Continuous Source Code Rejuvenation in Continuous Integration Pipelines

Endre Fülöp, Norbert Pataki* and Ábel Szauter

Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

Abstract

Most programming languages have a rather long history, they evolve time by time. However, code written in older version of a programming language is not updated automatically when a new standard is released. Moreover, the existing code may have different meaning according to the new standard in extraordinary cases. A rather hard and important task in software engineering is writing code that is maintable, comprehensible and prepared for long term. Continuous integration methods are quite popular because they increase the code quality, and maintainability as they execute many steps in a pipeline when someone makes a change in the code base. These pipelines typically execute the necessary build steps, evaluate the unit, component, integration and end-to-end test cases. Pipelines may use static analyzers to check conventions and subtle language-oriented problems. In this paper, we propose a new step in continuous integration pipelines which aims at the long-term code maintainability based on source code rejuvenation methods that ensure the utilization of more modern standards of the programming language. We take advantage of the Clang compiler infrastructure for our use cases, however, the approach is not specific to a language or another tool. We design a system of standalone plugins in which every plugin is responsible for a separate rejuvenation method. We analyze what are useful options to execute the transformations. We develop a criteria system, and evaluate these options.

Keywords

Continuous Integration, Source Code Rejuvenation, Clang

1. Introduction

One of the most fundamental questions in software engineering is how to write efficient, bug-free, maintainable, convenient source code that can be considered for long term. In modern software engineering, continuous integration (CI) pipelines are utilized often to increase the code quality with the execution of the test cases, and the source code is evaluated with analyzers regularly [1]. Typically, many steps of a comprehensive pipeline are executed when a developer makes a change in the codebase. CI servers provide rapid feedback if the codebase in the repository does not meet the expectations [2].

Programming languages are also in the focus of improvements. Newer and newer standards are available, more and more features the programming languages have and the developers are eager to use these new features [3]. In static analysis, new kind of methods have been proposed for modernize the source code called source code rejuvenation [4]. However, migrating the code base between different standards of a programming language is not that easy because the semantics of the code may change [5]. Special tools are required for the source code rejuvenation [6]. For instance, C++'s exception handling constructs changed significantly, a standalone tool is proposed [7]. The development of user-defined iterators is changed drastically in the recent standards of C++, and a tool for rejuvenation is presented [8].

However, source code rejuvenation methods are not well-known and only stand-alone solutions are available. CI servers are utilized, but typically not support rejuvenation, and thus codebases can become obsolete in the long term. In this paper, we propose an approach which defines a workflow of rejuvenation in CI pipelines.

SQAMIA 2024: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 9–11, 2024, Novi Sad, Serbia

*Corresponding author.

gamesh411@gmail.com (E. Fülöp); patakino@elte.hu (N. Pataki); abel.szauter@gmail.com (Á. Szauter)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attributiousing the International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

CEUR-WS.org/Vol-3845/paper08.pdf

The rest of this paper is organized as follows. Section 2 presents the basic components that are involved in the this research. Related work is discussed in Section 3. The proposed method is presented in Section 4. Finally, this paper is concluded in Section 5.

2. Basic components

2.1. Programming Languages

Many recently used programming languages have a rather long history and during the past years, many standards are released [9]. Programming languages extend their set of constructs when a new standard becomes available [10]. The new constructs make the development easier, make the code more comprehensible or elegant, or increase the safety [11]. However, some constructs can be specified as deprecated or not supported anymore [8]. Moreover, the backward compatibility may have a ungracious impact on the code quality [12]. The relevance of the history overview is important to understand how old code snippets can be survived with memory and safety issues. An important aspect of the history is to check how many subtle standards can be used and how complex the compatibility is.

The Fortran programming language was developed in 1957, therefore it has a long history that can be seen on Table 1. Many standards have been released during this long history, however, in a few cases, there was more than years between two consecutive releases [13].

Table 1
History of the Fortran programming language

FORTRAN	1957
FORTRAN II	1958
FORTRAN III	1958
FORTRAN IV	1961
FORTRAN 66 (ANSI standard)	1966
FORTRAN 77	1978
Fortran 90 (ANSI and ISO standards)	1991
Fortran 95 (ISO standard)	1997
Fortran 2003	2004
Fortran 2008	2010
Fortran 2018	2018
Fortran 2023	2023

The C programming language is older than 50 years, its history can be seen on Table 2. C has less releases compared to Fortran.

