a developer forks from an existing repository (i.e., project), works on certain module of her interest, and then submits the changed files to the repository using a pull request [6]. During pull request submission, the developer is expected to choose one or more code reviewers who would review the code carefully before accepting the changes as a contribution. Unfortunately, choosing an appropriate reviewer for a pull request is a significant challenge [2], and to date, GitHub does not provide any support for this. Reliable information on reviewers' expertise (e.g., technology skill) is often not readily available, and it needs to be carefully mined from the codebase. Thus, the task of identifying appropriate reviewers is even more challenging and time-consuming for novice developers since they are neither familiar well with the codebase nor are aware of the skills of the hundreds of fellow potential reviewers. Such challenge is prevalent not only in open source development but also in the industrial environment where a company (e.g., Vendasta Technologies) strives to maintain code quality in the commercial software development and encourages collaborations among the developers in the form of peer code reviews.

Fortunately, there have been several studies that recommend code reviewers by analyzing past code review history (e.g., line change history [2], review comments [16, 18]), project directory structure [13, 14], and developer collaboration network [18]. In short, the existing studies mostly rely on the work history of a developer (i.e., potential reviewer) within a particular project and her collaboration history with other developers for determining her expertise. However, no studies consider the cross-project experience or the experience in various specialized technologies of a developer, and thus, they fall short in handling certain challenges. First, in industry, software developers often reuse software components (e.g., libraries) that were previously developed by themselves for cost-effective and faster development. Thus, their contributions scatter throughout different projects in the code repositories of the organization. Such contributions are a great proxy to their experience. Unfortunately, the existing studies on code reviewer recommendation completely ignore such information in expertise determination, and their recommendations are merely based on the contribution details within a particular project. Second, underlying tools and technologies of software projects are rapidly changing, and modern projects often involve different specialized and cutting edge technologies such as `map-reduce`, `task queues`, `urlfetch`, `memcache` and `pipeline`.

ASE'16, September 03-07, 2016, Singapore, Singapore
© 2016 ACM. ISBN 978-1-4503-3845-5/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2970276.2970283

Hence, code reviewers for a pull request are expected to have expertise in such technologies. However, neither mining of the revision history of changed files nor mining of the developer collaboration history, as the existing studies do, might be sufficient enough to ensure that. Thus, a technique that can analyze both relevant cross-project experience and specialized technology experience of a developer for a pull request, is likely to overcome the above challenges.

In this paper, we present a novel code reviewer recommendation tool–CORRECT– for pull requests at GitHub. The tool determines eligibility of a developer as code reviewer for a pull request by analyzing her past work experience with (1) external software libraries and (2) specialized technologies used by the pull request. Reference to the external libraries (i.e., software units external to the working project) in the code generally suggests one's working experience with such libraries, and we call it *cross-project experience*. Our key idea is– *if a past pull request uses similar external software libraries or similar specialized technologies to the current pull request, then the past request is relevant to the current request, and thus, its reviewers are also potential candidates for the code review of the current request.*

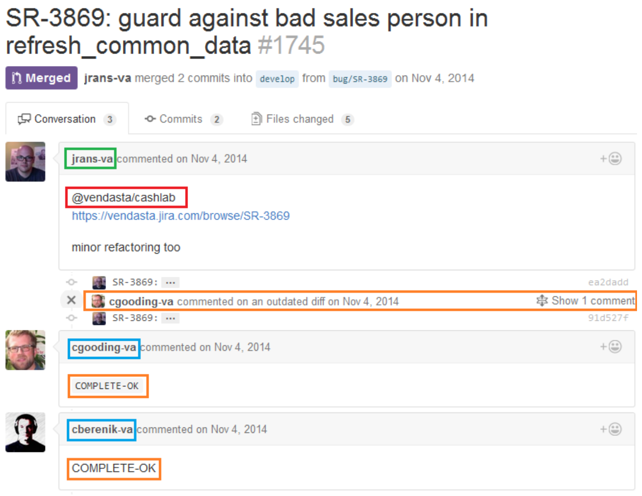
We first mine the external library and technology information from current pull request using static analysis, and then identify the relevant (i.e., similar) requests in terms of library and technology similarities from the recently submitted pull request collection. We then propagate the similarity score for each relevant request to its corresponding code reviewers as a proxy to the shared experience in external libraries and specialized technologies with the current request. Thus, each of the candidates accumulates scores for all relevant requests, and finally, the technique returns a ranked list of code reviewers. We adopt a client-server architecture for our recommendation system where the client module is packaged as a Google Chrome plug-in (i.e., as per the specification of Vendasta Technologies), and the server module is hosted as a web service. To summarize, our proposed tool provides the following features to support Vendasta software developers in the selection of appropriate code reviewers:

