*Preprint:* Marcin Kawalerowicz and Lech Madeyski, "Continuous Build Outcome Prediction: A Small-N Experiment in Settings of a Real Software Project" 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. 412–425, Springer International Publishing, 2021. DOI: 10.1007/978-3-030-79463-7_35

Preprint: https://madeyski.e-informatyka.pl/download/KawalerowiczMadeyski21.pdf

# Continuous Build Outcome Prediction: A Small-N Experiment in Settings of a Real Software Project

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] Department of Applied Informatics, Wroclaw University of Science and Technology, Poland. `Lech.Madeyski@pwr.edu.pl`

**Abstract.** We explain the idea of Continuous Build Outcome Prediction (CBOP) practice that uses classification to label the possible build results (success or failure) based on historical data and metrics (features) derived from the software repository. Additionally, we present a preliminary empirical evaluation of CBOP in a real live software project. In a small-n repeated-measure with two conditions and replicates experiment, we study whether CBOP will reduce the Failed Build Ratio (FBR). Surprisingly, the result of the study indicates a slight increase in FBR while using the CBOP, although the effect size is very small. A plausible explanation of the revealed phenomenon may come from the authority principle, which is rarely discussed in the software engineering context in general, and AI-supported software development practices in particular.

**Keywords:** software defect prediction · agile experimentation · continuous integration · machine learning

## 1 Introduction

While the raise of the complexity of software systems poses challenges that need to be addressed by Software Engineering processes, techniques and tools, the proposed novel approaches need to be evaluated against the earlier adopted ones. Software defect prediction (SDP) is an existing technique used to identify error-prone software modules. It is a cost-effective [1], software engineering assisting activity used to mitigate the problems that could arise if a software defect occurs. In our previous paper [2], we coined the idea of a lightweight version of Continuous Defect Prediction (CDP), named here Continuous Build Outcome Prediction (CBOP), that uses classification to label the possible build result (success or failure) based on historical data and metrics (features) derived from the software repository. In this paper, we build upon this idea and describe how we instantiated the CBOP practice in a real software project, as well as provide an empirical evaluation of the practice in real settings.

In CBOP, we use machine learning (ML) models that predict the continuous integration (CI) build results on the basis of historical CI results (success or failure) combined with metrics harvested from the software repository. A software developer is equipped with a set of tools that deliver continuous feedback by overseeing his actions.

Metrics are calculated on the fly and sent to the prediction model that checks if the changes developer is currently making in the source code might lead to a problem during the integration performed on the CI server. If a possible problem is detected, the feedback might lead the developer to be more cautious and to fix the problem before it manifests itself as a failing build.

In this article, we discuss the current state of knowledge and technology in the area of SDP (Section 2). We describe how the experiment aimed to evaluate the usefulness of CBOP in real-world, industrial settings was planned (Section 3) and executed (Section 4). We present the experiment results (Section 4.3). Discussion is presented in Section 5. Conclusions are presented in Section 6.

## 2 Background

It is not easy to automatically detect a software defect. That is why using machine learning to aid software defect prediction fascinated researchers for a long time (see, e.g., [3], [4], [5]). The idea was refined later in [6], [7], [8] and called just-in-time quality assurance or just-in-time defect prediction.

From one side, we build on top of a described and prototyped tool [9], [3] that performs defect prediction directly in the Integrated Development Environment (IDE). From the other side, we are using the unambiguous data from CI [10] server as a source of buggy/clean information and synthesizing it with the code metrics mined from the software repository.

Until now, the defect prediction approaches were based mainly on lexical examination of the commit message or the information from the bug tracking software [11]. We are classifying the change as buggy or not based on the result of the build on the CI server. We are using the build server as a definite source of information (oracle) about the "bug". If the build fails, we can presume the introduced change was buggy. This information is then fed into the model to improve it further. Finally, the effect we get is a constantly learning model based on unambiguous data derived from the build server.