Table 2
History of the C programming language

Initial release of C	1972
K&R C	1978
ANSI C, C89 (ANSI & ISO standards)	1989
C99 (ISO standard)	1999
C11	2011
C17	2018
C23	2024

The C++ programming language is based on the C programming language and has a rich history [14]. The language becomes even bigger regularly, it went through a quite radical modernization since 2011, however, many C code is still valid C++. History of C++ can be seen on Table 3.

Table 3
History of the C++ programming language

1st Edt.	1985
C++98 (ISO)	1998
C++03	2003
C++11	2011
C++14	2014
C++17	2017
C++20	2020
C++23	2023

Java has a rather compact history with many standards that can be seen on Table 4. After a rather big intervals, recently two standards are appeared in a year, so the release of the new standards has been sped up.

Table 4
History of the Java programming language

JDK 1.0	23rd January 1996	Java SE 13	17th September 2019
JDK 1.1	18th February 1997	Java SE 14	17th March 2020
J2SE 1.2	4th December 1998	Java SE 15	16th September 2020
J2SE 1.3	8th May 2000	Java SE 16	16th March 2021
J2SE 1.4	13th February 2002	Java SE 17 **(LTS)**	14th September 2021
J2SE 5.0	30th September 2004	Java SE 18	22nd March 2022
Java SE 6	11th December 2006	Java SE 19	20th September 2022
Java SE 7	28th July 2011	Java SE 20	21st March 2023
Java SE 8 **(LTS)**	18th March 2014	Java SE 21 **(LTS)**	19th September 2023
Java SE 9	21st September 2017	Java SE 22 (latest)	19th March 2024
Java SE 10	20th March 2018	Java SE 23	September 2024
Java SE 11 **(LTS)**	25th September 2018	Java SE 24	March 2025
Java SE 12	19th March 2019	Java SE 25 **(LTS)**	September 2025

Backward and forward compatibility is an important aspect in the evolution of standards which mean the code written according to an older standard should work in the very same way according to the new standard. Threrefore semantics of existing codebase should not be changed. However, there are well-known examples in which the compatibility is broken.

The code snippet on Figure 1 has different outcome according to C++03 and C++11 [5].

According to classical C++, the outcome is: `0 0 0 0 0`, however, according to modern C++, the outcome is: `0 1 2 3 4`.

2.2. Static Analysis and Souce Code Rejuvenation

Static analysis is an approach in which one reasons about a program based on the source without execution of the analyzed code [15]. It can be used for many purposes, for instance, finding or predicting bugs, refactoring or visualizing the code, or measuring code complexity [16].

```
class Foo
{
  static int cnt;
public:

  int x;

  Foo(): x( cnt++ ) { }
};

int Foo::cnt = 0;

int main()
{
  std::vector<Foo> v( 5 );
  for( int i = 0; i < v.size(); ++i )
  {
    std::cout << v[ i ].x << ' ';
  }
}
```

Figure 1: C++ code snippet with altering semantics

Nowadays, source code rejuvenation tools started to appear since they are enablers to modernize existing codebases in an automatic or a semi-automatic way. This is a source-to-source transformation that replaces obsolete languages constructs and approaches with modern code [4]. These rejuvenation tools can modify the existing code, so in this sense, they are similar to refactoring tools, however, they differ in many ways too. For instance, rejuvenation is directed transformation that increases the level of abstraction, however, refactoring is not directed, and they have different indicators as well [4]. Static analyzers are used in CI servers to find bugs, but source code rejuvenation tools are typically applied occasionally. Our aim is the application of code rejuvenation regularly in CI environment.

2.3. Clang

Clang is a comprehensive compiler infrastructure based on LLVM [17]. It contains many useful analyzers and tools, a C/C++ parser, an API for the further development of new static analyzers tools. The parser constructs the abstract syntax tree (AST) from the source and this built AST can be processed in a user-defined way. Clang is a popular solution in research, but many big companies take advantage of it. In this paper, we analyze how it can be applied for rejuvenation as a plugin system, and how the validation of the rejuvenation can be executed.

2.4. Continuous Integration

Continuous Integration is a software development methodology that emphasizes the regular merging of the developers' work. The source code is maintained in a version control repository and different steps in CI pipeline are executed when a new version becomes available. This approach provides fast feedback to the developers.