(a) automatically analyzes the technical details of a given pull request, and recommends a ranked list of appropriate reviewers for its code review,
(b) automatically captures and leverages two expertise dimensions of a developer–*cross-project experience* and *specialized technology experience*–for determining her expertise/eligibility as a code reviewer,
(c) offers customized recommendations for the developers using open authentication of GitHub,
(d) complements the existing pull request submission utility of GitHub through a Google Chrome plug-in, and
(e) provides a client-server architecture for seamless integration of our code reviewer recommendation service.

While this paper focuses on the tool aspect of our code reviewer recommendation approach, we refer the readers to the original paper [11] for further details.

2. MODERN CODE REVIEW (MCR)

Code review refers to a manual assessment of source code that identifies potential defects (e.g., logical errors) and quality problems (e.g., coding standard violations) in the code [3]. In recent years, code review has been assisted with various tools, which is less formal and more popular than the



SR-3869: guard against bad sales person in refresh_common_data #1745

Merged jrans-va merged 2 commits into develop from bug/SR-3869 on Nov 4, 2014

Conversation 3 · Commits 2 · Files changed 5

jrans-va commented on Nov 4, 2014

@vendasta/cashlab
https://vendasta.jira.com/browse/SR-3869

minor refactoring too

SR-3869: ... ea2dadd
cgooding-va commented on an outdated diff on Nov 4, 2014 · Show 1 comment
SR-3869: ... 91d527f

cgooding-va commented on Nov 4, 2014

COMPLETE-OK

cberenik-va commented on Nov 4, 2014

COMPLETE-OK

Figure 1: Code review interface at GitHub

traditional review techniques [2]. Such code review is called *Modern Code Review* (MCR) [3]. It is widely adopted both by the commercial organizations (e.g., Microsoft) and by the open-source communities (e.g., Android, LibreOffice). As an example, developer *jrans-va* (i.e., green box, Fig. 1) requests a development team– *cashlab* (i.e., red box, Fig. 1)– for code review during the submission of the pull request #1745. The request contains two commits associated with five changed source files. Two developers– *cgooding-va* and *cberenik-va* (i.e., blue boxes)– from the team analyze the commits from the pull request, perform the code review, and then post their feedback using comments (i.e., orange boxes, Fig. 1). Unfortunately, despite assistance from the static analysis tools [2], effective code review still remains a challenge, and *identifying appropriate reviewers* for the code review is even more challenging. To date, both reviewer selection and code review are performed manually at GitHub. Our tool recommends appropriate developers (e.g., *cgooding-va*) for such code review task (e.g., Fig. 1) at GitHub.

