Studying the Impact of Continuous Delivery Adoption on Atoms of Confusion Rate in Open-Source Projects

Diego N. Feijó
Federal University of Ceará
Fortaleza, Ceará, Brazil
diegofeijo@alu.ufc.br

Carlos D. A. de Almeida
Federal University of Ceará
Fortaleza, Ceará, Brazil
diego.andrade@ufc.br

Lincoln S. Rocha
Federal University of Ceará
Fortaleza, Ceará, Brazil
lincoln@dc.ufc.br

ABSTRACT

Atoms of Confusion (AoC) are indivisible code patterns that may cause confusion for developers when trying to understand them, and that have less confusing equivalent patterns. Previous works suggest it is a good practice to avoid them. While there are studies on AoC relating them to bugs, there is not much about their relationship with the practices of Continuous Integration and Continuous Delivery (CI/CD). Since CI/CD is generally praised as a group of good practices, related to better code being released reliably and faster to clients, there's a possibility that the presence of CI/CD would also impact the presence of AoC, possibly making them less prevalent since they can be problematic to development processes. To clarify this relationship, we analyzed 10 open-source long-lived Java libraries and 10 open-source Java projects for Android, to see if there was any difference in the AoC rate before and after the implementation of CI/CD. Our results show the AoC rate changed for all projects, but we could not find a statistically relevant relationship between these changes and CI/CD.

KEYWORDS

Software Engineering, Atoms of Confusion, Continuous Integration, Continuous Delivery, Mining Repositories, Open-Source Software

1 INTRODUCTION

Software engineering is trying to find solutions to produce software quickly and with more quality. This is also true for open-source projects widely used by end users and companies [12, 21], and they keep growing in scale. As such, developers should learn and adopt practices and methods that would help them in this regard.

Continuous Integration [7] and Continuous Delivery [10] are a set of practices in software engineering, which consists of pipelines that automatize the building, testing, and delivery processes of software, making it a faster and more reliable way to produce valuable software artifacts in short cycles [4]. Since quickness is relevant, it is undesirable to lose more time than necessary on activities such as understanding source code, which is an essential software developing task, but also very time-consuming, having developers spend 50% of their time on it [15, 20].

To avoid consuming even more time on this kind of task, it's relevant for projects to minimize the rate of confusing code they contain. Confusion is the lack of certainty a developer has about the execution of a piece of code. With this in mind, Atoms of Confusion are defined as indivisible code patterns that are easily identifiable, likely to cause confusion, and can be replaced by another pattern more unlikely to cause confusion [3].

Since AoC are known to confuse, there may be a relationship between the implementation and practice of CI/CD and the atoms' prevalence or lack thereof. As they can confound, this could also cause problems related to slowness in the software performance, maintenance, and even the introduction of bugs since the code is harder to understand. Some works have already studied the prevalence of AoC [9, 14], and even their impact during development [2, 9]. Results vary, but if AoC is a problem, is it possible that the practice of CI/CD could reduce its prevalence in the long run?

In this paper, we conducted a study to verify if the practice of CI/CD has any impact on the presence of AoC in open-source projects. To measure this presence, we used static analysis, using tools - BOHR and JMetriX - made with the Spoon [16] Java library, to extract the AoC rate, which we define as the number of AoC per line of code, from 20 different open-source Java projects, 10 being long-lived known Java libraries, and the other 10 being Android projects. These projects and their versions were filtered and selected by defined criteria to guarantee their relevance and fulfill this paper's goal. We used the Wilcoxon Signed-Rank Test to compare the data, and the results we found imply a lack of statistically relevant connection between the implementation of CI/CD and the AoC rate.

