Preprint: Marcin Kawalerowicz and Lech Madeyski, "Jaskier: A Supporting Software Tool for Continuous Build Outcome Prediction Practice" in IEA/AIE 2021. Advances and Trends in Artificial Intelligence. From Theory to Practice (H. Fujita, A. Selamat, J. C.-W. Lin, and M. Ali, eds.), vol. 12799 of Lecture Notes in Computer Science, (Cham), pp. 426–438, Springer International Publishing, 2021. DOI: 10.1007/978-3-030-79463-7_36

Preprint: https://madeyski.e-informatyka.pl/download/KawalerowiczMadeyski21b.pdf

Jaskier: A Supporting Software Tool for Continuous Build Outcome Prediction Practice

Marcin Kawalerowicz[1][0000−0002−8411−0199] and Lech Madeyski[2][0000−0003−3907−3357]

[1] CODEFUSION Sp. z o.o. and Faculty of Electrical Engineering, Automatic Control and Informatics, Opole University of Technology, Poland, marcin@kawalerowicz.net
[2] Faculty of Computer Science and Management, Wroclaw University of Science and Technology, Poland. Lech.Madeyski@pwr.edu.pl

Abstract. Continuous Defect Prediction (CDP) is an assisting software development practice that combines Software Defect Prediction (SDP) with machine learning aided modelling and continuous developer feedback. Jaskier is a set of software tools developed under the supervision and with the participation of the authors of the article that implements a lightweight version of CDP called Continuous Build Outcome Prediction (CBOP). CBOP uses classification to label the possible build results based on historical data and metrics derived from the software repository. This paper contains a detailed description of the tool that was already started to be used in the production environment of a real software project where the CBOP practice is being evaluated.

Keywords: software defect prediction · continuous integration

1 Introduction

Jaskier[3] is a software infrastructure aimed to support the Continuous Defect Prediction (CDP) practice proposed earlier by the authors [1]. CDP combines the unambiguous results of continuous integration (CI) like proposed in [2] with machine learning (ML) aided just-in-time quality assurance [3] and continuous feedback directly in the developer Integrated Development Environment (IDE) [4] and [5]. It implements the lightweight version of CDP called Continuous Build Outcome Prediction (CBOP). The basic idea of this practice is to combine the defect prediction techniques with the notion of continuous, just-in-time developer feedback on possible defects in their code. To implement CDP/CBOP, we needed to develop a set of client and server tools. Clients run on the software developer machine where code is written. The clients need to watch what a developer is doing and gather metrics related to the changes he is making before the source code gets to the software repository. On the server side, we need a self-learning prediction model that was made accessible to the client tools to get the prediction and show it to the developer. The key objective of this paper it to present Jaskier, a practical implementation of CBOP. We will look closely at the problem Jaskier is trying to solve amid related work. We will provide the

[3] Jaskier is a code name we have used to develop the software at a company one of the authors is running. Jaskier in Polish means buttercup – a small yellow flower common in Poland.

rationale behind developing the tool and provide a detailed description of the software architecture together with implementation details. We will throw a short look at an experiment where we have used Jaskier to evaluate the CDP/CBOP practice.

2 Problem statement and related work

Continuous integration (CI) is "an automated process that builds [...] the software after each change in source code, to ensure its health [and ...] provides immediate feedback [...]" [6]. CI is very broadly used in the software engineering industry. There are a few rules that need to be obeyed while using CI, that might interrupt the flow of the process of software development. If the build fails, no one is allowed to pull the faulty code from the repository. Furthermore, pushes to the repository are banned until the failing build is fixed (preferably by the person that broke the build). This fix needs to be done as soon as possible, because it interrupts the whole development process. Other team members need to wait with necessary actions on the repository (like pushing or merging). Sometimes, if the build problems are more severe, fixing them may take substantial amount of time and have a greater impact on the project. This needs to be avoided.

There are techniques and rules that help deal with such situation. For example, builds need to be fast and if necessary, they shall fail fast. Maximum of 10 minutes is often taken as a rule of thumb for the maximal build duration. The tests in the CI process are often sorted from the ones that are the fastest, to the ones that take longer to execute. The rationale is that if the build needs to fail, it should fail sooner than later. Another technique is gated check-ins or pre-commit tests. In this technique, the CI process is started using the code prepared to be passed to the repository before it is actually checked-in. Regardless of countermeasures: the builds are sometimes failing, and it causes additional work and delays.