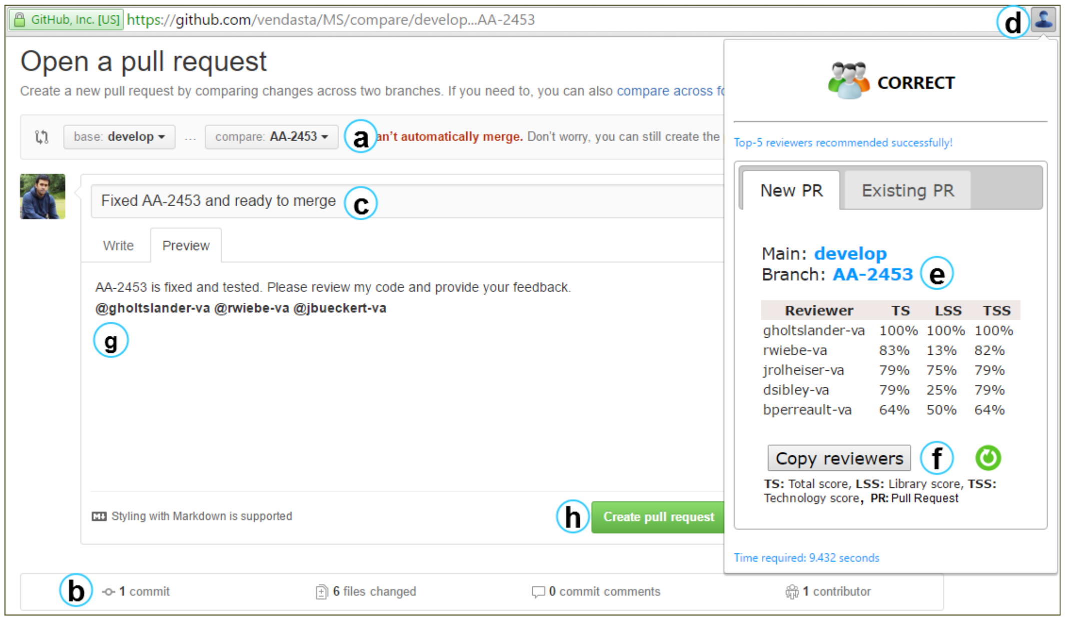
3. CORRECT: PROPOSED TOOL

Fig. 2 shows the user interface of CORRECT where we contribute in (d) browser's tool bar, (e)-(f) recommendation panel, and (g) pull request body panel. This section discusses different technical features provided by our tool.

(1) Use Cases: CORRECT provides automatic recommendation supports for two use cases of pull request based collaborative software development as follows:

(i) Submission of a New Pull Request: During the submission of a *new pull request*, a user (i.e., developer) compares her project branch (e.g., `AA-2453, Fig. 2-(a)`) with the base repository (e.g., `develop`, Fig. 2-(a)), and then looks for potential code reviewers. Our tool analyzes the changed source code files (e.g., Fig. 2-(b)) using static analysis, and then suggests a ranked list of appropriate code reviewers in the recommendation panel (e.g., Fig. 2-(e)).

(ii) Update of an Existing Pull Request: CORRECT can also recommend code reviewers for an *already submitted pull request* for which either no reviewers were assigned or inappropriate reviewers were assigned. This could be really helpful since inappropriate assignment of code reviewers often costs precious development time [14]. Our tool analyzes the changed source files from the pull request using static



GitHub, Inc. [US] https://github.com/vendasta/MS/compare/develop...AA-2453

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across f

an't automatically merge. Don't worry, you can still create the

Fixed AA-2453 and ready to merge

AA-2453 is fixed and tested. Please review my code and provide your feedback.
@gholtslander-va @rwiebe-va @jbueckert-va

Styling with Markdown is supported

Top-5 reviewers recommended successfully!

Main: develop
Branch: AA-2453

Reviewer	TS	LSS	TSS
gholtslander-va	100%	100%	100%
rwiebe-va	83%	13%	82%
jrolheiser-va	79%	75%	79%
dsibley-va	79%	25%	79%
bperreault-va	64%	50%	64%

TS: Total score, LSS: Library score, TSS: Technology score, PR: Pull Request

Time required: 9.432 seconds

Figure 2: User interface of CORRECT Tool

analysis, and then suggests the code reviewers.

(2) Automatic Mining of Relevant Artifacts: Our tool automatically mines both version control history and code review history of a software project for identifying appropriate code reviewers for a given pull request as follows:

(i) Analysis of Changed Source Code Files: Once either a branch (i.e., first use case) or a pull request (i.e., second use case) is selected, the tool collects all the changed files from each of their commits from the version control history. It accesses GitHub API end points for collecting the changed files, and uses `github-api`[1], a popular GitHub client library for the API access. Since each request or branch might involve a number of source code files, the tool only collects the path of the changed files from the API, and then applies that information to a local mirror of the GitHub repository for performing further static analysis.