CI methodology is typically supported with CI servers, like Jenkins or GitLab to name a few [18]. These tools are highly configurable, supported with many plugins for the comprehensive evaluation of the source code, the system under development, and the test cases [19].

A typical CI pipeline starts with the compilation and other build processes. After this successful step, different testing methods are launched. Unit testing, component testing, integration testing, end-to-end testing, UI test also can be executed. Static analyzers try to find different kind of bugs, code smells,

checking coding conventions. In case of any problem, developers are notified, and a global dashboard presents the result.

Continuous Delivery (CD) extends CI pipelines with deployment features, and DevOps pipelines can extend the pipeline with operational (e.g. monitoring) features.

3. Related Work

In this section, the related work is discussed. It is divided into two groups, the first one belongs to source code rejuvenation as stand-alone solutions. The second part belongs to the CI's utilization and configuration.

The topic of source code rejuvenation methods is recently researched intensively. For instance, modern way of C++ functors is proposed [20]. Modernization of C++ iterator development is shown [8]. Rejuvenation of C++'s exception specification is also implemented [7]. During the introduction of source code rejuvenation, the authors present a method for utilizing initializer lists [4]. However, these tools are standalone solution, not designed for execution in CI pipelines.

A/B Testing is a method which aims at increased user experience based on the evaluation of two similar versions of the same software [21]. This solution affects the deployed software at runtime, and can be executed via CI/CD pipelines [22].

Refactoring in CI is not a straightforward approach, however, a survey has been evaluated to check whether some developers are eager for it [23]. An automatic bot-like solution has been proposed for regular refactoring without introduction of CI [24]. However, refactoring and code rejuvenation differ in many ways [4].

4. The Proposed Method

In this section, we present our method. This approach has three major parts, the first one belongs to the execution of source code rejuvenation, the second one belongs to the validation whether the meaning of code is unchanged during the rejuvenation, and the third part is pipeline in the CI environment.

4.1. Source Code Rejuvenation

Clang Tooling API provides a powerful framework for building custom static analysis tools. Still, it has a limitation: the Clang driver can only load a single shared object simultaneously. This section describes alternative approaches to running multiple source-code analysis plugins on the analysed project. These approaches are then evaluated for using these analysis tools in a CI/CD pipeline, detailing the advantages and disadvantages.

The workflow involves using the implementations of multiple Clang Tooling plugins sequentially for a single source file. It assumes that there is no interference in the plugins, i.e., they do not modify the same files on disk, and there is the assumption that all of these plugins treat the source AST in a read-only fashion. Consequently, there is also the assumption that the order of execution does not matter (both in the plugins ran and the source files analyzed). The trivial use-case is that given the project description, i.e., the list of all compiler invocations that produce the end libraries and binaries, each tool should be run on each translation unit once.

The naive approach is the default supported workflow, which means running a Clang invocation for each plugin and each translation unit (see Figure 2). This is the most documented usage of the tool, and it requires no modification in the source or binary formats of the plugin or Clang. The downside is that it requires N times M clang invocations, where N is the number of plugins, and M is the number of translation units.

The next option is the implementations of multiple Clang Tooling plugins are merged into a single shared object, as seen in Figure 3. This shared object is then passed to the Clang driver, which loads it and executes all the contained plugins in a single run. This approach avoids the overhead of parsing the same

