A Quantitative Study of C/C++ FOSS Software Buildability

Dalibor Fonović, Tihana Galinac Grbac*

Juraj Dobrila University of Pula, Zagrebačka 30, Pula, HR-52100, Croatia

Abstract

Software build and integration procedures are responsible to transform developers' files into machine executable codes. This process separates the software development project from the software deployment phase and it represents a critical obstacle for software performance on the software developer side and deployment effectiveness from the software execution side. Although the automation of this process is an active research topic for almost half a century, the standardized procedures that would enable full automation across various software projects, but also across time, are still missing. The main reason lies in the variety of methods and tools used in software development and their evolution. Therefore, many researchers and practitioners report a low build success rate. Researchers have investigated this problem from software development, integrator, and software evolution perspectives. In this paper, a study is set up in order to contribute to the overall understanding of the level of standardization among FOSS software repositories and automation abilities. A rich source of information lies in FOSS repositories and mining these repositories with joined developers and deployment perspective would bring valuable knowledge into the software engineering profession.

Keywords

Build system, CMake, Make, Error logs, build time, empirical analysis

1. Introduction

Automation of software build and integration procedures is a long-lasting research topic especially attractive during the last decade [1]. The main role of software build and integration is to compile code, fetch and link various pieces of code and finally integrate all these files into a build executable unit. This process has an important role in the performances of software developers and software developer companies but also in run-time platforms where software is continuously integrated and installed on various platforms to serve numerous end users. In both cases, unsuccessful builds significantly impact waste costs. From the software developer's perspective, unsuccessful builds impact the developer's productivity and effectiveness to complete the tasks in a given time with reasonable quality levels. From the run-time hosting

SQAMIA 2022: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 11–14, 2022, Novi Sad, Serbia

*Corresponding author.

†These authors contributed equally.

dalibor.fonovic@unipu.hr (D. Fonović); tihana.galinac@unipu.hr (T. Galinac Grbac)

https://www.unipu.hr/dalibor.fonovic (D. Fonović); https://www.unipu.hr/en/tihana.galinac_grbac (T. Galinac Grbac)

0000-0002-4351-4082 (T. Galinac Grbac)

platform perspective it impacts the time delays, energy lost and unsuccessful service delivery to the end user. The further evolution of the software engineering profession follows the open network paradigm, in which software code development continuously interacts with software use. In such conditions, the automation of build procedures becomes a necessity to survive and not a matter of maturity of the process. Continuous integration and continuous delivery (CI/CD), is an example of good practice, frequently used in large-scale complex projects and industrial environments, which delivers its services to numerous users via Cloud infrastructure, enables rapid software changes and improves system stability.

Although the first automation ideas with the development of tools like for example Make build system are almost 50 years old [1], there are still many difficulties to build and integrate software projects [2, 3]. This is especially the case in open source software versioning repositories, in which there exists a huge variety of software development methods and tools used and a lack of standard procedures.

In this paper, a deep investigation of FOSS (free, open-source software) projects, stored in the GITHUB repository and openly available, is undertaken. The focus is on C/C++ programming language projects. The CMake and Make building tools are used. CMake is not a build system tool, but it is used as a generator of building scripts for build systems on various platforms and Make tool drives compiler and other build tools and uses the result of CMake to build system. In this study, the aim is to provide empirical evidence and to discuss the effectiveness of automation for FOSS software projects. The point is to stress how the lack of standards in build procedures impacts the abilities for automation. Although the derived knowledge from interconnecting these two worlds, developers and deployment perspectives, would provide valuable information that can be used to better develop software products.

The paper is organized as follows. After the Introduction in Sect. 1, the overview of related work is provided in Sect. 2. In Sect. 3 the research questions are defined and the methodology followed to undertake the experiment is explained. In sect. 4, the obtained results are discussed, along with the future work. Finally, Sect. 5 concludes the paper.

2. Related work

The automated build is a widely adopted best practice in industry, which is reported in numerous research and professional papers. However, these papers report a significant percentage of failed automated builds. The biggest problems from the integrator site are found to be troubleshooting build failures and overly long builds [4]. Previous studies have reported that almost 90% of error types encountered during build procedures are caused by build failures and that dependencies are the most common source of failure [5, 6].