(ii) Analysis of Code Review History: Our tool learns to recommend from past code reviews as was also learnt by existing literature [2, 14, 16]. It thus collects code review details of the 30 most recently submitted pull requests using GitHub API [11]. Since online API access could be time-consuming and could hurt the tool's performance, we adopt parallelization in the API access. In particular, we apply Java multi-threading to API access and further analysis for each of the past pull requests from the history.

(3) Automatic Recommendation of Code Reviewers: CORRECT returns a ranked list of five code reviewers for any given pull request (i.e., Fig. 2-(e)). The size of this recommendation is customizable, and the recommendation generally takes 10-15 seconds on average. The tool also provides additional insights to assist the user (i.e., developer) in the selection of appropriate code reviewers for her pull request. In particular, it provides relative expertise estimates (i.e., estimated by our original technique [11]) of the recommended reviewers on the *external software libraries* and *specialized technologies* used by the pull request. Our tool also provides several usability features as follows:

(i) Automated Use of Recommendation: Once code reviewers are recommended in the recommendation panel, the user can copy and paste the reviewers in pull request body panel (i.e., Fig. 2-(g)) by simply clicking *Copy reviewers* button (i.e., Fig. 2-(f)). Then she can submit the request by using *Create Pull Request* button (i.e., Fig. 2-(h)). The tool also provides a *Refresh* button (i.e., Fig. 2-(f)) to help the user start over the reviewer recommendation cycle.

(ii) Caching of Recommendation: Since we use a stateless protocol–HTTP, caching is a convenient way to improve the performance (i.e., response time) of the tool. Our tool uses `localStorage`, a storage feature of Google Chrome and other HTML5-capable browsers, to store the most recently collected recommendation result. In the case of repeated requests from the same page (i.e., branch or pull request), CORRECT displays previously stored result from `localStorage` database. The cache can also be cleared using the *Refresh* button (i.e., Fig. 2-(f)) if the user desires.

(4) Personalization & Optimization: CORRECT uses open authentication for GitHub API access, and thus, it has the potential not only for personalized reviewer recommendation but also for performance optimization as follows:

(i) Personalized Recommendation: Our tool captures a user's identity from the open authentication step, and then customizes the code reviewer recommendation for her. In particular, CORRECT discards self-reference (i.e., tool user herself as reviewer) from the recommendation list at present. However, other social aspects (i.e., developer collaboration history) could also be leveraged for further personalization of the reviewer recommendation.

(ii) Performance Optimization: GitHub restricts API access of an average registered user to a rate limit of 5,000 calls per hour. This restriction is likely to introduce *Denial of Service* issue with a tool (i.e., accessing GitHub API) if it is confined to one user account only, especially in an industrial context that involves frequent API access. Our tool overcomes that challenge using open authentication where the tool accesses the GitHub API on behalf of the logged in tool user, and thus, the access rarely exceeds the rate limit.

[1]https://github.com/kohsuke/github-api

4. WORKING METHODOLOGY

Fig. 3 shows the schematic diagram of our proposed tool–CORRECT. Our tool analyzes both version control history and code review history of a software project, and then suggests a ranked list of potential code reviewers for any given pull request. This section discusses the internal structures and working methodologies of the tool in brief while we refer the readers to the original paper [11] for further details.

Working Modules: CORRECT adopts client-server architecture, and it has two working modules–(1) *recommendation engine* and (2) *client module*. We package the *client module* as a Google Chrome plug-in and the *recommendation engine* as a web service. Once the plug-in is installed successfully, it appears as a *user icon* at the web browser's tool bar (e.g., Fig. 2-(d)). While the plug-in captures the technical details of a pull request from the web browser, the web service analyzes both the request and other relevant artifacts from the histories, and derives code reviewer recommendation for the request. Both modules communicate using REST and AJAX on top of HTTP.

Historical Data Collection: CORRECT collects 30 past *CLOSED* pull requests and their corresponding review details from a project for recommending code reviewers for a new pull request. We first identify each of those pull requests and extract their corresponding commits. Each of these commits can be identified using their SHA-1 based ID, and they generally contain one or more source files that were changed together. We collect such changed files from each of the selected past pull requests using GitHub API access and local repository analysis. We repeat the same steps for the new pull request, and collect the changed source files to be submitted to the base repository.