Having the model ready, we are able to give the developer continuous real-time feedback in the form of build outcome prediction. Using this feedback, the developer can assess if the changes, he is working on, are likely to introduce a bug into the project or not. We are achieving this by exposing the build outcome prediction model to the IDE the developer is working with. We have an IDE extension (add-on) that continuously communicates with the model sending the project measurements and receiving prediction in exchange. The prediction then is displayed in the developer IDE. This could make him aware that the changes he is introducing to the project might result in a build fail. Consequently, that might lead to a more careful examination of the changes and rule out the defect(s) that would lead to a failing build. The idea of the technique we are proposing is presented in Figure 1. The discrete process of repeatedly building the software, training the model and obtaining the predictions is enclosed in an uninterrupted feedback loop. Deriving the analogy from CI the continuous nomenclature was used for CDP and CBOP.

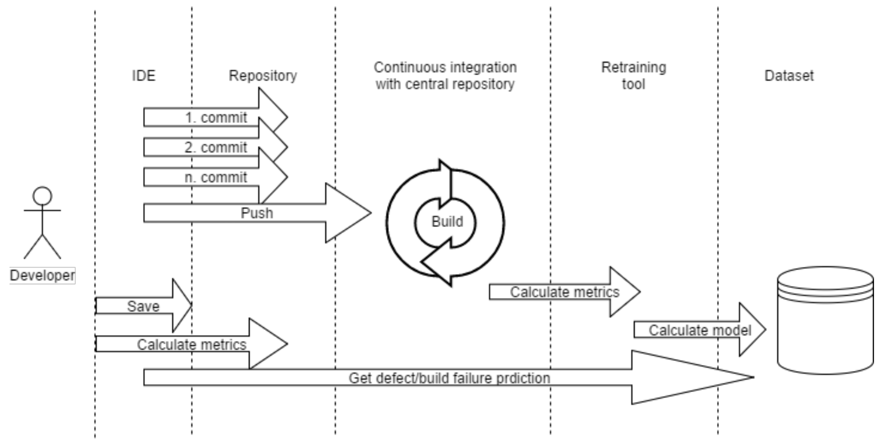

**Fig. 1.** Continuous build outcome prediction diagram

## 3 Experiment plan

### 3.1 Scoping

We define the goal of our experiment as follows [12]: *Analyze* continuous build outcome prediction (CBOP) practice *for the purpose of* evaluation of the practice *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 other words, we want to conduct the experiment to get the answer whether the new CBOP practice will reduce the rate of failing builds in a day-to-day work on an industrial-grade software project. We will have one object (a software project) and multiple subjects/participants (a software development team). According to the terminology used by Wohlin et al. [12], we will conduct "Multi-test within object study". Technically, it will be a quasi-experiment as the object, the software project, and the subjects (i.e., the software developers) will not be chosen randomly. Nevertheless, we will use randomization techniques in our quasi-experiment.

### 3.2 Context selection

The experiment will be conducted in vivo on an ongoing real business driven software project. Technically, it is a project consisting of two subprojects. One is a Microsoft .NET based project (ProjDotNet) in its 4th year of development, while the second project is a web-based Angular client (ProjWeb) that uses the ProjDotNet as an application programming interface (API). It is a software for banks and leasing companies. It is used in some large financial organizations on the German-speaking market. The software covers the complete loan and leasing contract lifetime from calculation, through offer creation to contract and contract recalculation, including after-contract object management and resell. The software was originally written in

Clarion (fourth-generation programming language). New software features are now added to the software using Microsoft .NET and the C# language. They contain Windows Communication Foundation(WCF)-based Web services and interoperability managed code calls using Component Object Model (COM). The other part of the software is an Angular based Web client that uses the .NET services as an API layer. Table 1 shows some data about the ProjDotNet and ProjWeb projects derived from it at the end of June 2018. According to the classification by Wohlin et al. [12], we

**Table 1.** Project details

| | |
|---|---|
| Number of distinct committers | 18 |
| Actively working in project | 6-8 developers/week |
| The total number of commits | over 10 000 |
| Total number of files | over 9 000 |
| Total Lines of Code | over 4 500 000 |
| Average build commits per active week | approx. 60 |

have an on-line project, involving professional developers, specific, and real problem.

The CBOP practice was not introduced to the project before the start of the experiment. There was neither defect prediction nor build failure prediction used in this project from its beginning. The build results recording was started on 2018-06-20 and lasted until 2018-10-02.