The quantitative analysis of main factors of build break in industry environment [6] has shown that there is a correlation between the number of simultaneous contributors on branch and type of work performed and roles of contributors. Coordinated action among contributors on a branch to improve build performance is proposed.

Recent works have reported on the development of automation tools that improve build tools. Major big software and service integrators have invested in the development of such tools. Microsoft developed a CloudBuild system that helps to improve build times up to 10x and improve service availability on 99% [7]. At the same time, Google works on Bazel build and test tool for targeting optimized performances in relation to scalable build demands for multi-language and multi-platform projects. Facebook Buck build system [8] is also targeting build speed, procedures that would improve build correctness, and a better understanding of dependencies. There are also numerous other works on the same topic [9, 10, 11].

Although there is a growing body of developments in new tools, the traditional tools, that are *de facto* standards in the software development community, like CMake, are still very much used in large-scale projects, especially if there is a need to integrate with the software of other companies [12]. There are still numerous challenges that have to be addressed to move this tool into a faster and more optimized for large-scale scalable projects.

3. Study

In this section, the research questions and hypotheses are defined, and the data collection methods and tools are described.

3.1. Research questions

The goal of this study is to replicate the study in [5, 2]. The original study was performed within the industry (Google) environment investigating the programmer's perspective within the continuous delivery process. The quantitative study in the other paper reports on automatically Java build target archives from the FOSS projects. The reproducibility of the results from these two previous studies for projects written in C/C++ programming languages and stored in the FOSS environment is investigated here.

RQ1: How often builds fail? The project build success/failure while automating the build process in GITHUB projects is reported.

RQ2: Why do builds fail? It is verified how many software projects from the GITHUB repository suffer from the same problems in comparison to industrial projects reported in [2, 3].

3.2. Build procedure

In this section, the methods and tools used to build the projects from GITHUB are described. The data collection process is automated by using Python scripts and GITHUB APIs. The whole process is presented in Figure 1. It consists of three steps: obtaining the list of links to the software project repositories in GITHUB, cloning the repositories from the obtained links at GITHUB to the local repositories, and the build process with CMake and Make tools. During the whole process, the local database is created for data extraction. We collected the following metrics:

- Build result (successful or failed)
- Error code
- Project name
- Project path in GITHUB
- Build execution time

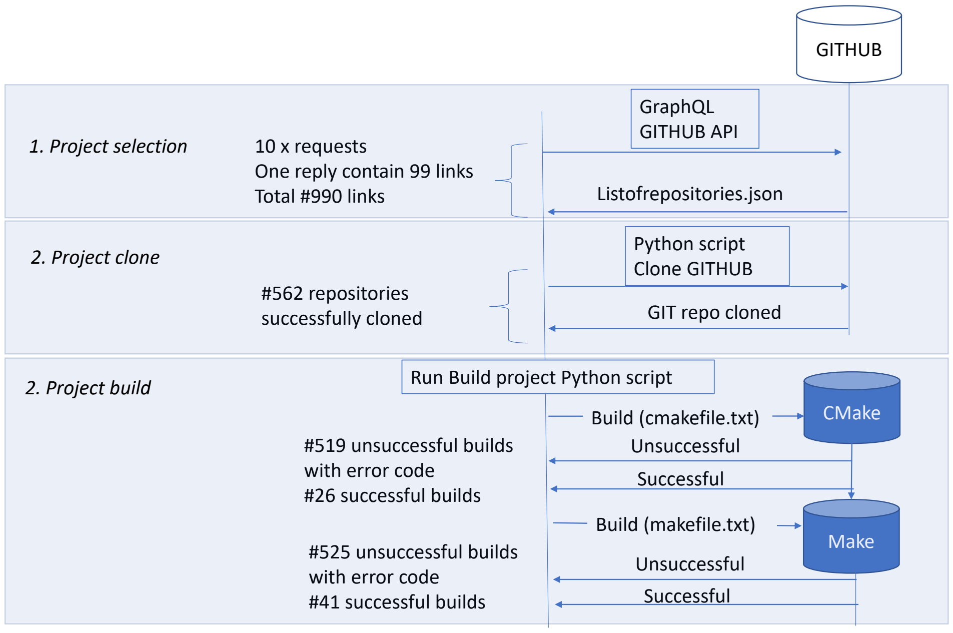

Figure 1: High level map of automated build procedure.