2 BACKGROUND

2.1 Atoms of Confusion

As said before, Atoms of Confusion are easily identifiable and indivisible patterns of code capable of making the developer misunderstand the code they're in and how it executes. While the original study and list of Atoms of Confusion was based on the C programming language [8], different programming languages have different Atoms of Confusion candidates, and there are already studies that proposed their own lists of AoC candidates for specific programming languages [3, 5, 11, 18]. By definition, AoCe can be transformed into other patterns of code that function in the same way and which are less likely to be confusing, making the use of AoC unnecessary. However, they are still prevalent in software projects from various contexts [9, 14], even finding themselves overlapping with recommendations from popular code style guides [8]. AoC have a tendency to grow in numbers when the project grows in size, even disproportionally so when compared to the increase in lines of code [14]. They were also shown to be connected to the presence of bugs and frequently appeared in bug fixes [9], pointing to the possibility of them being dangerous and a problem that projects may want to get rid of. However, their impact is lacking, or at least unclear, in the context of code reviews,

seemingly not causing confusion comments, nor being removed on pull requests [2].

2.2 Continuous Practices in a Nutshell

In an effort to properly separate and define the continuous practices and DevOps, Ståhl et al. [17] proposed a few ways to understand the practices, as a set of separate definitions. This was done because of the great ambiguity these terms have both in the industry and in the literature. Continuous Integration is the frequent integration of developers' works, usually daily at least. This is a practice dependent on the developers' behavior since they are the ones who actively integrate their work. Continuous Delivery is treating each change made to the project as a potential release candidate, in other words, it needs to be properly tested and verified by a continuous delivery pipeline. This is a development process, not connected to a developer's actions since the pipeline is automatized. In fact, it is possible for a Continuous Delivery pipeline to be present, while some developers do not act with Continuous Integration in mind. Continuous Deployment is constantly and rapidly placing release candidates, previously evaluated during Continuous Delivery, in a production environment, usually for customer use.

According to these definitions, our focus is on the Continuous Delivery aspect of "CI/CD", or just CD. Liu et al. [13] often uses the complete acronym. Still since their study's method of identifying "CI/CD" was the presence of CI/CD services and pipelines, it's implied their focus is the same as ours in this aspect.

3 STUDY METHODOLOGY

3.1 Research Goal and Question

This paper's main objective is to know the impacts that CI/CD may have on the prevalence of AoC. Our research question will summarize this goal.

RQ. Is there a statistically significant difference between the rate of Atoms of Confusion before and after CI/CD adoption in open-source Java projects?

The AoC rate is the metric we chose to check the prevalence of AoC, and it was the chosen metric because of the tendency for the number of AoC to grow as the projects grow in size [14]. This means the number of AoC would probably not decrease with the implementation of CI/CD, not while the project kept increasing in size. As such, the AoC rate would instead represent the presence of AoC independently from the project size.

With this goal in mind, we first selected our projects following a set of criteria, for a total of 20 projects. We then extracted the data from these projects with two tools of static analysis, BOHR [14] and JMetrix[1] to acquire the metrics to be used during the analysis to get the results we are searching for. Finally, we discuss the different results and their implications.

3.2 Selection of Projects

We select 20 open-source projects to serve as subjects of our empirical investigation, divided into two groups: long-lived Java libraries (in the first 10 lines of Table 1) and Java-based Android projects (in the last 10 lines of Table 1). All of them are from open-source repositories from GitHub, and were chosen based on the following criteria, based on the one used by Fairbanks et al. [6]: (i) the projects must have adopted CI/CD at some point in their development; (ii) the projects must have been active as recently as 2022; (iii) they must have at least 25 stars and at least 2 contributors, to avoid personal and school projects; and (iv) the projects must have at least 8 version tags, otherwise the project's history with CI/CD and AoC would be too short, threatening the statistical analysis.

Table 1: GitHub's Information of Studied Projects.