We then analyze the code review details of each of the past pull requests, and collect their corresponding reviewers using GitHub API access. In particular, we collect both the reviewers who were referred to during the submission (e.g., *rwiebe-va*, Fig. 2-(g)) and the reviewers who actually reviewed the pull request (e.g., *cgooding-va*, Fig. 1). Such historical information provides the foundation (i.e., ground truth) for the learning and evaluation of our tool.

Code Review Skill & Reviewer Ranking: Our key idea is– the developers who have reviewing experience on similar (i.e., relevant) past pull requests are suitable candidates for reviewing the current pull request to be submitted [2, 14]. Once changed source code files and review details from the past pull requests are collected, we determine their relevance to the current request based on their shared external libraries (e.g., `vapi, vform`) and adopted specialized technologies (e.g., `taskqueue, ndb`) in the changed files. In particular, we extract the external library or specialized technology names from each pull request, and determine *cosine similarity* between the current request and each of the past requests. We then propagate the similarity estimates (as a proxy to review expertise) to the corresponding code reviewers of the past requests. Thus, according to CORRECT, the software developers who have more experience on the attached external libraries (i.e., cross-project experience) and the adopted specialized technologies in the changed files of the current pull request, are more appropriate for the code review than the ones having less experience.

Example: Let us consider R_3 (Fig. 3) is a pull request to be submitted, and the submitter is looking for one or more code reviewers for the request. R_1 and R_2 are two past requests similar to R_3 containing one or more changed files. From Fig. 3, we see that each of R_1 and R_2 includes three libraries, adopts three specialized technologies, and is reviewed by a different set of developers. Similarly, R_3 also includes three external libraries and adopts three specialized technologies in the changed files. In order to recommend reviewers for R_3, CORRECT first determines the *cosine similarity* between libraries and technologies of R_3 and those of R_1 and R_2. It then applies those scores to the corresponding reviewers of R_1 and R_2. Thus, the developers who have the most review experience with similar past requests, bubble up in the ranked list for code reviewers. From Fig. 3, we see that reviewer A scores the top (i.e., 1.17) within all the reviewers according to our ranking algorithm, and thus, reviewer A is recommended as the code reviewer for the current request–R_3. We recommend the top five code reviewers [2, 14] from such a ranked list for any given pull request.

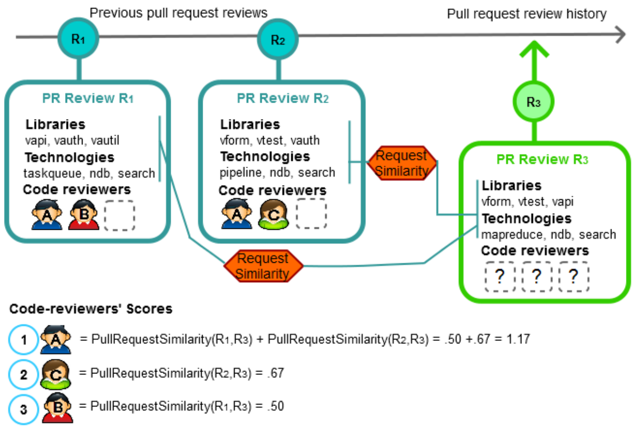


Figure 3: Working methodology of CORRECT, taken from the original paper [11]

5. A USE CASE SCENARIO

By means of a use case scenario, we attempt to explain how our tool–CORRECT–can help a software developer in choosing appropriate code reviewers for her pull request from within the context of a web browser.

Suppose, a developer, *Alice*, has started to collaborate on a new software project –`SR`– of Vendasta Technologies. She first forks from the base project which provides her a local copy of the project with complete access for code editing and committing. She then starts to fix a reported bug with ID– `SR-3869`– where she deletes 28 lines of code and adds 26 lines of code to the local project. When she is done with the fixation, it is found that the changes were made to five source code files bundled into two commits (i.e., Fig. 4). Then she attempts to submit the changes to the base repository using a pull request. Modern software companies like Vendasta often have a mandatory requirement for code review in order to maintain the code quality. Hence, she is also concerned about submitting the changes of higher code quality. During the pull request submission, she thus attempts to choose a list of expert developers who would review the changes before accepting them as a contribution to the base project. To date, GitHub does not provide any support for this task, and thus, she faces several challenges at this stage–(1) Who is the most appropriate code reviewer for these changes? (2) How to determine the code review skill of a developer? and