In order to mitigate this problem we are proposing to use CBOP, a lightweight implementation of CDP [1]. This technique is based on the years of research in the area of Software Defect Prediction (SDP) [5,7,3,8,9,10,11,12] that was implemented as software tools in [4,5]. Our approach is based rather on CI data [2] than lexical examination of the commit message or the bug tracking software entries [13].

Continuous Defect Prediction (CDP) is a practice we proposed that relays on machine learning and data form bug tracking software or build server. The goal of this practice is to detect possible software defects before they manifest themselves. Continuous Build Outcome Prediction (CBOP) is a subset of CDP that deals specifically with failing builds. Jaskier is the first attempt to put the idea into production. Build fail prediction model is calculated based on the build results in correlation with the software changes that triggered that build. The goal is to give the developer a tool that can run in the background of their IDE and, in real-time, gather the intelligence about the changes developer applies to the project and its possible outcomes. Based on these changes, the tool calculates the metrics and uses them to estimate the possibility that the changes can disrupt the build server or cause a bug. By doing so, we hope to give the software developer additional insight into the outcomes of his work. Maybe if the developer knows that the changes, he made are predicted to be risky, he can mitigate the problem before it occurs and disrupt the development flow.

3 Software framework and architecture

To give a comprehensive view of the tool and depict it as accurate as possible, we will use the “4+1” model view of architecture provided by Philippe Kruchten [14]. We use the “4+1” model because it was designed to describe software intensive/complex system like Jaskier. Furthermore is not bound to any notation or tool. It will allow us to show our system from the point of view of various stakeholder. Figure 1 shows the “4+1” view model divided into five views:

1. Logical View provides a logical design of the software.
2. Process View describes the integration aspects of the software.
3. Physical View provides an overview of how the software is mapped to hardware.
4. Development View describes the organization of the software project in a development environment.
5. Use Case View which describes usage scenarios of the software form the point of view of the end-users.

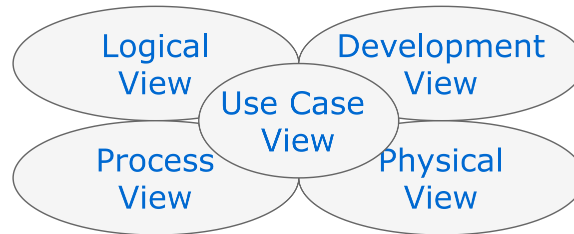

Fig. 1. ”4+1” View Model of Software Architecture

3.1 Logical View

Jaskier is a set of tools that address the problem described in Section 2. The software’s logical design can be described in terms of the static structure and dynamic behaviour. In terms of the structure, it consists of independent and autonomous components:

1. IDE plugins for Microsoft Visual Studio and Microsoft VSCode used to gather just-in-time data and present predictions to the developer based on that data,
2. Web Service prediction interface that abstracts the access to a prediction model,
3. Model that can live in the cloud or on-premises and is used to generate predictions over a Web Service,
4. Model training module - Azure ML Training Experiment for the cloud-based model or R script for the on-premises model,
5. Statistics Harvester - a simple .NET Framework tool for gathering the statistics to be used for training (it is using the same mechanisms as the IDE plugin to do so),
6. Jaskier Database – a place where the statistics together with the build results are stored.

3.2 Process View

Figure 2 shows the integration of the tools described in Section 3.1. IDE plugins reside inside Visual Studio or VSCode and are constantly monitoring the work of the developer. If the developer changes something in the project (e.g., saves a file), the plugins are comparing local repository with the changes developer made to the source code to calculate current statistics. These statistics then are sent to the Web Service. The web service relays the statistics to the Prediction Model. Based on the model, a prediction of build failure is sent back through the Web Service. The prediction then is presented to the developer inside IDE using the plugin.

