TRABAJO FIN DE MÁSTER

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA.

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

E.T.S. INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

PREDICCIÓN AUTOMÁTICA DEL RESULTADO DE INTEGRACIÓN CONTINUA EN EL DESARROLLO DE SOFTWARE MODERNO.

AUTOMATIC PREDICTION OF CONTINUOUS INTEGRATION OUTCOME IN MODERN SOFTWARE DEVELOPMENT.

Realizado por
Joaquín Alejandro España Sánchez

Tutorizado por
Gabriel Jesús Luque Polo
Francisco Javier Servant Cortés

UNIVERSIDAD DE MÁLAGA

Escuela Técnica Superior de Ingeniería Informática

—

Universidad de Málaga,
Málaga, Septiembre de 2024

Índice general

Resumen – En el contexto del desarrollo de software moderno, la Integración Continua (*CI*) es una práctica ampliamente adoptada que busca automatizar el proceso de integración de cambios de código en un proyecto. A pesar de ofrecer numerosas ventajas, implementarla conlleva una serie de costos significativos que deben ser abordados para la garantizar la eficiencia a largo plazo. La fase de Integración Continua puede resultar costosa tanto en términos de recursos computacionales como económicos, llevando a grandes empresas como Google y Mozilla a invertir millones de dólares en sus sistemas de *CI* [1]. Han surgido numerosos enfoques para reducir el costo asociado a la carga computacional evitando ejecutar construcciones que se espera que sean exitosas [2]. Sin embargo, estos enfoques no son exactos, llegando a hacer predicciones erróneas que omiten ejecutar construcciones que realmente fallan. Además de los costos asociados con la carga computacional y económica de la *CI*, otro problema al que se enfrentan los equipos de desarrollo de software es el tiempo que deben esperar para obtener *feedback* del resultado del proceso de *CI* [3]. Este tiempo de espera en ocasiones puede ser significativo y puede afectar negativamente a la productividad y eficiencia del equipo, así como a la capacidad de respuesta ante problemas y ajustes rápidos en el desarrollo. Así, en este trabajo nuestro objetivo es reducir el costo computacional en CI, al mismo tiempo que maximizamos la observación de construcciones fallidas. Para ello, se ha realizado un estudio empírico sobre técnicas existentes [2,4,5,6,7,8], y se ha propuesto una implementación, *JAES24*, que busca contribuir a las mismas. Posteriormente, se han realizado una serie de experimentos para verificar y validar la efectividad de *JAES24* en comparación con otras técnicas existentes. Finalmente, se desarrollarán unas conclusiones sobre los resultados obtenidos y se propondrán posibles líneas de trabajo futuro.

Palabras clave: Integración Continua, Predicción de Builds, Costo de Mantenimiento, Aprendizaje Automático, Fallas de Builds, Predicción de Fallas, Ahorro de Costos, Predicción de Resultados de Builds, Mantenimiento de Software, Características de Builds

Abstract – In the context of modern software development, Continuous Integration (CI) is a widely adopted practice that aims to automate the process of integrating code changes in a project. Despite offering numerous advantages, implementing CI involves significant costs that need to be addressed to ensure long-term efficiency. The Continuous Integration phase can be costly in terms of computational and economic resources, leading large companies like Google and Mozilla to invest millions of dollars in their CI systems [1]. Several approaches have emerged to reduce the cost associated with computational load by avoiding running builds that are expected to be successful [2]. However, these approaches are not accurate, often making erroneous predictions that skip running builds that actually fail. In addition to the costs associated with computational and economic load of CI, another problem faced by software development teams is the time they have to wait to get feedback on the CI process outcome [3]. This waiting time can sometimes be significant and can negatively impact team productivity and efficiency, as well as the ability to respond to issues and make quick adjustments in development. Therefore, the objective of this work is to reduce the computational cost in CI while maximizing the observation of failed builds. To achieve this, an empirical study on existing techniques has been conducted [2,4,5,6,7,8], and an implementation, *JAES24*, has been proposed to contribute to these techniques. Subsequently, a series of experiments have been conducted to verify and validate the effectiveness of *JAES24* compared to other existing techniques. Finally, conclusions will be drawn on the obtained results and possible future lines of work will be proposed.