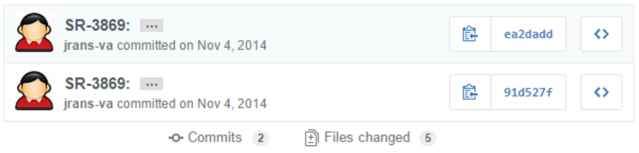

SR-3869: ...
jrans-va committed on Nov 4, 2014
ea2dadd
SR-3869: ...
jrans-va committed on Nov 4, 2014
91d527f
Commits 2
Files changed 5

Figure 4: Example use case: Commits & changed source code files (to be submitted)

Table 1: Libraries & Technologies of Use Case

External Library	Specialized Technology
`vapi, vtax, vbcsdk, vautil, vbcsdk.keys, vautil.validators.email`	`google.appengine.ext, ndb, search, google.appengine.api.search`

(3) Can we possibly identify appropriate reviewers from the past code reviews or version control history after all?

She might consider the original authors of the changed files as reviewer candidates. However, this might not be practical since the changed files might be authored by a number of developers over the years who might not be even with the company anymore. For this use case, we note that nine developers authored the changed files. *Alice* still needs to identify the most appropriate reviewers from those authors by herself with little or no helpful insights about them, which is a challenging task. The task is even more challenging for *Alice* if she is novice and/or non-familiar with the fellow developers or the code repositories of the company.

Now, let us assume that *Alice* has installed CORRECT plug-in on her Chrome web browser, and she initiates a pull request for submission. Our tool automatically collects the changed source files from the forked project using GitHub API access. Then it suggests her a ranked list of potential code reviewers by analyzing the most recently submitted similar pull requests (i.e., with code reviews) from the version control history of the base project. In particular, the tool automatically extracts *external library* and *specialized technology* information (e.g., Table 1) from each of the changed source files from each pull request, and then leverages the extracted information for code reviewer recommendation (Section 4). Besides ranking, the tool also provides additional insights on the library and technology related experience of the code reviewers (i.e., Table 2) which help *Alice* choose the right reviewers. For example, the top three reviewers–*cberenik-va, cgooding-va* and *ywang-va*–in the recommended list have the maximum expected experience, and she can confidently choose them as the reviewers for her changes. Thus, to overcome the challenges she faced previously, our tool (1) automatically suggests her a ranked list of appropriate code reviewers for the pull request (to be submitted), (2) automatically captures and leverages external library experience as well as specialized technology experience of the developers as suitable proxies to their code review skills, and (3) automatically mines both version control history and code review history using GitHub API access for deriving the recommendation. In short, our tool does all the heavy lifting for *Alice* in the background, and she can just get the recommendation by simply clicking a button during the pull request submission. More interestingly, CORRECT provides the recommendation within the context of the web browser which helps her maintain the usual work flow (i.e., within GitHub) and avoid the unexpected context-switching. In the context of Vendasta Technologies, we chose Google Chrome as the web browser. However, any browser plug-in capable of HTTP access can easily consume our recommendation service.

Table 2: Recommended Reviewers for Use Case

Reviewer	Total Score	Library Score	Technology Score
cberenik-va	100%	64%	100%
cgooding-va	99%	100%	99%
ywang-va	75%	52%	76%
sgryschuk-va	8%	17%	8%
ksookocheff-va	6%	0%	6%

6. EVALUATION & VALIDATION

One of the most effective ways for evaluating a code reviewer recommendation system is to consult with actual code reviews and the reviewers assigned for them from a codebase. We evaluate our technique using the real code review data from Vendasta codebase. In particular, we use 13,081 pull requests and their code review details from Vendasta as our oracle in evaluating CORRECT against a number of popular performance metrics. In order to further validate our findings and demonstrate its superiority, we experiment using 4,034 pull requests from six open source systems of three different programming languages, and compare with the state-of-the-art technique. While we discuss our evaluation and validation in brief as follows, the details can be found in the original paper [11].