### 3.3 Hypothesis formulation

We formulate the following hypotheses:

H0 (null hypothesis): Using the CBOP practice (aka CBOPon) does not influence the Failed Build Ratio (FBR).

H1: Using CBOP reduces FBR.

### 3.4 Variables selection

In the experiment we will control the time when we apply CBOP or not (independent variable in our experimental design). We will make the predictions available to the developers only in certain times through the toolset we provide. The idea is to turn the prediction on (i.e., use CBOP) and off, labelled as A and B, on a workday basis using a randomization schema. We will have days where the prediction is available to the developer and days where it is not. The experiment will be conducted during 4 weeks of real development.

The build failure count will be used to calculate FBR. Every push to the central repository ends with the build on the CI server. The possible results one may observe on Jenkins CI are: SUCCESS, FAILURE, UNSTABLE, ABORTED, NOT_BUILD[3].

[3] Jenkins CI Build results: http://javadoc.jenkins-ci.org/hudson/model/Result.html

Only the results of type FAILURE are counted as failed builds. We do not count UNSTABLE builds as failures as they are not necessarily associated with failures but, e.g., bad smells in production code or tests. The ABORTED and NOT_BUILD are not counted as failures as well. As a result, we calculate the failed build ratio (FBR) as follows: $FBR = \frac{\#FAILURE}{(\#UNSTABLE + \#ABORTED + \#NOT\_BUILD + \#SUCCESS + \#FAILURE)}$.

## 3.5 Selection of subjects

The experiment will be conducted on a sample of the population of software developers. The company we are conducting the experiment in is a subcontractor of a larger software development company. The subcontractor leases 4 resources to its partner. One resource is 160 hours of work in one calendar month. It amounts to 640 hours of work (all of them not necessarily working full hours on that specific project). The work is conducted by 5-7 people. 4 to 5 software developers, software tester and a user experience specialist. All developers involved in the project are professionals with masters or bachelors degree in computer science with various experience levels: from 1 to 5 years of industrial experience as a software developer. The main task of all developers described here is the work in the subcontracted project described in detail in Section 3.2.

On the contractor side, there are 4 to 5 developers and one intern involved in that project. The developers are working full time on that project, while the intern is working only part time. One of the developers on the contractor is a software architect with 5+ years of experience and the three remaining developers are juniors with 1 to 3 years of experience.

Because we are interested in the largest possible sample of subjects, we used a convenience sampling technique to select the subjects – we selected all available subjects involved in the project. For the planing phase of the experiment we used the sample size of 4 for all the calculations. We were sure that we will be able to conduct the experiment with at least 4 developers (working in the project on the subcontractor side). During the execution of the experiment we were able to incorporate additional developers on the contractor side (the number of participants amounted 9).

## 3.6 Choosing experiment design type

In essence, we have here an experiment with one factor (the use of CBOP) and two treatments (presence or absence of the prediction). The participation of four developers from the contractor side in the experiment was certain. This rather small number of participants led us to use a small-n experimental design to study the effect of CBOP.

Dugard et al. [13] provided a list of experimental designs for single-case and small-experiments. Careful examination of the conditions for those experiments enabled us to find a proper design for our situation: "A small-n repeated-measures design with two conditions and replicates". We have at least two participants and two conditions to compare (A - No build outcome prediction available for the developer, B - build outcome prediction available for the developer). Each participant will receive each of the conditions on at least two occasions and it is possible to assign the conditions randomly.

Because we have expected at least 4 participants, we have not considered any of the “single-case” designs provided by Dugard et al. [13]. Other designs (Small-n one-way design, A small-n repeated-measures design, Two-way factorial small-n design) have been considered and were rejected as we knew that we will be able to measure the participants multiple times, but we will not have multiple levels of factors.

Because we will be able to change the condition multiple times, we have not considered the phase AB or ABA designs. They have their limitations making them more suitable for drug trials or medical experiments. Such trials need to have a clear distinction between the “intervention” (B after A) or even “withdrawal” (A after AB) and the baseline (A). Where we do have a distinction between the phases, we do not need to be that careful about the negative outcomes of our experiment. Thus, we will not conduct “The multiple baseline AB design” or “Multiple baseline ABA design” although technically possible in our case but too limited. In our case, a frequent succession of different conditions will be possible, thus an ”Alternate design” will be a better choice according to Bulte and Onghena [14].