- Project creation date

3.2.1. Project Selection

Firstly, the Postman application with GraphQL API [1] is used for searching the GITHUB repositories for C/C++ languages. The parameters used in the query were as follows:

```
{
  "queryString": "is:public  srchived:false language:c++",
  "refOrrder":{
  "direction": "DESC",
  "field": "TAG_COMMIT_DATE"
  }
}
```

Since this API requires a GITHUB account, our personal account is used. The GITHUB personal account has a limit for using the search API in terms that it lists only 99 results per search query. Therefore, the same query was run several times. As a result of GITHUB search API, a file of 990 links to the project repositories in GITHUB is created. The time span of the project creation time was from January 2008 to June 2022. In Figure 2, the number of projects that resulted from the search function is presented. The number of projects is depicted in the

[1]http://docs.github.com/en/graphql

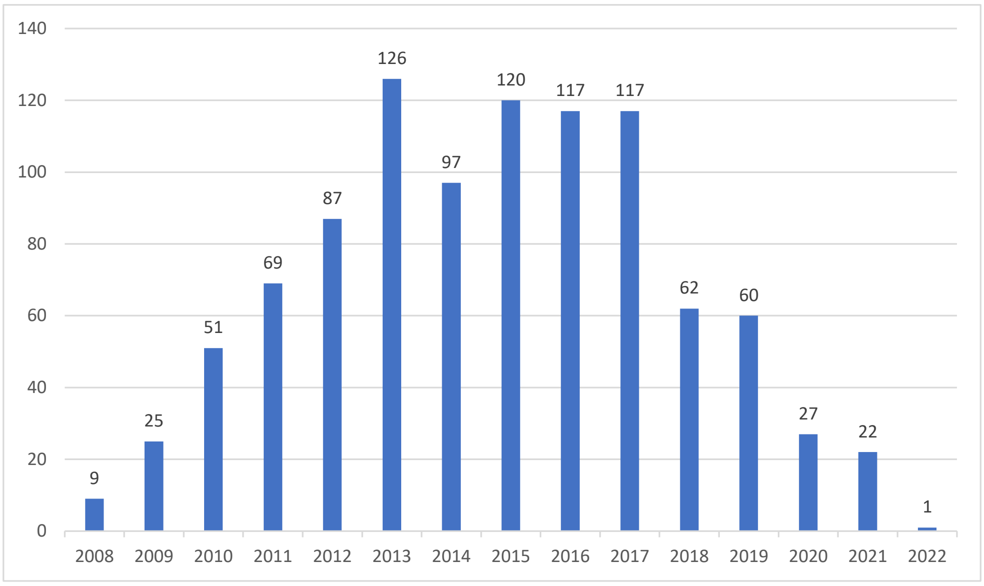

Figure 2: Histogram of the number of considered projects per creation year.

time-span of 15 years, from 2008 until 2022. From the figure, it could be concluded that the majority of projects are dated since 2013 until 2017.

This kind of search lists all the software repositories in which the developer included tag c or cpp into his GITHUB repository. Therefore, the software repository list also contains links to projects in which there is just a minor part of C or C++ projects. Some repositories are related to C# programming language and do not contain any C or C++ line of code.

3.2.2. Cloning software repositories

The second part of the Python script is developed for automation of cloning and fetching GITHUB directories. A total of 562 software projects are successfully cloned and that process took 3.31 hours. The cloning process was stopped because of limited disc space at our local repository.

3.2.3. Build process

In the third part of the automation script, after the cloning process, a Python script for a build software project is created. As the first step, a build folder is created, and from this build folder, the CMake command is called (call a Cross Platform Make application). The results of a CMake build are stored in a database, so that the individual elements can be linked and accessed. If the CMake command was executed successfully the Python script stored the results in the $\text{CMake}_{\text{log}}$ output file. The build successfully file has stored also the time needed for the build process. Otherwise, if CMake did not pass successfully, the error code message for that project is printed