Evaluation using Vendasta Systems: We evaluate our recommendation technique [11] using a collection of 13,081 pull requests from 10 subject systems (of Vendasta Technologies) and four state of the art performance metrics– Top-K Accuracy, Mean Reciprocal Rank, Mean Precision and Mean Recall. CORRECT provides a Top-5 accuracy of 92.15% and a Mean Reciprocal Rank of 0.67 with 85.93% precision and 81.39% recall which are highly promising according to relevant literature [2, 14, 16].

Comparison with the State-of-the-Art: We validate the performance of our technique by comparing with Thongtanunam et al. [14], the state-of-the-art technique for code reviewer recommendation which outperformed the earlier techniques. Our technique–CORRECT– provides 11.43% improvement in Top-5 accuracy and about 10% improvement in both precision and recall over the state-of-the-art. Three statistical tests– *MWU, Cohen's d* and *Glass* $\triangle$ – also suggest that such improvements are statistically significant.

Experiments with Open Source Projects: Although CORRECT was sufficiently evaluated using *Python* systems from Vendasta, we conduct another experiment with six open source projects from GitHub written in three different programming languages–*Java, Python* and *Ruby*– to generalize our findings. In this case, CORRECT recommends with a Top-5 accuracy of 85.20%, a Mean Reciprocal Rank of 0.69, a Mean Precision of 84.76% and a Mean Recall of 78.73%. Comparison demonstrates that our technique outperforms the state-of-the-art [14] with statistically significant margin. Further investigations also confirm that CORRECT does not show bias to any programming languages or any project types–*open source* and *closed source*.

Evaluation Plan with User Study: While CORRECT is found promising based on empirical evaluation, we plan to evaluate the tool using a user study involving professional developers from Vendasta. The goal of the study is to determine the usability and usefulness of the tool based on actual developers' feedback. In the user study, we plan to involve at least 10 developers working on 10 different running projects. Each participant will install the tool, use it for two controlled tasks (i.e., code reviewer assignment), and then will evaluate the recommendation provided by the tool with a predefined rating scale. We would then collect the numerical ratings as well as their qualitative feedback to triangulate them with our empirical findings.

7. RELATED WORK

Code Reviewer Recommendation: Existing studies recommend code reviewers by analyzing mostly code review or version control history [2, 13, 14, 16, 18] and developer collaboration networks [18]. Balachandran [2] proposes *ReviewBot* that analyzes *line change history* of the affected source lines from a given review request, and then identifies code reviewers from that history for the request. However, existing findings suggest that most of the lines are generally changed only once [14] which makes the line change history really scarce and thus, the performance of ReviewBot is limited. Thongtanunam et al. propose *RevFinder* [14] that identifies relevant review requests using *File Path Similarity (FPS)* [13], and then recommends reviewers from those requests for a review request at hand. RevFinder also outperformed earlier techniques including ReviewBot [14]. On the other hand, CORRECT identifies relevant pull requests using *external library similarity* and *specialized technology similarity* which are found to be more effective than File Path Similarity[14] for estimating relevance between pull requests, and thus for reviewer recommendation. In our earlier work [11], we show that our technique outperformed RevFinder with statistically significant performance improvements. Another recent work [16] applies machine learning on past code reviews, and combines textual similarity with File Path Similarity [14]. Thus, it suffers from similar issues as of RevFinder such as pull request relevance issue, and that the learned models could be biased to the subject systems under study.

The remaining technique–Yu et al. [18] analyzes past review comments and developer collaboration networks for reviewer recommendation. While we use library and technology similarity between pull requests for determining relevant past requests, they use review comment similarity (i.e., textual similarity) for the same purpose. Besides, their idea is still not properly evaluated or validated.