Project	#Stars	#Tags	#Commits
commons-lang	2,503	96	7.270
commons-dbcp	312	66	2,827
struts	1,216	143	6,658
commons-codec	409	44	2,423
commons-bcel	216	35	2,508
commons-compress	282	76	4,129
commons-configuration	179	77	3,857
commons-net	211	75	2,918
freemarker	874	44	2,343
commons-vfs	197	55	3,656
infinity-for-reddit	3,461	118	2,033
gestureviews	2,325	16	432
discreet-launcher	167	61	639
xupdate	2,134	32	246
colorpickerview	1,417	19	278
opentracks	678	141	5,311
presencepublisher	70	50	206
asteroidossync	91	28	1,088
unexpected-keyboard	619	28	616
shitter	197	108	1,678

For the Java libraries, we used the projects from Almeida et al. [1]. These repositories were curated from the SmartShark dataset[2], and filtered by checking the presence and proper use of CD. They are all from Apache and also fit the criteria. Some projects that did not pass all the filters in this work were manually checked and were considered candidates if they passed the criteria we established.

For the Android projects, we needed to filter them ourselves from the projects identified by Liu et al. [13] as having adopted a CI/CD service. Their dataset contained a large number of Android repositories from three different sites. We limited ourselves to those from GitHub, around 4,000, filtering only the repositories that passed the criteria. After the filtering, we had 240 projects to choose from.

3.3 Atoms of Confusion Rate Metric

To compare the AoC rate (ACR) between the periods with and without CI/CD, we first determined how that would be measured. For that, we extract two metrics from each considered release: the number of atoms of confusion (NAC) and the number of lines of code (LOC). Next, we calculate the ratio between NAC and LOC to compute the AoC rate (ACR = NAC ÷ LOC). We compute the ACR metric for all

[1] https://github.com/lincolnrocha/JMetriX

[2] https://smartshark.github.io/

considered releases in each project and group it into two periods, AoC rate before and after CI/CD adoption. Finally, we compute the statistical mean and median of the ACR metric for each period and project, and use them as proxies to statistically compare the rate of AoC before and after CI/CD adoption across the studied projects.

3.4 Data Mining

As mentioned before, we already had a list of candidates to choose the long-lived libraries from. Still, to get the Android projects, we first needed to analyze the list of projects from GitHub and filter only the ones that fit the criteria. For that, we used Python, and the ghAPI[3], a Python library to communicate with GitHub's API, and filtered the projects from Liu et al. [13] with our established criteria. After the filtering, we randomly chose projects avoiding ones that had implemented CI/CD since the first version, making it impossible to compare the before and after of its implementation or those that implemented it so recently and have no history with CI/CD, until we get ten projects.

After choosing the projects, we downloaded several versions from each, trying to balance the number before and after the implementation of CI/CD. These versions each had their code statically analyzed with Java programs made with Spoon: BOHR and JMetriX. Spoon is an open-source Java library that analyzes and transforms Java source code. BOHR, created by Mendes et al. [14], is a tool made to identify AoC and extract data related to them from the source code of Java projects. The AoC it can identify are based on the AoC list for the Java programming language suggested by Langhout and Aniche [11], and are shown in Table 2, as well as the equivalent less-confusing pattern for each. JMetriX is a tool made to extract general metrics and information about Java source code. It was used to extract the number of lines of code (LoC) metric from the projects.

3.5 Data Analysis

From the mining, we got the number of LoC and the number of AoC for each version of each project we selected. With this, we calculated the ACR. Since the number of AoC is considerably lower than the number of LoC, we use this metric on a 10^{-3} scale.

With this metric in hand for each project, we calculated means and medians, for versions "Before CI/CD" and "After CI/CD" since we wanted to compare the possible impact of CI/CD on the ACR considering the version history. After we had the means and medians per project, we put them together, and compared them "Before CI/CD" and "After CI/CD". To make this comparison, we used the Wilcoxon Signed-Rank Test to compare the data and to check if there's a relation between them. For this test, the result is a p-value which must be 0.05 or lower for the null hypothesis, that there's no relation between data, to be rejected and statistically relevant.