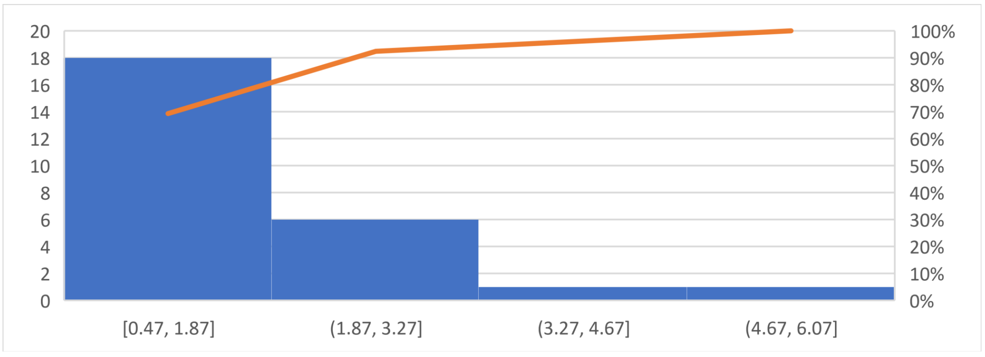

Figure 3: Build duration for CMake.

in the $\text{CMake}_{\text{error}}$ output file. Furthermore, the Make tool is used to build the projects and call the Make command for each created project repository. The results of a Make build are stored in the same database as above, so that the individual elements can be linked and accessed. If the Make command was executed successfully the Python script stored the results into the Make_{log} output file. Otherwise, if Make did not pass successfully the error code message for that project is printed in the $\text{Make}_{\text{error}}$ output file.

4. Results

The result of the build automation was that only 562 out of 990 projects have been processed. Only 26 of these 562 projects passed CMake build successfully, which is less than 5% and only 39 projects passed Make build successfully which is less than 7%. The average build duration was 1.65 seconds for CMake and 11.45 seconds for Make tool. The histograms of build times are presented in Fig. 3 and Fig. 4. In those figures, the horizontal axis represents the build time, and the vertical axis the number of projects and the percentage of projects in the given class. More than 90% of the projects were built within less than 2 seconds in CMake and less than 24 seconds in Make, whereas less than 5% of the projects need 5 to 6 seconds for the build in CMake and 1 to 2 minutes in Make.

The error codes for CMake unsuccessful builds were collected systematically within an Excel table. The reasons of unsuccessful build are categorized into the following categories:

- no cpp app or no CMake configured
- CMake configuration or version error
- dependency
- computer architecture not supported
- other

Error category: no cpp app or no CMake configured This error categorizes the automation script can not find CMake file. The reason for this may be that the project is not C++ project or

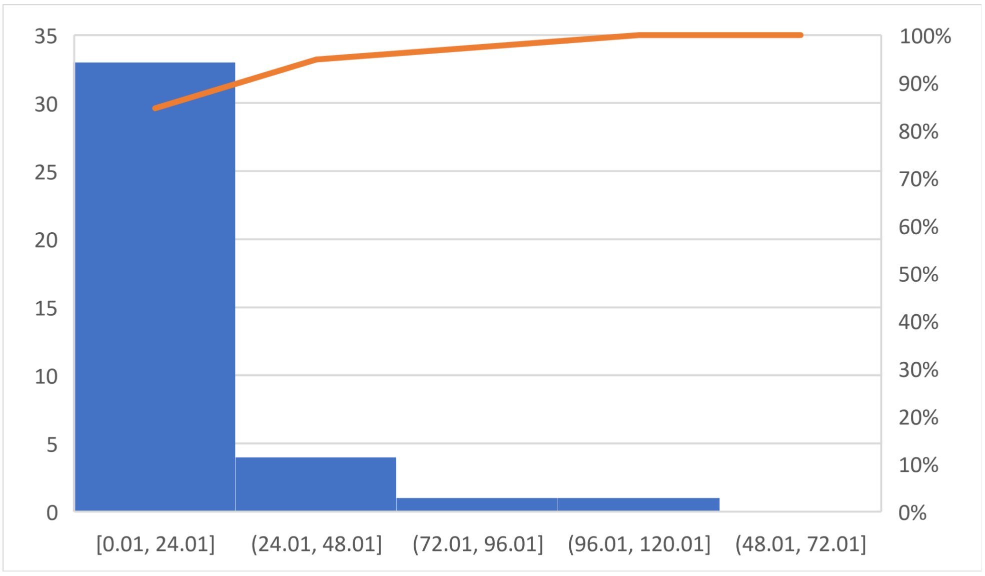

Figure 4: Build duration for Make.

the developer did not prepare across system file for build or the developer used some other tool for cross-build (e.g. Ant, Maven, Scons, QMak, Basel, Conan, B2).