Keywords: Continuous Integration, Build Prediction, Maintenance Cost, Machine Learning, Build Failures, Failure Prediction, Cost Saving, Build Outcome Prediction, Software Maintenance, Build Features

1. Introducción

La Integración Continua (*Continuous Integration, CI*) es una práctica de desarrollo de *software* que busca automatizar el proceso de fusión de cambios de código en un proyecto, donde cada integración es verificada mediante la ejecución automática de pruebas. Este proceso busca la detección temprana de errores y mejorar la calidad del software, permitiendo una integración más frecuente y rápida del trabajo de todos los desarrolladores. Las buenas prácticas de *CI* [9] permiten una rápida detección de errores y su resolución, un *feedback* rápido, la reducción de errores que provienen de tareas manuales, unas tasas de *commits* y *pull requests* más altas, una calidad del *software* mayor, reconocer errores en producción temprano antes del despliegue, etc. Numerosos son sus ámbitos de aplicación: *software* empresarial, desarrollo de aplicaciones web, proyectos de código abierto, aplicaciones móviles, etc. Todo ello, haciendo uso de las distintas herramientas que existen en el mercado [10], como *GitHub Actions*, *Jenkins*, *Travis CI*, *CircleCI*, *Azure DevOps*, entre otras.

El ciclo de vida de la Integración Continua, a pesar de ofrecer numerosas ventajas, conlleva grandes costos asociados debido a los recursos computacionales [11] necesarios para ejecutar las construcciones, comúnmente denominadas *builds*. A lo largo de este trabajo, nos referiremos como costo computacional al hecho de ejecutar una *build*, es decir, el proceso de construir el *software* y ejecutar todas las pruebas cuando la *CI* es lanzada. Este costo asociado se acentúa en empresas de gran tamaño, donde el número de *builds* que se ejecutan diariamente es muy elevado [12,13]. Además, hay que sumar la larga duración que pueden tene la ejecución de las *builds* en este tipo de empresas.

En los últimos años, han surgido numerosos enfoques centrados en reducir el costo computacional asociado a la ejecución de *CI* [2,4,5,6,7,8]. La idea principal de estos enfoques es reducir el número de *builds* que se ejecutan, prediciendo el resultado antes de su ejecución y, por lo tanto ahorrándose ese costo computacional. Las *builds* predichas como construcciones exitosas (*build pass*) no se ejecutan, mientras que las predichas como construcciones fallidas (*build failure*) sí se ejecutan. De esta forma, se mantiene el valor conceptual de la *CI*, que es la detección temprana de errores, pero reduciendo el costo computacional asociado en el proceso. Este estudio toma como punto de partida el algoritmo de *machine learning SmartBuildSkip* [2]. La idea principal es realizar una contribución a este algoritmo, usando features más significativas para predecir, o bien realizando una variante propia del mismo que mejore los resultados obtenidos. Por lo tanto, este estudio se enmarca en el desarrollo de software moderno, específicamente en el ámbito de la Integración Continua y la predicción automática del resultado de dicha integración.

La memoria queda organizada de la siguiente forma: en primer lugar, se realiza un estudio del estado del arte que sitúa los antecedentes previos a la Integración Continua y la predicción automática de resultados de *builds*. Posteriormente, se establecen los objetivos y preguntas de investigación que pretende este estudio responder. A continuación, se describe en detalle el problema a resolver, los principales obstáculos que se plantean y sus posibles soluciones. Acto seguido, se desarrolla con detalle nuestra propuesta al problema, describiendo las tecnologías usadas y el desarrollo de la solución. Después se presentarán las pruebas y resultados obtenidos, comparando la solución con otras existentes, a modo de validar y verificar la aportación de nuestra solución. Seguidamente, se comentarán las amenazas a la validez, una parte esencial en cualquier trabajo de investigación. Este apartado nos permite identificar y discutir posibles limitaciones que podrían afectar a la validez de los resultados y a las conclusiones. Por último, se darán unas conclusiones sobre los resultados obtenidos y se propondrán posibles líneas de trabajo futuro.