We will be able to apply each treatment multiple times using a randomization schema, giving us the possibility to perform several observations (replicates) on each condition.

Choosing the right period to alternate between the phases was not an easy task. We have gathered some actual data from 6 weeks preceding the start of the experiments on the rate of the various build results. We have come up with an average of 54 builds per week with six failed builds per week. Having this data, we decided that a day will be a good choice for the period to alternate between the phases in our experiment. It should give us enough data to draw meaningful conclusions.

Next, we have proceeded to calculate the possible power of the experiment. We expected to have a least 4 participants in our team and 4 weeks to perform the experiment, giving us 20 observation days (assuming 5 working days per week).

## 4 Execution

### 4.1 Experimental setup

We have a working experimental CBOP setup in a commercial software project. The CBOP deployment in the project was possible thanks to our own dedicated tool called Jaskier [15]. Detailed information about the experiment instrumentation including the model prediction model creation is available in an online appendix [16].

### 4.2 Validity evaluation

In accordance with Wohlin et al. [12], we discuss threats to validity of our research.

The threat to the internal validity that applies to our research is that a subject may react differently as time passes (maturation). We think this threat is to a large extent addressed by the CBOP on (aka CBOPon) / off (aka CBOPoff) randomized assignment, see Table 2.

The threats concerning the problems to generalize the results of the research to a wider population of software developers are threats to external validity. The threat of "interaction of selection and treatment" is addressed to some extent by the fact that in both teams there is a wide distribution of experience and knowledge (from juniors to seniors), good diversity in culture and educational background (two countries, different education paths). The threat of "interaction of setting and treatment" is mitigated through the use of two technically different projects (.NET and Angular), the usage of industry standard tools (Visual Studio, VSCode) and a real, not toy-like, project. However, this threat needs to be considered because the technical differences between other types of projects might impact the ability to generalize the results. We conduct the experiment during the period of several days what mitigates the threat of "interaction of history and treatment". The time of day should not impede the observations. The research was conducted solely in a real-life software development project, what makes the external validity considerations much less critical.

Another type of threats we have considered are construct validity threats. They concern generalizing the results of the experiment to the ideas behind it. To address "inadequate preoperational explication of constructs" we have defined, as clearly as possible, what we are looking for - less failing builds. We hope to widen the generalization to a broader concept of defect (not only failing build) prediction, but we decided to start with a problem of broken builds that developers have to deal with most often. Other threat that we think needs to be considered is the "restricted generalizability across constructs". It is important to check if the approach we are proposing will not effect the project in a negative way. For example, whether the usage of the prediction will not give the developers false beliefs and thus result in more careless committing. We have similar concerns regarding the social threats to the experiment. It is possible that the behavior of the software developers will change due to the usage of our CBOP tools. It might be that they will feel more secure while receiving positive predictions and thus they will more boldly commit insecure changes to the repository.

The last type of threats we have considered are the threats to conclusion validity. In our case, it is the ability to draw a correct conclusion about the influence of CBOP on FBR in a software development project. Important threat is a potentially low statistical power of the experiment (because of the limits in the number of participants involved in the study). We tried to mitigate it by involving as many developers as we could. Finally, we were able to use data from 9 individual developers.

### 4.3 Analysis

In the experiment, we have analyzed 310 project days worth of data coming from a total of 9 developers. Table 2 shows the assignments for individual developers. The experiment assignments were prepared 40 days in advance. Different developers worked different number of days in the project under investigation (because of sick days, vacations, different project assignments etc.). Some days the developers did not make any commit resulting in a build. Those days were omitted from the results. Although in those days the developers were assigned a phase. If a developer participated in the experiment longer than 40 days, the sequence started from the beginning. If a developer participated in less than 40 days then only the days in which she or he participated were taken into account. The developers were working simultaneously in Visual Studio