Error category: CMake configuration or version error This error code is generated due to CMake policy rules. For example, older applications require a lower level of CMake tool. Hence, the main problem with old applications is that the build procedure reports an error with a version error. It may also happen that some CMake instructions may be not compatible with instructions from older CMake versions.

Error category: Dependency This error is mostly reported with the message “A required package was not found.” This means that there exists a CMake file and for a successful build, some additional packages should be installed to build the project successfully. This is because the developer did not fully automate the build procedure while preparing the CMake file. In other words, a developer could prepare a fully automated script, so that the installation of dependencies happens automatically at the project build. However, this option was not possible in some earlier versions of CMake, and even in versions that contain this option, this is only an optional procedure and it is left to the developer to decide the level of automation for the project. Note that some dependencies might be critical in terms of reliability, size, or some other parameter and the developer may decide to leave this installation decision to the project builder - if this is the case of an open repository.

In some older CMake projects, this automation was not possible (e.g. *Fetchcontent* module is available since version CMake 3.11).

Error category: computer architecture not supported The software developer did not develop an application for Unix and the build failed because of the platform.

Error category: other Sometimes the build environment asks for parameters for the building

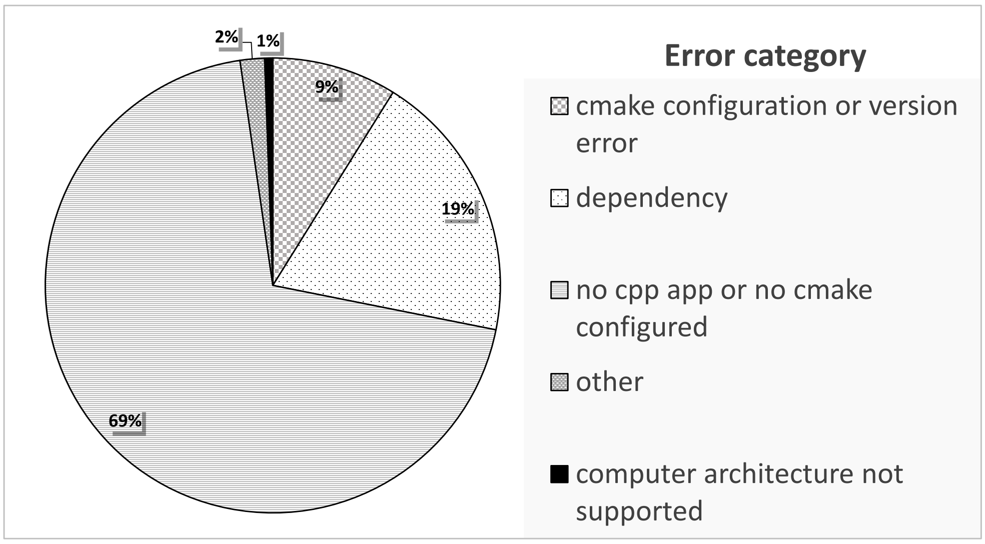

Figure 5: Percentage of error categories in unsuccessful builds.

procedure. For example, these may be specifications of the deployment platform (e.g. some specific platform, such as Arm platform) as part of the building procedure. This is when developers describe in the documentation how to build the project and in this specification, it is defined how to set up the parameters. This is why the error code other may appear.

The pie chart of obtained error categories is presented in Fig. 5. The 69% of unsuccessful builds occurred because the automation script did not find a CMake file. This means that the developer did not prepare the software project for cross platform build or it prepared the project for cross platform build but using some other tool (not CMake).

Analyzing the error messages from the Make process we identified that 98% of unsuccessful builds are ended with the error code "No targets specified and no makefile found." The remaining 2% failed because of compilation warnings and linking issues.

The obtained results indicate that the open source community is still using a vast variety of methods and tools and there is a lack of overall standards in build and integration procedures. The 69% software projects are unsuccessfully built because lack of automation script. Although CMake is reported as the preferred and dominant automation tool for C++ software projects it seems that the open source community is still rarely using it.

Furthermore, more than 50% of unsuccessful builds that had a CMake file happen because of missing dependencies. Similar results are already reported in [2, 3].