2. Antecedentes y trabajos relacionados

2.1. Antecedentes

Este trabajo se centra en la implementación, evaluación y mejora del algoritmo de predicción de *CI* propuesto en [2]. Para comprender mejor el contexto en el que se desarrolla, primero vamos a presentar

algunos de los conceptos básicos de *CI* y del problema que nos ocupa. En primer lugar, se describirá el ciclo de vida de *CI*, junto a las dos casuísticas que pueden darse en el proceso de integración. En segundo lugar, hablaremos sobre la extracción de características, un aspecto fundamental para algoritmos de predicción. Por último, se hablará del consumo de recursos computacionales que supone la implementación de *CI*, de ahí la principal motivación de este trabajo, la reducción de dichos costes.

2.1.1. El ciclo de vida de la Integración Continua. La Integración Continua es un proceso iterativo en el cual varios contribuidores hacen cambios sobre un mismo código base añadiendo nuevas funcionalidades, para luego integrarlas a la misma linea temporal de desarrollo, de forma controlada y automatizada. Cada integración se realiza a través de la compilación, construcción y ejecución de pruebas automatizadas sobre el código fuente [14]. Aunque pueda parecerlo, la Integración Continua no es un proceso trivial, en [15] se describen las buenas prácticas de *CI* que deben seguirse para garantizar la calidad del software, algunas de las cuales han sido fuertemente adoptadas en el sector, como son el punto de código fuente único, la automatización de *builds* y el desarrollo de pruebas unitarias o de validación interna. Sin embargo, en el mundo real, la forma de aplicar cada una de estas técnicas y la prioridad con la que se aplican puede estar fuertemente influenciada [9] por la cultura empresarial donde se desarrollen.

Ejemplo práctico: Un desarrollador hace un *commit* (una instantánea de los cambios realizados) y mediante una acción de *push*, lo envía al repositorio central. El servidor de *CI* [10] (Jenkins, Travis CI, GitHub Actions, etc.) detecta automáticamente este nuevo *commit* y desencadena el *pipeline* de *CI*. El servidor extrae el nuevo código del repositorio y comienza a construir la aplicación, lo que denominados la fase de construcción o *build*. Esta parte puede incluir la compilación del código fuente, la instalación de dependencias, etc. Una vez que la aplicación está construida, se ejecutan una serie de pruebas automatizadas (*Self-Testing code*) [11]. Dichas pruebas pueden ser pruebas unitarias, pruebas de integración, pruebas funcionales o pruebas de interfaz de usuario. Dependiendo del resultado de las fases anteriores, podemos encontrarnos dos casos:

- **La *build* ha sido exitosa**: todas las pruebas han pasado con éxito. En este caso, el servidor de *CI* puede desplegar la aplicación en un entorno de pruebas o producción.
- **La *build* ha fallado**: alguna de las pruebas ha fallado. En este caso, el servidor de *CI* suele notificar a los desarrolladores y detiene el despliegue de la aplicación.

2.1.2. Características de las *builds*. Al ejecutarse una *build*, se pueden extraer de ella una serie de características con las que algoritmos de predicción pueden predecir el resultado de la integración. Tener un conjunto de *features* bien seleccionadas y significativas mejorará la precisión de los modelos. La mayoría de estudios utilizan catacteríticas extraídas directamente de la base de datos de TravisTorrent [7], sin embargo, estas características no son las mejores para predecir *builds* que fallan, es decir, *builds failures*. Algunos enfoques [7,8] hacen uso de características basadas en la *build* actual, la *build* anterior y el histórico ligado a todas las ejecuciones de *builds* anteriores. Sin embargo, esto supone una limitación para la detección de los primeros *failures* [2], ya que estos dependen mucho del resultado de la *build* anterior y, por definición, estarán siempre precedidos por una *build* exitosa.