**Table 2.** CBOPoff (A) / CBOPon (B) assignments for individual developers

| Developer no. | Assignments plan | Days in exp. | Obs. count | Failure build count | Total build count |
|---|---|---|---|---|---|
| Developer 1 | AAAAAABBBBBBBBBBBBBB BBBBAAAAABBBBBBBBBBB | 44 | 33 | 16 | 63 |
| Developer 2 | AAAAAAABBBBBBBBBBBBB BBBBAAAAAAAAABBBBBBB | 27 | 20 | 3 | 32 |
| Developer 3 | AAAAAAAABBBBBBBBBBBB BBBBAAAAAAAAABBBBBBB | 37 | 27 | 1 | 47 |
| Developer 4 | AAAAAAAAAABBBBBBBAAA AAAAAAAABBBBBBBBBBBB | 60 | 84 | 58 | 227 |
| Developer 5 | AAAAAAAAAAABBBBBBBBB BBBAAAAAABBBBBBBBBBB | 43 | 29 | 1 | 65 |
| Developer 6 | AAAAAAAAAAAABBBBBBBB BBAAAAAAAABBBBBBBBBB | 54 | 61 | 39 | 134 |
| Developer 7 | AAAAABBBBBBBAAAAAAAA AAAAABBBBBBBBBBBBBBB | 35 | 50 | 27 | 124 |
| Developer 8 | AAAAAAAAABBBBBBBBAAA AAAABBBBBBBBBBBBBBBB | 40 | 55 | 59 | 239 |
| Developer 9 | AAAAABBBBBBBBBBBBBBB BBBAAAAAAABBBBBBBBBB | 46 | 51 | 26 | 139 |

on the back-end .NET services and in VSCode working on the Angular front web client for the services. The results were aggregated for both projects separately.

**Analysis of descriptive statistics** In total, CBOP was turned off during 173 days and turned on during 237 days of the experiment. FBR dependent variable, calculated as described in the Section 3.4, is plotted in Figure 2.

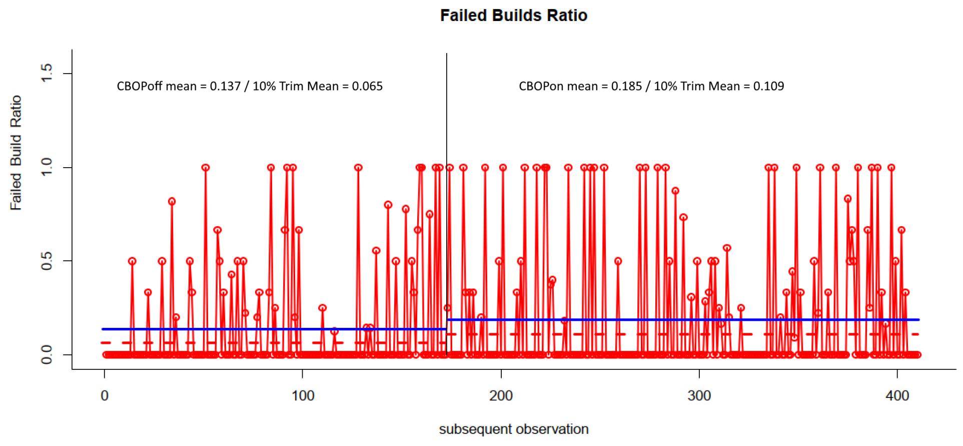

**Fig. 2.** Failed Build Ratio (FBR)

We see a slight increase in the mean of FBR when the CBOP was turned on, from 0.137 (CBOPoff) to 0.185 (CBOPon). Figure 3 shows the box plot for the same data. The median in both phases was 0.