4 STUDY RESULTS AND DISCUSSION

4.1 Research Question Answer

To answer our research question, we first computed the mean and median of the ACR metric for each studied project and summarized them in Table 3. Next, we employ the Wilcoxon Signed-Rank Test

[3]https://ghapi.fast.ai/

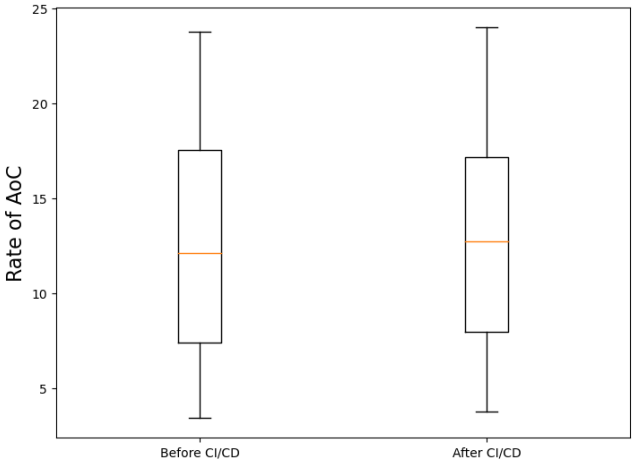

Figure 1: The boxplot of rates of AoC before and after CD adoption.

to compare them before and after CI/CD adoption. Thus, one can verify whether there are statistically significant differences between them.

The Wilcoxon Signed-Rank Test is a nonparametric test to compare paired data samples. First, for the ACR's mean comparison, we defined the null and alternative hypotheses as follow: $H_0^{\overline{x}}$: $\overline{x}(\mathtt{ACR_b}) = \overline{x}(\mathtt{ACR_a})$ (it means that there is no statistical difference between ACR before and after CI/CD adoption), $H_1^{\overline{x}}$: $\overline{x}(\mathtt{ACR_b}) > \overline{x}(\mathtt{ACR_a})$ (it means that ACR before is significantly higher than ACR after CD adoption), and $H_2^{\overline{x}}$: $\overline{x}(\mathtt{ACR_b}) < \overline{x}(\mathtt{ACR_a})$ (it means that ACR before is significantly lower than ACR after CD adoption). Next, for the ACR's median comparison, we defined the null and alternative hypotheses as follows: $H_0^{\widetilde{x}} : \widetilde{x}(\mathtt{ACR_b}) = \widetilde{x}(\mathtt{ACR_a})$, $H_1^{\widetilde{x}} : \widetilde{x}(\mathtt{ACR_b}) > \widetilde{x}(\mathtt{ACR_a})$, and $H_2^{\widetilde{x}} : \widetilde{x}(\mathtt{ACR_b}) < \widetilde{x}(\mathtt{ACR_a})$. The hypotheses $H_0^{\widetilde{x}}$, $H_1^{\widetilde{x}}$, and $H_2^{\widetilde{x}}$ are equivalent to the $H_0^{\overline{x}}$, $H_1^{\overline{x}}$, and $H_2^{\overline{x}}$ hypotheses.

Table 4 summarizes the statistical test results. After applying the statistical test to all projects, we were able to identify that in both cases the null hypotheses ($H_0^{\overline{x}}$ and $H_0^{\widetilde{x}}$) could not be rejected (i.e., any of the statistical test results have a significance level of $\alpha < 0.05$). Thus, the alternative hypotheses $H_1^{\overline{x}}$, $H_1^{\widetilde{x}}$, $H_2^{\overline{x}}$, and $H_2^{\widetilde{x}}$ are all rejected. Therefore, the statistical results indicate that the adoption of CI/CD has no significant impact on the AoC rate.

4.2 Discussion

Some projects, like commons-dbcp, actually had their ACR reduced after the implementation of CI/CD, going against the tendency of increase, but the behavior is not consistent throughout the projects, with some even having their rates increased. The magnitude of the changes is also inconsistent. Visualizing the data, we can actually see that the impact, when considering all projects, was practically none. The rate of AoC changed for all projects, as mentioned before and shown in Table 3. However, when analyzing the boxplot in Figure 1 of both before and after CI/CD, we see the difference in distribution between them is hard to visualize, as they are both

Table 2: List of Atoms of Confusion identifiable by the BOHR tool. Adapted from Mendes et al. [14]