Los *build failures* pueden categorizarse en una serie de tipos. *Rausch et al.* [16] muestra una categorización de las *build failures* según el tipo de error que las origina, identificándose un total de 14 categorías. En el estudio se demostró que más del 80 % de los errores se producían en la fase de ejecución de pruebas o *tests*. Todo ello, teniendo en cuenta el *dataset* de estudio, 14 proyectos de código abierto basados en Java que usan Travis CI. En [19] se realiza un estudio a gran escala con 3.6 millones de *builds* en el que se demuestra que factores como la cantidad de cambios en el código fuente, el número de *commits*, el número de archivos modificados o la herramienta de integración usada tienen una relación estadísticamente significativa con las compilaciones fallidas.

2.1.3. El costo de la Integración Continua. La implementación de la Integración Continua, a pesar de ofrecer numerosas ventajas, también supone un coste computacional y económico. Además del costo computacional que supone ejecutar la *CI*, debemos de sumarle el costo que supone el tiempo no productivo de los desarrolladores si estos no saben como proceder sin saber el resultado de la integración. Hilton et al. [2] estudiaron los beneficios y costes de aplicar *CI* en proyectos de código abierto. Entre el costo de aplicar *CI*, encontraron que: algunos proyectos no adoptan *CI* porque sus desarrolladores no están familiarizados con ella, que es frecuente que se realicen cambios en la configuración de *CI* y que estos sean debidos a versiones obsoletas en las dependencias y, por último, que el tiempo asociado a *builds* exitosas es menor que el tiempo asociado a *builds* fallidas. Klotins et al. [21] realizaron un trabajo con múltiples casos de estudio en el que se encontró que la aplicación de *CI* mejoraba notablemente los procesos de desarrollo internos en las empresas, sin embargo, se destaca la necesidad de evaluar las consecuencias de aplicar este tipo de desarrollo desde una perspectiva del cliente en la adopción de entregas continuas. En [20] se presenta un estudio empírico en el que se resalta que los desarrolladores vieron los *build failures* como difíciles de resolver, pensando que podrían retrasar el desarrollo de *software* y reducir la productividad del equipo. Además, se menciona que factores técnicos como humanos desempeñan un papel fundamental en la adopción de *CI*.

2.2. Trabajos relacionados

Existen numerosos estudios que buscan reducir el costo asociado a la Integración Continua mediante la creación de *predictors* [8,5,2,18,7,17,4,1,22]. Hassan et al. [8] estudiaron un total de 402 proyectos Java con información de 256,055 *builds*, procedentes de la base de datos de TravisTorrent. En su estudio se utilizan características extraídas directamente de la base de datos de TravisTorrent y otras propias, relacionando *features* propias de la *build actual* y de la anterior. En su propuesta se utiliza un predictor que usa *K-Mean Clustering* para clasificar las *builds* en exitosas y fallidas. En [5] se propone una solución novedosa que usa Programación Genética Multi-Objetivo, sin utilizar técnicas de aprendizaje automático. En su estudio encontraron que características como el tamaño del equipo, la información de la última *build* o el tipo de archivos cambiados, pueden indicar el potencial fallo de una *build*.

Servant et al. [2] propone un algoritmo que utiliza técnicas de *Machine Learning* para la predicción de *CI*. Parten de dos hipótesis principales: la primera, que las *builds* exitosas son mas numerosas que las fallidas y, la segunda, que muchas *builds* fallidas se producen consecutivamente. En él se proponen una serie de *fatures* tanto para la predicción sobre el mismo proyecto como para la predicción entre proyectos, también llamada *cross-project prediction*. Además, se propone un algoritmo que usa *Random Forest* como clasificador. Nuestro estudio se centra en la implementación, evaluación y mejora de este algoritmo.