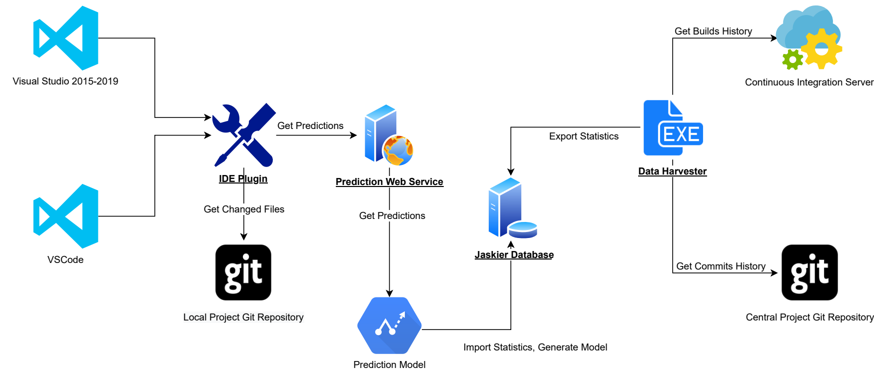

Fig. 2. Jaskier logical view

On the build server side, a new commit to a central repository triggers a build. After the build Data Harvester is started, it goes over the repository and gathers the statistics knowing whether those changes broke the build. The data is stored in Jaskier Database and then used to train the model once again.

3.3 Physical View

The IDE plugins run solely on the developer machine. It is a machine running on Microsoft Windows in case of Visual Studio 2015-2019 or macOS, Linux or Windows machine in case of VSCode. The Prediction Web Service needs a web server to run. For example, a standalone Microsoft IIS or Azure Cloud-based web server. The prediction model can live on a standalone Microsoft R Server or in Azure ML. Jaskier Database is a Microsoft SQL Server database. It also needs a standalone SQL Server engine to run or can be hosted in Azure Cloud. The Data Harvester resides on the Continuous Integration server.

Figure 3 shows the on-premises setup and figure 4 depicts the Azure Cloud setup of Jaskier.

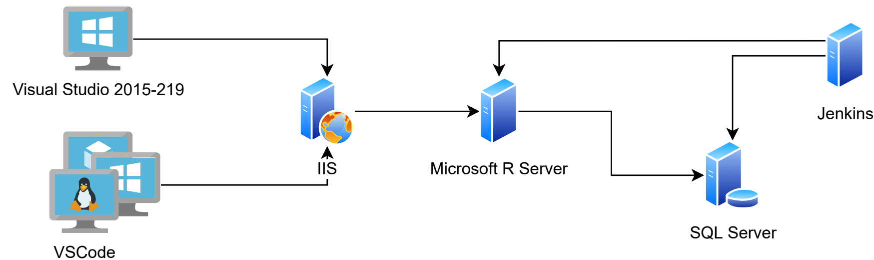

Fig. 3. Jaskier physical view

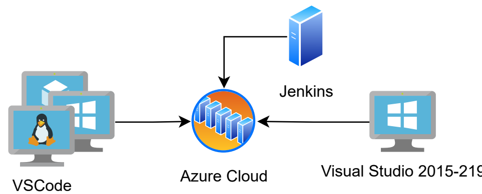

Fig. 4. Jaskier physical cloud view

3.4 Development View

From the development point of view, the Jaskier toolchain consists of the following modules:

1. Codefusion.Jaskier.API – interfaces and DTOs (Data Transfer Objects),
2. Codefusion.Jaskier.Common – common functionalities (database definition, requests and responses, helper classes, services),
3. Codefusion.Jaskier.Client.CLI – command line Jaskier client,
4. Codefusion.Jaskier.Client.VS2015 – Visual Studio 2015-2019 extension,
5. Codefusion.Jaskier.Client.VSCode – VSCode extension,
6. Codefusion.Jaskier.Export.CLI – Data Harvester code,
7. Codefusion.Jaskier.Web – Prediction Web Service,
8. R scripts – R scripts used on Microsoft R Server for training and prediction.

Figure 5 shows the project references and implementations in Jaskier Visual Studio solution.

3.5 Use Case View

The use case view gives a good perspective on how the system is being used by its users. Figure 6 shows the Use Case Diagram for Jaskier. We have two actors. A software developer is a primary actor. Software developers get predictions from the system. The second actor is someone (or sometimes an automated process) that prepares prediction models. Table 1 describes the use cases in more detail.