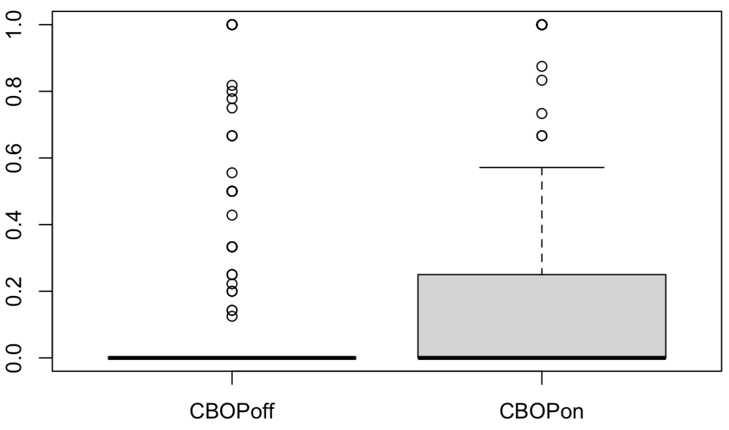

**Fig. 3.** Failed build ratio (Y-axis) box plot

As variances are not equal ($F = 0.70099$, $df1 = 172$, $df2 = 236$, $p-value = 0.01366$), we may use Welch's t-test, but the null hypothesis can not be rejected ($t = -1.5579$, $df = 400.34$, $p - value = 0.12$). Unfortunately, the data are autocorrelated in the CBOPon phase and have a visible trend in the CBOPoff phase. In such situation, transformation using differencing is recommended. Unfortunately, it does not remove autocorrelation. Hence, the Welch's t-test may not be reliable and we proceed further analyzing a robust measure of central location and effect sizes.

Mean is not a robust measure of the central location and can be strongly influenced by outliers, especially when the number of observations is small. However, the 10% trimmed mean (which is a more robust measure of the central tendency) also increased from 0.065 (when CBOPoff) to 0.109 (when CBOPon). Standard deviation increased in a similar manner from 0.281 (when CBOPoff) to 0.335 (when CBOPon). We will discuss possible explanations of such behaviour in Section 5, but now we focus on the size of the observed effect. Table 3 contains descriptive statistics for all phases and developers.

**Effect size** It has become a recommended practice to assess the magnitude of a treatment effect (CBOPon vs CBOPoff in our case) using effect size measures as it gives a sense of practical importance [17,18,19]. To grasp the effect of CBOPon vs CBOPoff, we report the most common effect size measures:

– $ES$ calculated as the difference between the intervention CBOPon and baseline CBOPoff means divided by the standard deviation of the baseline ($ES = \frac{M_{CBOPon} - M_{CBOPoff}}{SD_{CBOPoff}}$),

**Table 3.** Descriptive statistics for each developer for CBOPoff (off) / CBOPon (on)

| | Obs. count | | Mean | | 10% trim mean | | Median | | SD | |
|---|---|---|---|---|---|---|---|---|---|---|
| | off | on | off | on | off | on | off | on | off | on |
| Developer 1 | 12 | 21 | 0.194 | 0.337 | 0.133 | 0.298 | 0.000 | 0.000 | 0.388 | 0.447 |
| Developer 2 | 5 | 15 | 0.000 | 0.133 | 0.000 | 0.077 | 0.000 | 0.000 | 0.123 | 0.352 |
| Developer 3 | 15 | 12 | 0.000 | 0.021 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.072 |
| Developer 4 | 43 | 41 | 0.216 | 0.118 | 0.168 | 0.069 | 0.000 | 0.000 | 0.290 | 0.233 |
| Developer 5 | 12 | 17 | 0.000 | 0.020 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.081 |
| Developer 6 | 29 | 32 | 0.254 | 0.269 | 0.215 | 0.216 | 0.000 | 0.000 | 0.380 | 0.414 |
| Developer 7 | 22 | 28 | 0.102 | 0.221 | 0.032 | 0.175 | 0.000 | 0.000 | 0.257 | 0.364 |
| Developer 8 | 24 | 31 | 0.044 | 0.214 | 0.013 | 0.146 | 0.000 | 0.000 | 0.123 | 0.350 |
| Developer 9 | 11 | 40 | 0.136 | 0.197 | 0.056 | 0.121 | 0.000 | 0.000 | 0.323 | 0.325 |

– $d-index$ uses a pooled standard deviation and thus may be more appropriate when the variation between the phases differs ($d-index=\frac{M_{CBOPon}-M_{CBOPoff}}{SD_{pool(CBOPoff,CBOPon)}}$).

Both values ($ES=0.16923$, $d-index=0.15156$) are below 0.87 and indicate "small effect size". However, when there is a trend in any phase, then both measures are not appropriate and we need to use more sophisticated effect size measures. In our case, there is a visible trend in the baseline CBOPoff, see Figure 4.