Saidani et al. [18] propone un predictor que utiliza Redes Neuronales Recurrentes (*RNN*) basadas en Memoria a Largo Plazo (*LSTM*). Sus estudios revelan que este tipo de técnicas ofrecen mejores resultados que las que usan técnicas de *Machine Learning*, obteniendo mejor rendimiento en términos de *AUC*, *F1-Score* y *accuracy* cuando se trata de validación entre proyectos. En [7] se propone una nueva solución en la que se usa un predictor que es dependiente del histórico de *builds* pasadas para poder hacer sus predicciones. En este estudio existen métodos de selección de *features* que selecionan determinadas *features* en función del tipo de proyecto que se esté evaluando. En otro artículo, Ouni et al. [17] propone una solución de línea de comandos donde se consigue mejorar el estado del arte en términos de *F1-Score*, sin embargo, solo se tiene en cuenta el estudio [8] comentado anteriormente. Jin et al. [4] propone un nuevo predictor que mejora el ahorro de costo y la observación de *builds* fallidas, mejorando el estado del arte. Finalmente, Jin et al. [6] propone una solución que emplea técnicas de selección de *builds* y dos técnicas de selección de tests. Esta solución ejecuta seis técnicas existentes y luego usa los resultados como *features* para un clasificador *Random Forest*.

Por otro lado, no existen proyectos documentados que usen técnicas de predicción de*CI* para el ahorro de costos, por lo que es complicado evaluar el impacto económico real que estas pueden causar. Liu et al.

[22] utiliza simulación de procesos *software* y experimentos basados en simulación para evaluar el impacto de estos *predictors* de *CI* en un entorno más realista. Entre sus descubrimientos, vieron que existe poca diferencia entre los *predictors* del esado del arte y las estrategias aleatorias en términos de ahorro de tiempo. Sin embargo, en casos donde el ratio de *builds* fallidas es mayor, la estrategia aleatoria tendría un impacto negativo. Además, en proyectos donde la proporción de *failures* es muy pequeña, el uso de *CI* predictiva no es mucho mejor que saltar *builds* de forma aleatoria. A pesar de esto, se demuestra que el uso de técnicas de *predictive CI* puede ayudar a ahorrar el costo de tiempo para ejecutar *CI*, así como el tiempo promedio de espera antes de ejecutar la *CI*.

3. Objetivos y preguntas de investigación

En un estudio de carácter exploratorio como el que se propone, definir unos objetivos y preguntas de investigación se convierte en una tarea fundamental para la correcta orientación del trabajo. En este sentido, los objetivos nos permiten establecer una serie de metas a alcanzar, mientras que las preguntas de investigación nos ayudan a centrar el estudio en aspectos concretos que queremos responder. Los objetivos de la investigación son los siguientes:

- **OB-1**: implementar un algoritmo de aprendizaje automático que genere un modelo predictivo (un *predictor*) basado en un conjunto de características *features* extraídas de las *builds*.
- **OB-2**: utilizar la *API* de GitHub para obtener datos relevantes sobre las *builds*, como su histórico, características asociadas, resultados anteriores de la integración continua.
- **OB-3**: desarrollar e implementar diferentes algoritmos de predicción con la selección de diferentes características con el objetivo de proporcionar múltiples opciones a la hora de predecir el resultado de la integración continua.
- **OB-4**: implementar una interfaz gráfica que sirva como punto de entrada de datos para el algoritmo de predicción y que permita visualizar los resultados obtenidos.

Las preguntas de investigación tienen el objetivo de estructurr y orientar el proceso de investigación. A continuación se establecen las preguntas de investigación junto a las métricas usadas para su evaluación:

- **PI-1**: ¿Qué algoritmo de predicción produce los mejores resultados en la predicción automática del resultado de la integración continua?
  - **Métrica**: *accuracy*, *precision*, *recall* y *F1-score* del modelo.
- **PI-2**: ¿Qué características de las *builds* son más significativas en la predicción?
  - **Métrica**: importancia de cada feature a través de la interpretación de los coeficientes del modelo.