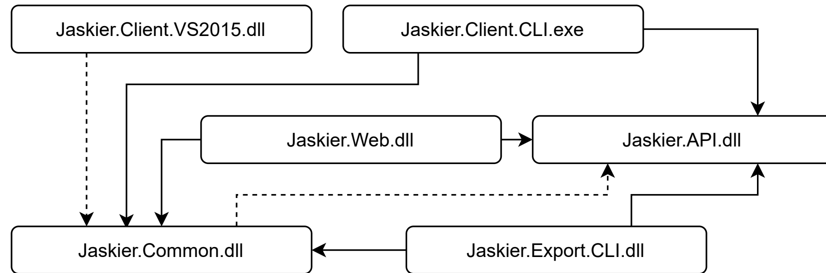

Fig. 5. Jaskier project references and implementations (stripped line).

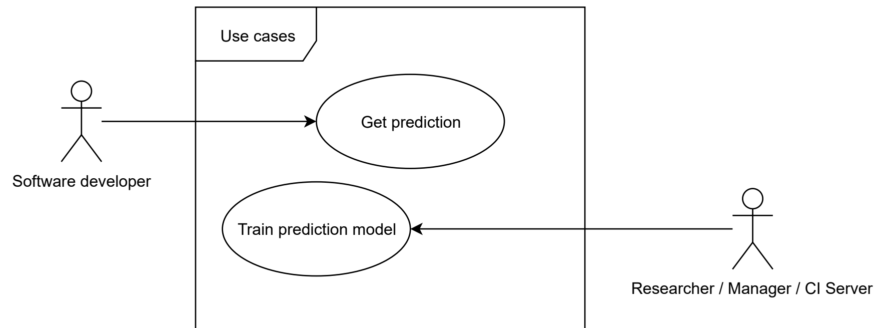

Fig. 6. Jaskier use case view

Table 1. Jaskier use cases

Use case	Actor	Basic flow
Get predictions	Software developer	While software developer writes his source code the just-in-time prediction of the build result is provided in his IDE.
Train prediction model	Manager / Researcher / CI Server	Prediction model is created (trained or re-trained) by software development manager, researcher or automatically by the CI server.

4 Implementation details

In this section, we will extend the short overview of the implementation provided in Section 3.4, have a closer look into the implementation details, and provide information about the individual parts of the software.

4.1 Clients

Jaskier has (at the moment of writing this text) three clients: Visual Studio and VSCode Extensions and a CLI tool.

Visual Studio Extension and the CLI tool are written in C# and use Microsoft .NET Framework. The VSCode Extension is written in TypeScript.

Visual Studio Extension is shown in Figure 7. The core functionality of the C# client is defined in the IChangeTrackerService interface placed in Codefusion.Jaskier.API project. GitChangesTrackerService is an implementation of this interface in Codefusion.Jaskier.Common project. As the name suggests, this implementation uses Git to determine the changes. It knows the repository path, and based on that repository, it can determine which files changed since the last commit. The library LibGit2Sharp[4] is used to operate on the Git repository.

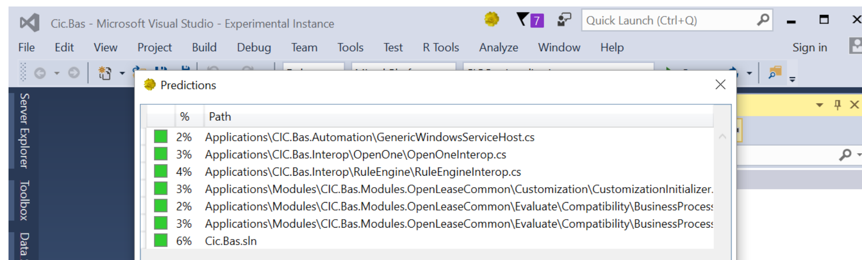

Fig. 7. Prediction extension results in Microsoft Visual Studio

If any other source code management system needs to be used, the IChangeTrackerService needs to be implemented for that system.

To determine if a prediction is needed and to retrieve the prediction for any given change, implementation of IPredictionService is used. The implementation of this service is used in C# client applications.