Atom of Confusion Name	Acronym	Snippet with AoC	Snippet without AoC
Infix Operator Precedence	IOP	`int a = 2 + 4 * 2;`	`int a = 2 + (4 * 2);`
Post-Increment/Decrement	Post-Inc/Dec	`a = b ++;`	`a = b ;` `b += 1 ;`
Pre-Increment/Decrement	Pre-Inc/Dec	`a = ++b ;`	`b += 1 ;` `a = b ;`
Conditional Operator	CO	`b = a == 3 ? 2 : 1;`	`if ( a == 3){ b = 2;}` `else {b = 1;}`
Arithmetic as Logic	AaL	`(a - 3) * (b - 4) != 0`	`a != 3 && b != 4`
Logic as Control Flow	LaCF	`a == ++a > 0 || ++b > 0`	`if (!( a + 1 > 0 )) {b += 1;}` `a += 1`
Change of Literal Encoding	CoLE	`a = 013;`	`a = Integer.parseInt("13", 8);`
Omitted Curly Braces	OCB	`if (a) f1 ( ); f2 ( );`	`if (a){ f1( ); } f2( ) ;`
Type Conversion	TC	`a = (int) 1.99f;`	`if (a){ f1( ); } f2( ) ;`

Repurposed Variables	RV	`int a [] = new int[5];`
		`a[4] = 3;`
		`while (a[4] > 0) {`
		`  a[3 - v1[4]] = a[4];`
		`  a[4] = v1[4] - 1;}`
		`System.out.println(a[1]);`
			`int a [] = new int[5];`
			`int b = 5 ;`
			`while (b > 0) {`
			`  a[3 - a[4]] = a[4];`
			`  b = b - 1;}`
			`System.out.println(a[1]);`

very similar. The boxplot shown considers the mean rates, but the boxplot for the medians is virtually the same.

5 THREATS TO VALIDITY

We will use the threats to the validity of our study as presented by Wohlin et al. [19]. Those are threats to conclusion, construct, internal or external validity.

Conclusion Validity. To deal with threats to our study's conclusion, we chose the Wilcoxon Signed-Rank Test so that our conclusion was statistically relevant.

Internal Validity. It is possible that some other variables could interfere by increasing or decreasing the rate of AoC in ways we do not understand. However, it is not the intention of this paper to imply causal relationships.

Construct Validity. There's a possibility of human bias being present when the projects are chosen. To avoid this, the manual choice of projects was made at random after the filtering, being invalidated only if the project did not fit our criteria. Other parts of our analysis were automated to avoid mistakes and bias.

External Validity. The sample size of this study is only 20 projects, which can threaten its generalization. To mitigate it, we tried to diversify our projects with not only long-lived Java libraries but also projects from another context entirely, that is, Android projects. We also filtered the projects, getting only relevant ones, while avoiding personal projects.

6 RELATED WORK

Gopstein et al. [8] introduced the concept of Atoms of Confusion as the smallest piece of code that can cause confusion in developers, making them misunderstand what the code actually does, which can lead to mistakes during development and when doing tasks. This work also focused on AoC in the context of the C programming language, while other works studied the AoC in different programming language contexts, such as Castor [3], with Swift; Langhout and Aniche [11], with Java; Torres et al. [18], with JavaScript; and [5], with Python. Other works also study the relationship of AoC with different metrics, such as Bogachenkova et al. [2] which analyzed the possible relation between AoC, code review, and pull requests.

Gopstein et al. [9] also made a study to check the prevalence of AoCs in the context of open-source C and C++ projects, while also studying the possible impact on the number of bugs, and number of bug fixes that contain atoms. Mendes et al. [14] similarly studied AoC and their prevalence in a specific context, open-source long-lived Java libraries, but also analyzed the co-occurrence of atoms, and created the tool for AoC detection we used in our study.