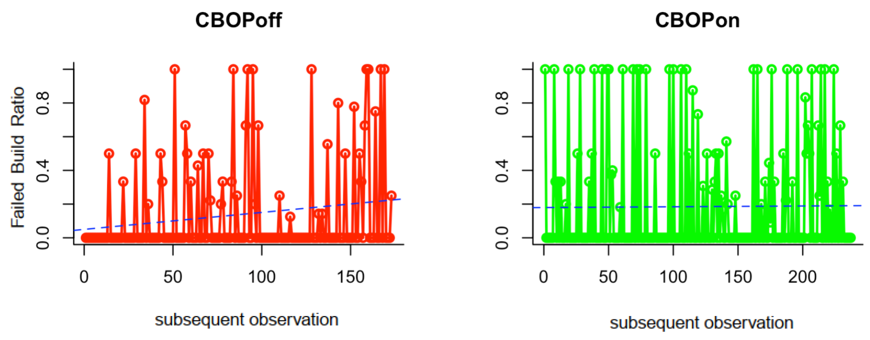

**Fig. 4.** Visualisation of trend

Hence, we report and rely on the following effect size measures:

– $PEM$ (the percentage of the data points in intervention phase (CBOPon) exceeding the median of the baseline phase (CBOPoff)) [20],
– $PAND$ (the percentage of all non-overlapping data) [21].

$PEM=0$ (i.e., lower than 0.5) can be interpreted as “not effective” according to Ma [20]. $PAND=0.578$ (i.e., lower than 0.69) can be interpreted as “debatable effectiveness” according to Parker et al. [21]. This indicates that CBOP is “not effective" or even its effectiveness is, contrary to the expectation, slightly in the opposite direction, i.e., may lead to a tiny increase in FBR.

## 5 Discussion

The experiment results indicate that our hypothesis that by using CBOP (CBOPon) one can positively influence the failing builds ratio was not reflected in results. By using our CBOP setup, developers were causing slightly more failing builds than without the CBOP in place (CBOPoff). The effect size was very small, but it prompted us to elaborate on plausible explanations. One of the possible reasons (we have considered as a threat before the experiment) is that developers equipped with a tool that was supposed to shelter them from the problems of failing builds became more careless. They start to commit a code of lesser quality because they had positive feedback about it.

This effect could be caused by the human tendency to use judgement heuristics while making decisions. It is generally easier for a human being to use simple strategies while finding solutions for a complex problem. One of those judgement heuristics is an authority principle. This principle was catalogued by Cialdini among "Six Principles of Influence" [22]. Authority principle says that humans tend to comply with the people they see to be in a position of authority (like shown in [23]). The authority principle applies also to non-human authorities like a law or legal precedent ([24]). Our software tool Jaskier can be seen as a form of non-human authority. It is prominently visible in IDE and gives informed predictions about the coming build result. What if the software shows a false positive (it informs the developer that everything will be fine, but in fact there is a problem in the code base)? It might be the case that in such situations developers use a simple judgement heuristic and do not review the code sufficiently but check it in.

The authority principle explanation needs a rigorous test in order to fully explain the observed phenomena. It might also be interesting to look on the result from another analysis level and analyze the developer cognitive functioning (learning, thinking, reasoning, remembering, problem solving, decision making, and attention) [25].

## 6 Conclusions and future work

We have demonstrated the effect of employing the new CBOP practice (for predicting failures on the CI server) in a real software project. Based on historical CI data (build success vs. failure information) and metrics calculated from the software repository, we are able to create a prediction model for a build failure. We are matching the historical CI results with the commits that led to the CI build and based on that data we create the classification model. This model is used to predict how dangerous the changes the developer introduced to the project are in respect to the build outcome.

We have build a set of tools that form a practical implementation of the CBOP idea. We used CBOP in order to evaluate it in a commercial software project.

We are working to improve the prediction models, although using random forests we are currently reaching 95% of prediction accuracy (based on k-fold cross-validation).

According to developers, the acceptance of the new practice and the supporting toolset is beyond doubt if the performance of the prediction model is high.