Expert Recommendation: Kintab et al. [9] identify expert developers on a code fragment of interest by exploiting *code similarity* with other segments. Similar technique is applied by da Trindade et al. [5] where they develop a communication network among documents, source code and developers, and then recommend dominant developers as experts. Yang [17] studies the developer network using *code review relationship*, and identifies core and peripheral developers using different network properties. There exist several studies in the domain of *bug triaging* that analyze duplicate bug reports [8] or apply IR-based traceability [10] techniques for recommending appropriate developers for bug fixation. Several studies are also conducted on expert user recommendation at Stack Overflow that analyze cross-domain contributions [15] or question difficulty [7] for expertise estimation. While these expert recommendation techniques are somewhat similar to ours, their context of recommendation is different and thus, comparing ours with them is not feasible. Of course, we introduced two novel and effective expertise paradigms (*cross-project experience* and *specialized technology experience*) which were not exploited by any of the recommendation systems. This makes our proposed tool–CORRECT– significantly different from all of them.

8. CONCLUSION

To summarize, we propose a novel tool– CORRECT– for code reviewer suggestion for pull requests at GitHub for Vendasta Technologies. It automatically captures the experience of a developer with the external libraries (i.e., cross-project experience) and specialized technologies used in a given pull request, applies such experiences as proxies to code review skill of the developer, and then suggests a ranked list of appropriate code reviewers. Our recommendation technique is substantially evaluated and validated using empirical data. We package our solution as a web service and a plug-in for Google Chrome browser. The tool can assist a developer in choosing appropriate code reviewers during the submission of a new pull request or during the update (i.e., reviewer assignment) of an existing pull request.

REFERENCES

[1] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proc. ICSE*, pages 712–721, 2013.

[2] V. Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *Proc. ICSE*, pages 931–940, 2013.

[3] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern Code Reviews in Open-Source Projects: Which Problems do they Fix? In *Proc. MSR*, pages 202–211, 2014.

[4] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In *Proc. FSE*, pages 257–268, 2014.

[5] C.C. da Trindade, Y.A.M. Barbosa, A.K.O. Moraes, J.O. de Albuquerque, and S.R.L. Meira. An Expert Recommender System to Distributed Software Development: Requirements, Project and Preliminary Results. In *Proc. SBSC*, pages 161–168, 2009.

[6] G. Gousios, M. Pinzger, and A. v. Deursen. An Exploratory Study of the Pull-based Software Development Model. In *Proc. ICSE*, pages 345–355, 2014.

[7] B. V. Hanrahan, G. Convertino, and L. Nelson. Modeling Problem Difficulty and Expertise in Stackoverflow. In *Proc. CSCW*, pages 91–94, 2012.

[8] K. Kevic, S.C. Muller, T. Fritz, and H.C. Gall. Collaborative Bug Triaging using Textual Similarities and Change Set Analysis. In *Proc. CHASE*, pages 17–24, 2013.

[9] G. Kintab, C. K. Roy, and G. Mccalla. Recommending Software Experts using Code Similarity and Social Heuristics. In *Proc. CASCON*, page to appear, 2014.

[10] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging Incoming Change Requests: Bug or Commit History, or Code Authorship? In *Proc. ICSM*, pages 451–460, 2012.

[11] M. M. Rahman, C. K. Roy, and J. Collins. CORRECT: Code Reviewer Recommendation Based on Cross-Project and Technology Experience. In *Proc. ICSE*, pages 222–231, 2016.

[12] P.C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D.M. German. Contemporary Peer Review in Action: Lessons from Open Source Development. *TSE*, 29(6):56–61, 2012.

[13] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida. Improving Code Review Effectiveness Through Reviewer Recommendations. In *Proc. CHASE*, pages 119–122, 2014.

[14] P. Thongtanunam, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who Should Review my Code ? In *Proc. SANER*, pages 141–150, 2015.

[15] R.l Venkataramani, A.l Gupta, A. Asadullah, B. Muddu, and V. Bhat. Discovery of Technical Expertise from Open Source Code Repositories. In *Proc. WWW*, pages 97–98, 2013.

[16] X. Xia, D. Lo, X. Wang, and Xiaohu Y. Who Should Review this Change? Putting Text and File Location Analyses Together for More Accurate Recommendations. In *Proc. ICSME*, 2015.

[17] Xin Yang. Social Network Analysis in Open Source Software Peer Review. In *Proc. FSE*, pages 820–822, 2014.

[18] Y. Yu, H. Wang, G. Yin, and C. Ling. Reviewer Recommender of Pull-Requests in GitHub. In *Proc. ICSME*, pages 609–612, 2014.