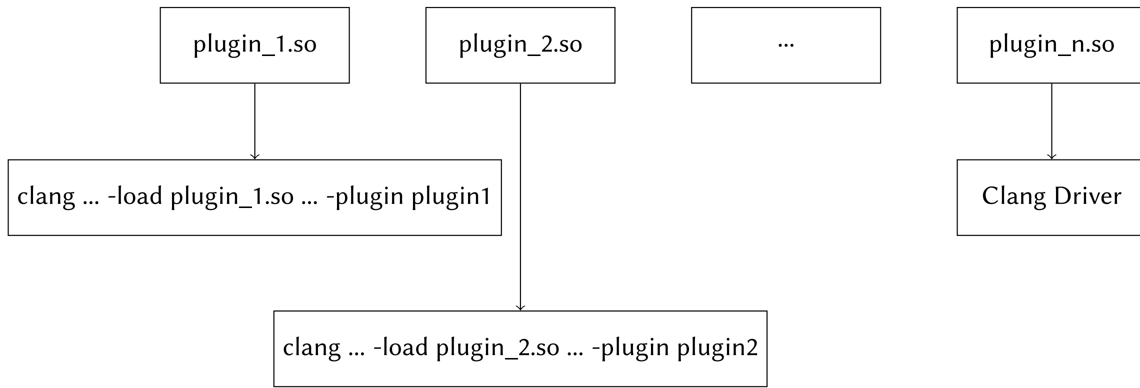

Figure 2: Workflow of Running Plugins Separately

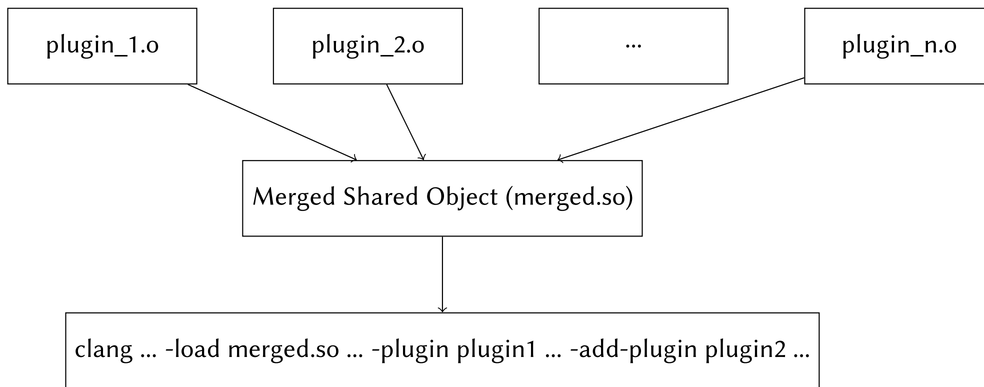

Figure 3: Workflow of Merging Multiple Plugins into a Single Shared Object

file multiple times, which would be necessary if the plugins were run separately. The disadvantage is that merging the plugin source code requires merging, which may lead to code maintenance challenges.

In the next option, a custom `MultiplexConsumer` combines multiple `ASTConsumers` from different plugins into a single consumer. This allows multiple plugins to be executed in a single Clang run without merging their implementations, thus simplifying the loading process compared to Approach #1. The downside is that this requires modifying the plugin source, as the `ASTConsumer` subclass types must be extracted and exposed to the proxy plugin implementation.

In the next option, we modify the plugins to support an extern C API for creating and destroying instances of the `ASTConsumer` subclass. The proxy plugin can dynamically load these shared objects and instantiate the `ASTConsumers`. This approach requires modifying the plugin source but allows dynamic plugin loading. The disadvantage is that this requires modifying the source of the plugins to support instantiating the `ASTConsumer` subclasses via C-language-linkage functions.

In next option, we patch Clang to support loading multiple shared object files. This is the most intrusive change, as it requires deploying a custom Clang binary. However, it allows the most flexibility in loading plugins, and it does not require modifying the plugins' source code.

The comparion and evaluation of the analyzed approaches can be found in Table 5.

4.2. Validation

Upgrading C++ code from an older standard to a newer one carries the risk of semantic differences, which means that compiling and running the upgraded code may yield different results. Investigating these differences is both necessary and important.

Table 5
Comparison of approaches to execute multiple plugins

Criteria	Naive	#1	#2	#3	#4
Requires multiple Clang invocations to use	Yes	No	No	No	No
Requires recompilation of Clang	No	No	No	No	Yes
Requires refactoring of the plugin	No	No	Yes	Yes	No
Requires recompilation of the plugin	No	No	Yes	Yes	No
Requires relinking of the plugin	No	Yes	Yes	No	No

During the compilation of C++ code, the compiler constructs an Abstract Syntax Tree (AST), which stores pointers to the parsed statements and expressions. Tools such as clang-tidy allow us to check, filter, and even modify the AST. By comparing the ASTs of code compiled with the older and newer standards, we can identify the main differences.

Using clang++, we can generate an AST dump which is the representation of the AST in raw text. However, comparing these dumps directly can be challenging due to the memory addresses of the pointers shown in the AST which most of the time are different.

To efficiently work with ASTs, we can use Clang LibTooling, a well-known C++ library that allows developers to utilize the Clang AST for software development. The LibTooling library provides the `RecursiveASTVisitor` class, which facilitates the comparison of two ASTs by allowing us to easily iterate over the AST nodes using a Depth-First Search (DFS) algorithm.