5. Conclusion

Automation of build and integration procedures across projects is the best practice from industry. There are numerous benefits of automation of these procedures, such as developer performance and deployment effectiveness. Although this good practice is well adopted within industry, there is still a lack of automation in the open source community. From the analysis, it is identified that 69% of projects had no CMake file and 98% of projects had no makefile. Therefore, building and integrating projects from open source repositories sometimes may represent a challenge for less experienced developers and limit software reuse.

Furthermore, the open source projects that had CMake or make file seem to suffer from similar building problems as industrial builds and the most dominant is dependency issue. It is found that the majority of unsuccessful builds happened because of some missing or wrong dependency file.

In this project, a simple automation script is run, and future work should focus on how to improve the script and enable a better build success rate. Moreover, with this simple script, it is not only that the build success rate is lower, but also the projects that were successfully built were mostly simple applications and libraries. It resulted in build failure for complex projects. However, this is still an open and active research problem, and leading integrator companies like Facebook, Microsoft, and Google are continuously working on improving the build and integration tools. Since CMake is *de facto* standard for building cross project applications, they also report on the use of this tool and challenges that need to be addressed for future work.

This paper explores the possible obstacles while automatizing building and integration procedures of software projects stored in open source repositories. The final goal is to develop an effective automation script for software projects that are stored in open source repositories and thus enable non-expert developers to better reuse and deploy software projects more effectively. Furthermore, the aim is to collect a dataset that would enable knowledge mining and connecting software files with executable files that would enable the study of these relations and better understanding and management of the software development process targeted to reliable, energy-efficient, and safe solutions.

This paper explores randomly selected C/C++ FOSS software projects and analyzes automation buildability. However, a significant portion of selected projects did not build. A majority of these projects were also not created for CI/CD purposes, to be automatically built via generated build script, without a human intervention to setup the build procedure. The point here is to increase the developers' awareness on the importance of buildability issues, to improve software reuse, which is actually the main idea of FOSS software.

6. Acknowledgments

This paper acknowledges the support of the Croatian Science Foundation research project CSF no. HRZZ-IP-2019-04-4216 “Reliability and Safety in Complex Software Systems: From Empirical Principles towards Theoretical Models in View of Industrial Applications (RELYSOFT)” and the support of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No. 2020–1–PT01–KA203–078646: “SusTrainable - Promoting Sustainability as a Fundamental

Driver in Software Development Training and Education".

References

[1] S. I. Feldman, Make — a program for maintaining computer programs, Software: Practice and Experience 9 (1979) 255–265. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402. doi:https://doi.org/10.1002/spe.4380090402.

[2] M. Sulír, J. Porubän, A quantitative study of java software buildability, in: Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 17–25. URL: https://doi.org/10.1145/3001878.3001882. doi:10.1145/3001878.3001882.

[3] N. Kerzazi, F. Khomh, B. Adams, Why do automated builds break? an empirical study, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 41–50. doi:10.1109/ICSME.2014.26.

[4] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, D. Dig, Trade-offs in continuous integration: Assurance, security, and flexibility, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 197–207. URL: https://doi.org/10.1145/3106237.3106270. doi:10.1145/3106237.3106270.

[5] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, R. Bowdidge, Programmers' build errors: A case study (at google), in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 724–734. URL: https://doi.org/10.1145/2568225.2568255. doi:10.1145/2568225.2568255.

[6] N. Kerzazi, F. Khomh, B. Adams, Why do automated builds break? an empirical study, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 41–50. doi:10.1109/ICSME.2014.26.

[7] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, S. Kandula, Cloudbuild: Microsoft's distributed and caching build service, in: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), 2016, pp. 11–20.

[8] Buck build, Facebook buck, 2022. URL: https://buck.build.

[9] conan, Jfrog conan center, 2022. URL: https://conan.io/center/b2.

[10] Fastbuild, Fastbuild, 2022. URL: http://fastbuild.org.

[11] C. Lebeuf, E. Voyloshnikova, K. Herzig, M.-A. Storey, Understanding, debugging, and optimizing distributed software builds: A design study, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 496–507. doi:10.1109/ICSME.2018.00060.

[12] K. Nguyen, T. Nguyen, Q.-S. Phan, Analyzing the cmake build system, in: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2022, pp. 27–28. doi:10.1109/ICSE-SEIP55303.2022.9793901.