VSCode Extension is shown in Figure 8.

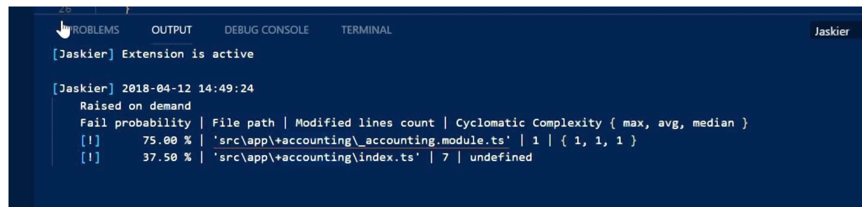

Fig. 8. Prediction extension results in Microsoft VSCode

Because of the nature of VSCode, which is an HTML5 application, our Jaskier extension is written entirely in TypeScript. The architecture of this extension roughly resembles the architecture of the Visuals Studio extension. It is also service-based and loosely coupled. We have the PredictionsService class that corresponds to the IPredictionService interface from the C# version and GitChangesTracker, which corresponds to the IChangeTrackerService interface.

[4] https://github.com/libgit2/libgit2sharp

4.2 Data Harvester and Database

Another important interface in Jaskier is IBuildInfoServie. This interface defines the methods used to harvest the build information from a CI system. At the moment of writing this article, two implementations of this interface exist–one for Bitbucket and one for TravisTorrent. Bitbucket implementation was used in the performed preliminary empirical evaluation of CBOP in real project, while TravisTorrent implementation was used to create a data mining package for [1].

To harvest the statistics (metrics) from a repository, IBuildStatisticsService needs to be implemented. In Jaskier, GitBuildStatisticsService is implemented. It uses Git repository to calculate the code metrics for any given branch. To determine the changes that need the statistics calculated, it uses the before mentioned GitChangesTrackerService.

The whole configuration of the dependencies in the Harvester project (Codefusion.Jaskier.Export.CLI) is done using the SimpleInjector[5] inversion of control library. By using the Dependency Injection (DI) pattern, it is possible to configure the needed libraries on runtime in app.config.

The Harvester runs as a standalone Windows executable. It needs a direct connection to the database to store the statistics and commit information.

The database used in Jaskier runs on Microsoft SQL Server engine. It is a simple relational database. Figure 9 shows the database diagram with the tables that Jaskier needs to run. There are also other tables in the database that were added to facilitate Agile Experimentation [15] within the tool.

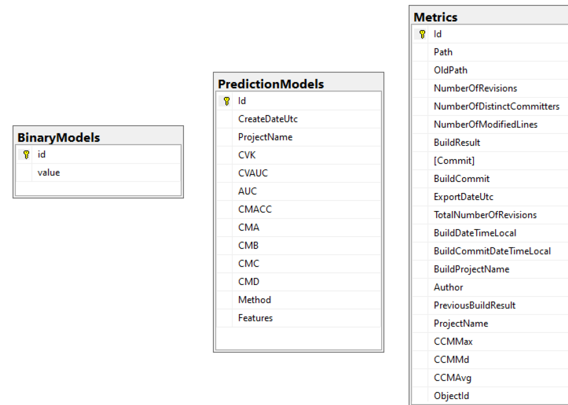

Fig. 9. Part of the Jaskier database

The central table in Jaskier is the Metrics table. This table stores the data gathered by the Data Harvester. For every changed file in any given commit, a set

[5] https://simpleinjector.org/

of metrics together with the build status is stored. Data from this table is used to train and retrain the prediction models.

Those prediction models can also be stored in the database in the BinaryModels table. This table is only used for an on-premises installation of Jaskier. In a cloud base setup, the models are stored on Azure Storage. Details on that will be provided in Section 4.3.

The table PredictionRequests stores all the build result inquiries from the Jaskier clients together with the resulting prediction.

4.3 Prediction Model

The Prediction Web Service serves as a communication mean between the Clients and the Prediction Model. The Prediction Model can be calculated, stored, and used in two different ways:

1. By using an on-premises Microsoft Machine Learning Server,
2. By using Microsoft Azure ML cloud infrastructure.

At the time of writing Jaskier, the Microsoft Machine Learning Server official name was Microsoft R Server. This server's history dates to the time when R was maintained and developed by Revolution Analytics, which was bought by Microsoft in 2015.