On CI/CD, Almeida et al. [1] studied the relation of CI/CD with Bug-fixing time, where it was found that there was a decrease in bug-fixing time after CI/CD implementation. Our work is a derivative of this paper, as we consider a different variable, within a different context but made our studying process in similar ways. Fairbanks et al. [6] verified the impact of CI/CD on commit velocity and number of reported issues, analyzing over 12, 000 repositories from GitHub, and GitLab, with roughly 4, 500 of them having CI/CD. Liu et al. [13] analysis was focused on the context of Android apps, analyzing more than 80, 000 repositories from GitHub, GitLab, and Bitbucket, and finding the presence on CI/CD in roughly 10% of them. Their focus was to check the extent of CI/CD adoption, and the use of different CI/CD services in the projects.

Table 3: Summary of projects and AoC rate statistics. ACR_b (ACR before CI/CD adoption), and ACR_a (ACR after CI/CD adoption). The $\overline{x}$ and $\widetilde{x}$ stand for statistical mean and median, respectively.

Project	$\overline{x}(\text{ACR}_\text{b})$	$\overline{x}(\text{ACR}_\text{a})$	$\widetilde{x}(\text{ACR}_\text{b})$	$\widetilde{x}(\text{ACR}_\text{a})$
commons-lang	11.73	12.29	11.54	12.52
commons-dbcp	6.17	4.00	5.72	3.77
struts	7.15	8.13	6.69	8.44
commons-codec	22.12	18.07	22.12	18.07
commons-bcel	7.46	7.20	7.51	6.85
commons-compress	23.77	20.53	23.47	20.61
commons-configuration	5.14	3.76	4.79	3.47
commons-net	10.64	12.09	11.16	11.56
freemarker	20.79	19.96	20.36	20.15
commons-vfs	7.11	6.99	7.40	7.00
infinity-for-reddit	13.34	17.04	13.27	16.99
gestureviews	22.97	23.99	23.15	24.15
discreet-launcher	3.42	7.53	3.47	8.49
xupdate	14.96	15.26	15.06	15.20
colorpickerview	9.39	11.09	9.39	11.09
opentracks	16.44	17.48	16.25	17.53
presencepublisher	21.17	13.32	20.25	13.86
asteroidossync	12.47	13.17	12.96	13.17
unexpected-keyboard	12.82	15.04	12.64	15.01
shitter	9.88	9.46	9.23	9.32

Note: $\overline{x}(\text{ACR}_\text{a})$, $\overline{x}(\text{ACR}_\text{b})$, $\widetilde{x}(\text{ACR}_\text{a})$, and $\widetilde{x}(\text{ACR}_\text{b})$ value are given in 10^{-3} scale.

Table 4: The Wilcoxon hypotheses statement and the test results. The symbols ✓ and ✗ indicate the result of the null hypothesis test (✓ fail to reject, and ✗ reject).

Wilcoxon Hypothesis	Value of p-value
$H_0^{\overline{x}} : \overline{x}(\text{ACR}_\text{b}) = \overline{x}(\text{ACR}_\text{a})$ (✓) $H_1^{\overline{x}} : \overline{x}(\text{ACR}_\text{b}) > \overline{x}(\text{ACR}_\text{a})$ (✗) $H_2^{\overline{x}} : \overline{x}(\text{ACR}_\text{b}) < \overline{x}(\text{ACR}_\text{a})$ (✗)	0.67
$H_0^{\widetilde{x}} : \widetilde{x}(\text{ACR}_\text{b}) = \widetilde{x}(\text{ACR}_\text{a})$ (✓) $H_1^{\widetilde{x}} : \widetilde{x}(\text{ACR}_\text{b}) > \widetilde{x}(\text{ACR}_\text{a})$ (✗) $H_2^{\widetilde{x}} : \widetilde{x}(\text{ACR}_\text{b}) < \widetilde{x}(\text{ACR}_\text{a})$ (✗)	0.62

7 CONCLUSION AND FINAL REMARKS

We analyzed 20 open-source Java projects, 10 long-lived libraries, and 10 Android projects, with the intention of checking if the implementation of CI/CD had an impact on the ACR, which we use as a proxy for the presence of AoC. We filtered and chose the repositories following specific criteria to get relevant projects for the analysis. Then, we made a static analysis on all project versions to get the important data, such as LoC and Number of AoC, to calculate the ACR. We then made a comparison using the means of each project's rate before and after the implementation of CI/CD. Our results imply there's no statistically significant relationship between the implementation of CI/CD and the ACR.