Although we were not able to support our hypothesis that "Using CBOP reduces FBR" by the obtained results, we plan to extend the toolset to capture brother plateau of defects and reevaluate brother CDP practice. The social behavioral explanation (impact of non-human authority) of the results might also to be narrowed to cognitive functioning of a single developer in its environment. More rigorous testing of the explanation is needed in order to fully explain the phenomenon.

The more sophisticated effect size measures had to be used because of the trend in the CBOPoff baseline. They indicate that CBOP is "not effective" (according to the PEM measure) or its effectiveness is "debatable" (according to the PAND measure). Descriptive statistics also suggest that the effectiveness of CBOP is low and slightly in the opposite direction than expected (i.e., CBOP may in fact lead to increase in FBR) but increasing FBR in the CBOPoff baseline (see Figure 4) does not allow to come to strong conclusions. The data obtained for this experiment is available though a download[4].

We reached a plausible explanation of the reveled phenomenon building upon the authority principle, which is rarely discussed in the software engineering context in general, and AI/ML-supported software development practices in particular, but we think deserves attention and should be taken into account with accelerating use of AI/ML techniques.

## References

1. Arora, I., Tetarwal, V., Saha, A.: Open issues in software defect prediction. Procedia Computer Science **46**, 906–912 (2015). https://doi.org/10.1016/j.procs.2015.02.161
2. 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
3. Kim, S., Whitehead Jr., E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. **34**(2), 181–196 (2008)
4. 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
5. 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)
6. 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)
7. 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)
8. 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)

[4] https://doi.org/10.6084/m9.figshare.14222273.v1

9. 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)
10. 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
11. 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)
12. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Computer Science. Springer (2012)
13. Dugard, P., File, P., Todman, J.: Single-case and Small-n Experimental Designs: A Practical Guide to Randomization Tests, 2nd edn. Routledge (2012)
14. Bulté, I., Onghena, P.: An r package for single-case randomization tests. Behavior research methods **40**, 467–478 (2008)
15. Kawalerowicz, M., Madeyski, L.: Jaskier: A Supporting Software Tool for Continuous Defect Prediction Practice. In: Proceedings of the 34th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems. Springer (2021)
16. Kawalerowicz, M., Madeyski, L.: Appendix to "Continuous Build Outcome Prediction: A Small-N Experiment in Settings of a Real Software Project". https://madeyski.e-informatyka.pl/download/KawalerowiczMadeyski21CBOPApp.pdf
17. Ferguson, C.J.: An effect size primer: A guide for clinicians and researchers. Professional Psychology: Research and Practice **40**(5), 532–538 (2009)
18. Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., Pohthong, A.: Robust Statistical Methods for Empirical Software Engineering. Empirical Software Engineering **22**(2), 579–630 (2017). https://doi.org/10.1007/s10664-016-9437-5
19. Madeyski, L.: Test-Driven Development: An Empirical Evaluation of Agile Practice. Springer, (Heidelberg, London, New York) (2010)
20. Ma, H.H.: An alternative method for quantitative synthesis of single-subject researches. Behavior Modification **30**(5), 598–617 (2006)
21. Parker, R.I., Hagan-Burke, S., Vannest, K.: Percentage of All Non-Overlapping Data (PAND) : An Alternative to PND. The Journal of Special Education **40**, 194–204 (2007)
22. Cialdini, R.: Influence: The Psychology of Persuasion. Collins Business Essentials. HarperCollins e-books (2009)
23. Bickman, L.: The social power of a uniform. Journal of Applied Social Psychology **4**, 47–61 (1974). https://doi.org/10.1111/j.1559-1816.1974.tb02807.x
24. Schneider, A., Honeyman, C., of Dispute Resolution, A.B.A.S.: The Negotiator's Fieldbook. American Bar Association, Section of Dispute Resolution (2006)
25. Fisher, G.G., Chacon, M., Chaffee, D.S.: Chapter 2 - theories of cognitive aging and work. In: B.B. Baltes, C.W. Rudolph, H. Zacher (eds.) Work Across the Lifespan, pp. 17–45. Academic Press (2019). https://doi.org/https://doi.org/10.1016/B978-0-12-812756-8.00002-5