The setup of a standalone ML server is aimed at the enterprise level. At that level, all aspects of the software need to be controlled and under the strict supervision of the software company, which is using it. In such an environment, no cloud or multi-tenant setup is possible or wanted. In that setup, the Prediction Model is calculated using the R scripts and stored in the database.

Alternatively, to an on-premises R usage within Microsoft Machine Learning Server, Jaskier is prepared to work with Microsoft Azure ML as a provider of Prediction Models. In that case, the training, retraining, and prediction are done using Experiments in Microsoft Azure cloud. Experiments in Azure are configured using a diagramming tool. The basic steps of a Training Experiment are to read the data (Import Data) from the database and select the proper columns for the experiment (Select Columns in Dataset). Then to split the data into training and scoring set (Split Data). The training part of the data goes to model training (Train Model) which takes the input from a training algorithm (e.g., Two-Class Boosted Decision Tree). The scoring set plus the model's output goes to scoring (Score Model) and evaluation (Evaluate Model). The Training Experiment can be published as a web service to facilitate retraining functionality. In that case, input and output knots are created: Web Service Input for new data and two Web Service Outputs for the model and for the scoring output. The Training Experiment, when ready, can be transformed using Azure ML functionalities into Prediction Experiment. The Prediction Experiment has a similar structure but gets the output model from the Training Experiment as input for scoring. It can also be published as a Web Service to provide a prediction for external data sources.

4.4 Prediction Web Service

In order to abstract the cloud and on-premises installation of the prediction service, a Prediction Web Service was built. It was placed in the solution as Jaskier.Web project. It is an ASP.NET which hosts three endpoints:

1. Training endpoint,
2. Prediction endpoint,
3. Telemetry endpoint.

The training endpoint abstracts the Azure ML Training Experiment Web Service and the Microsoft Machine Learning Server training R script. If called, it will retrain the model using current data from the database. As an input, it gets the project name for which the prediction model should be retrained.

The prediction endpoint abstracts the Azure ML Prediction Experiment and the Microsoft Machine Learning Server prediction R script. If called, it will provide the prediction for a given (in the request) set of changes. As an input, it gets the list of PredictionRequestFile objects for a given project. As an output, it delivers a list of PredictionResponse objects that contain the calculated probabilities.

The prediction endpoint in the case of Microsoft Machine Learning Server saves the requested data into the PredictionRequests table in the database and generates unique Guid for every request. This Guid is then passed as a command-line parameter to the R script. In the case of Azure ML Prediction Experiment, the Rest API call is made to Azure ML. The call contains all the input data. It triggers a synchronous prediction execution in the cloud.

The telemetry endpoint gets the information in form of a PutTelemetryRequest object. It contains a set of telemetry information like the username, machine name, user IP address, plugin version, Visual Studio version, an action and action payload. Action contains the name of the action the developer took during the Jaskier usage, and the payload contains additional information to the action. At the time of writing the article, the telemetry service gathered information about the user opening or closing the Jaskier pane in Visual Studio. This feature was introduced to Jaskier to be able to tell if the user was potentially seeing the provided prediction.

5 Experiment using Jaskier

The first evaluation of Jaskier was conducted in real software development settings during an experiment aimed to analyze continuous build outcome prediction (CBOP) practice for the purpose of evaluation with respect to its effectiveness to reduce the build failure rate (BFR) from the point of view of the researcher and project manager in the context of a small group of professional software developers (subjects) working in an industrial-grade software project (object)."

In order to achieve this goal, we have modified Jaskier to facilitate "A small-n repeated-measure design with two conditions and replicates" experiment design. The experiment is described in detail in [16]. Here we will provide brief overview of the results. We gathered from 27 to 60 days of data from 9 developers in a professional

setup. Contrary to our expectation, we have registered a ”small effect size” indicating that getting the prediction results while using Jaskier was negatively affecting BFR. The drop was very slight but noticeable. During the phase with CDP/CBOP turned off we measured the mean of BFR to be 0.137. While the CDP/CBOP prediction was turned on the measured the mean BFR to be 0.185. The working hypothesis to explaining the situation is either: (1) the model quality was insufficient and got the developers false believe they are on the road to successful build while there were in fact rendering the build failure; (2) or the developers merely seeing the build prediction were more restless, and that caused their commits to fail the build more often.