4. Descripción del problema

5. Detalles de la propuesta

6. Resultados

7. Amenazas a la validez

8. Conclusiones y trabajos futuros

Referencias

1. Xianhao Jin and Francisco Servant. 2023. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. ACM Trans. Softw. Eng. Methodol. 32, 4, Article 93 (July 2023), 39 pages. https://doi.org/10.1145/3576038
2. Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1145/3377811.3380437
3. Xianhao Jin and Francisco Servant. 2021. CIBench: a dataset and collection of techniques for build and test selection and prioritization in continuous integration. In Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings (ICSE '21). IEEE Press, 166–167. https://doi.org/10.1109/ICSE-Companion52605.2021.00070
4. Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. J. Syst. Softw. 188, C (Jun 2022). https://doi.org/10.1016/j.jss.2022.111292
5. Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting continuous integration build failures using evolutionary search. Inf. Softw. Technol. 128, C (Dec 2020). https://doi.org/10.1016/j.infsof.2020.106392
6. Xianhao Jin and Francisco Servant. 2023. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. ACM Trans. Softw. Eng. Methodol. 32, 4, Article 93 (July 2023), 39 pages. https://doi.org/10.1145/3576038
7. Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2021. BuildFast: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20). Association for Computing Machinery, New York, NY, USA, 42–53. https://doi.org/10.1145/3324884.3416616
8. Foyzul Hassan and Xiaoyin Wang. 2017. Change-aware build prediction model for stall avoidance in continuous integration. In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '17). IEEE Press, 157–162. https://doi.org/10.1109/ESEM.2017.23
9. O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst and M. -A. Storey, Üncovering the Benefits and Challenges of Continuous Integration Practices,ïn IEEE Transactions on Software Engineering, vol. 48, no. 7, pp. 2570-2583, 1 July 2022, doi: 10.1109/TSE.2021.3064953.
10. Rostami Mazrae P., Mens T., Golzadeh M., Decan A. On the usage, co-usage and migration of CI/CD tools: A qualitative analysis (2023) Empirical Software Engineering, 28 (2), art. no. 52, Cited 15 times. DOI: 10.1007/s10664-022-10285-5
11. M. Hilton, T. Tunnell, K. Huang, D. Marinov and D. Dig, Üsage, costs, and benefits of continuous integration in open-source projects,"2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 2016, pp. 426-437.
12. Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16). Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358
13. Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15). IEEE Press, 483–493.
14. Hilton, Michael; Tunnell, Timothy; Huang, Kai ; Marinov, Darko ; Dig, Danny. Usage, costs, and benefits of continuous integration in open-source projects (2016) ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 426 - 437 DOI: 10.1145/2970276.2970358
15. M. Fowler and M. Foemmel, "Continuous integration," https://tinyurl.com/ycbl2uhj, 2006, [Online; accessed 2-Aug-2023].
16. Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17). IEEE Press, 345–355. https://doi.org/10.1109/MSR.2017.54
17. Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2021. BF-detector: an automated tool for CI build failure detection. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021). Association for Computing Machinery, New York, NY, USA, 1530–1534. https://doi.org/10.1145/3468264.3473115

18. Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the prediction of continuous integration build failures using deep learning. Automated Software Engg. 29, 1 (May 2022). https://doi.org/10.1007/s10515-021-00319-5
19. Md Rakibul Islam and Minhaz F. Zibran. 2017. Insights into continuous integration build failures. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17). IEEE Press, 467–470. https://doi.org/10.1109/MSR.2017.30
20. Hong, Yang; Tantithamthavorn, Chakkrit; Pasuksmit, Jirat; Thongtanunam, Patanamon; Friedman, Arik; Zhao, Xing; Krasikov, Anton; Practitioners' Challenges and Perceptions of CI Build Failure Predictions at Atlassian (2024), pp. 370 - 381. DOI: 10.1145/3663529.3663856
21. Eriks Klotins, Tony Gorschek, Katarina Sundelin, and Erik Falk. 2022. Towards cost-benefit evaluation for continuous software engineering activities. Empirical Softw. Engg. 27, 6 (Nov 2022). https://doi.org/10.1007/s10664-022-10191-w
22. B. Liu, et al.,"The Why, When, What, and How About Predictive Continuous Integration: A Simulation-Based Investigationïn IEEE Transactions on Software Engineering, vol. 49, no. 12, pp. 5223-5249, 2023. doi: 10.1109/TSE.2023.3330510