REFERENCES

[1] Carlos D. A. de Almeida, Diego N. Feijó, and Lincoln S. Rocha. 2022. Studying the Impact of Continuous Delivery Adoption on Bug-Fixing Time in Apache's Open-Source Projects. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 132–136. https://doi.org/10.1145/3524842.3528049

[2] Victoria Bogachenkova, Linh Nguyen, Felipe Ebert, Alexander Serebrenik, and Fernando Castor. 2022. Evaluating Atoms of Confusion in the Context of Code Reviews. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 404–408. https://doi.org/10.1109/ICSME55016.2022.00048

[3] Fernando Castor. 2018. Identifying Confusing Code in Swift Programs. (2018).

[4] Lianping Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software* 32, 2 (2015), 50–54. https://doi.org/10.1109/MS.2015.27

[5] José Aldo Silva da Costa, Rohit Gheyi, Fernando Castor, Pablo Roberto Fernandes de Oliveira, Márcio Ribeiro, and Baldoino Fonseca. 2023. Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. *Empirical Software Engineering* 28 (05 2023). https://doi.org/10.1007/s10664-023-10311-0

[6] Jeffrey Fairbanks, Akshharaa Tharigonda, and Nasir U. Eisty. 2023. Analyzing the Effects of CI/CD on Open Source Repositories in GitHub and GitLab. arXiv:2303.16393 [cs.SE]

[7] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.

[8] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 129–139. https://doi.org/10.1145/3106237.3106264

[9] Dan Gopstein, Henry Hongwei Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 281–291.

[10] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education.

[11] Chris Langhout and Maurício Aniche. 2021. Atoms of Confusion in Java. arXiv:2103.05424 [cs.SE]

[12] Valentina Lenarduzzi, Davide Taibi, Davide Tosi, Luigi Lavazza, and Sandro Morasca. 2020. Open Source Software Evaluation, Selection, and Adoption: a Systematic Literature Review. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 437–444. https://doi.org/10.1109/SEAA51224.2020.00076

[13] Pei Liu, Xiaoyu Sun, Yanjie Zhao, Yonghui Liu, John Grundy, and Li Li. 2023. A First Look at CI/CD Adoptions in Open-Source Android Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 201, 6 pages. https://doi.org/10.1145/3551349.3561341

[14] Wendell Mendes, Oton Pinheiro, Emanuele Santos, Lincoln Rocha, and Windson Viana. 2022. Dazed and Confused: Studying the Prevalence of Atoms of Confusion in Long-Lived Java Libraries. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 106–116. https://doi.org/10.1109/ICSME55016.2022.00018

[15] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*. 25–35. https://doi.org/10.1109/ICPC.2015.12

[16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (08 2015). https://doi.org/10.1002/spe.2346

[17] Daniel Ståhl, Torvald Mårtensson, and Jan Bosch. 2017. Continuous practices and devops: beyond the buzz, what does it all mean? 440–448. https://doi.org/10.1109/SEAA.2017.8114695

[18] Adriano Torres, Caio Oliveira, Márcio Okimoto, Diego Marcilio, Pedro Queiroga, Fernando Castor, Rodrigo Bonifacio, E.D. Canedo, Márcio Ribeiro, and Eduardo Monteiro. 2023. An Investigation of confusing code patterns in JavaScript. *Journal of Systems and Software* 203 (05 2023), 111731. https://doi.org/10.1016/j.jss.2023.111731

[19] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated.

[20] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976. https://doi.org/10.1109/TSE.2017.2734091

[21] Øyvind Hauge, Claudia Ayala, and Reidar Conradi. 2010. Adoption of open source software in software-intensive organizations – A systematic literature review. *Information and Software Technology* 52, 11 (2010), 1133–1154. https://doi.org/10.1016/j.infsof.2010.05.008 Special Section on Best Papers PROMISE 2009.