6 Conclusion and future work

Jaskier was created to be the first practical implementation of the CDP/CBOP practice. A set of tools was developed under the supervision and with the participation of the authors of the article by a professional software development company. The set of tools consists of three developer clients (for Visual Studio, VSCode and command line), a web service to facilitate the data exchange between the clients and prediction model living in the cloud or on on-premises machine learning server. A set of scripts to train and retrain the model together with the scripts to provide predictions was developed. Jaskier was made available as an open source on GitHub[6].

In order to facilitate the evaluation of the CDP/CBOP practice, Jaskier was extended with the ability to support an experiment devised by the authors of this article.

Jaskier, with its cloud-based prediction system, was used to examine the BFR in a professional software development project. The result was contrary to the desired outcome. It showed that the use of our tool was impacting BFR in a slightly negative way. Nevertheless, Continuous Build Outcome Prediction (CBOP) is only a part of CDP practice. Failing build is one type of issue CDP aims to fight. It would be very interesting to see how the prediction of other types of issues (e.g., real post-release software defects) performs in Jaskier. Next steps would be to build prediction models based on the bug reports in a bug tracking system and use them in Jaskier. It would examine if this kind of prediction, if provided to the developer, would perform better in a software development project.

References

1. Madeyski, L., Kawalerowicz, M.: Continuous Defect Prediction: The Idea and a Related Dataset. In: 14th International Conference on Mining Software Repositories (May 20-21, 2017. Buenos Aires, Argentina)., pp. 515–518 (2017). https://doi.org/10.1109/MSR.2017.46
2. Finlay, J., Pears, R., Connor, A.M.: Data stream mining for predicting software build outcomes using source code metrics. Information and Software Technology **56**(2), 183–198 (2014). https://doi.org/10.1016/j.infsof.2013.09.001

[6] https://github.com/ImpressiveCode/ic-jaskier

3. Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engineering **17**(4), 375–407 (2010). https://doi.org/10.1007/s10515-010-0069-5
4. Madhavan, J.T., Whitehead Jr., E.J.: Predicting buggy changes inside an integrated development environment. In: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange, eclipse '07, pp. 36–40. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1328279.1328287
5. Kim, S., Whitehead Jr., E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. **34**(2), 181–196 (2008). https://doi.org/10.1109/TSE.2007.70773
6. Kawalerowicz, M., Berntson, C.: Continuous Integration in .NET. Manning Pubs Co Series. Manning (2011)
7. Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J.: On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering **14**(5), 540–578 (2009). https://doi.org/10.1007/s10664-008-9103-7
8. D'Ambros, M., Lanza, M., Robbes, R.: Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Engineering **17**(4-5), 531–577 (2012)
9. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2013). https://doi.org/10.1109/TSE.2012.70
10. Jureczko, M., Madeyski, L.: Cross–Project Defect Prediction With Respect To Code Ownership Model: An Empirical Study. e-Informatica Software Engineering Journal **9**(1), 21–35 (2015). https://doi.org/10.5277/e-Inf150102
11. Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 17–26 (2015)
12. Yang, X., Lo, D., Xia, X., Sun, J.: Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology (2017)
13. Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.G.: Is it a bug or an enhancement?: A text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, CASCON '08, pp. 23:304–23:318. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1463788.1463819
14. Kruchten, P.: The 4+1 view model of architecture. IEEE Software **12**(6), 42–50 (1995). https://doi.org/10.1109/52.469759
15. Madeyski, L., Kawalerowicz, M.: Software Engineering Needs Agile Experimentation: A New Practice and Supporting Tool. In: Software Engineering: Challenges and Solutions, *Advances in Intelligent Systems and Computing*, vol. 504, pp. 149–162. Springer (2017). https://doi.org/10.1007/978-3-319-43606-7_11
16. Kawalerowicz, M., Madeyski, L.: Continuous build outcome prediction: A small-n experiment in settings of a real software project. In: Proceedings of the 34th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems. Springer (2021)