Although comparing ASTs using DFS sounds straightforward, there may be differences that do not affect the results because they do not involve semantic changes. Over the years, the C++ standard library has undergone many changes, making it safer and more optimized. While these improvements should not impact the program's output, they can alter the AST. Therefore, it is crucial to distinguish differences that affect semantics from those that do not.

To ignore differences that do not affect the result, we can use other methods besides AST comparison. Symbolic execution, for instance, allows us to determine the program's output without running it by replacing variables with symbolic values. This technique is powerful for program analysis and verification.

Another approach is to check for differences between standards and determine whether a given statement has changed. The cppreference.com site documents changes between standards, but having a database with these changes would be more efficient for searching than using the cppreference site directly.

There are also tools such as Creduce and Delta Debugging, which run the code with different standards and investigate the differences.

4.3. Workflow in Continuous Integration Pipelines

In the workflow, we evolve how to use the proposed methods to ensure continuous rejuvenation in a CI pipeline.

The proposed workflow consists of the following steps:

1. Evaluation of code as it is currently available in the repository
2. Execute the source code rejuvenation processes
3. Validate whether the rejuvenation does not break the backward compatibility
4. Request for approvement
5. Evaluation of rejuvenated code
6. The rejuvenated code is commited back to repository

The pipeline starts with the ordinary validation of the committed code. If no problem is found the rejuvenation process is executed and checks if there is no compatibility issue. In the pipeline, an approvement should be required for the responsibility. After the approvement, the rejuvenated code is

evaluated as a usual CI solution. If any step of the pipeline fails, the entire process is stopped and the problem is reported that can be evaluated. If anything is fine, than the code is committed back which means the pipeline transaction is committed as well.

The implementation of this pipeline is ongoing. We started to use Jenkins as a CI server because it is widely-used and comprehensively configurable [2].

5. Conclusion and Future Work

In this paper, a rather important question is discussed: how we can write source code that is bug-free, modern, maintainable. Programming languages have a quite long history, and because of the backward compatibility, codebases may become old in a bad way. Source code rejuvenation solutions are recently researched, but the proposed tools are standalone ones. Continuous Integration methods are popular because of the rapid feedback to developers if something goes wrong in the repository. We propose an approach in which rejuvenation methods become the part of the CI pipeline. We analyze how to deal this issue in different aspects. We analyze how to integrate the rejuvenation methods, how to validate them and how CI pipeline can introduce this solution.

Currently, our approach belongs to the C/C++ realm, we use the Clang compiler infrastructure. However, the ideas are not specific to a language or a CI tool, but general ones. As a future work, we should create a generic solution that can be specified for a programming language. Moreover, a general CI plugin can be implemented for a seamless work.

References

[1] A. Révész, N. Pataki, Integration heaven of nanoservices, in: Proceedings of the 21st International Multi-Conference INFORMATION SOCIETY, IS'2018, volume Volume G: Collaboration, Software and Services in Information Society, 2018, pp. 43–46.

[2] Á. Révész, N. Pataki, Visualisation of Jenkins pipelines, Acta Cybernetica 25 (2021) 877–895. URL: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4119. doi:10.14232/actacyb.284211.

[3] B. Stroustrup, Thriving in a crowded and changing world: C++ 2006–2020 4 (2020). URL: https://doi.org/10.1145/3386320.

[4] P. Pirkelbauer, D. Dechev, B. Stroustrup, Source code rejuvenation is not refactoring, in: J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, B. Rumpe (Eds.), SOFSEM 2010: Theory and Practice of Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 639–650. doi:10.1007/978-3-642-11266-9_53.

[5] T. Brunner, N. Pataki, Z. Porkoláb, Backward compatibility violations and their detection in C++ legacy code using static analysis, Acta Electrotechnica et Informatica 16 (2016) 12–19. URL: http://aei.tuke.sk/papers/2016/2/03_Bruner.pdf.

[6] W. Lucas, F. Carvalho, R. C. Nunes, R. Bonifácio, J. Saraiva, P. Accioly, Embracing modern C++ features: An empirical assessment on the KDE community, Journal of Software: Evolution and Process 36 (????) e2605. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2605. doi:https://doi.org/10.1002/smr.2605.

[7] E. Fülöp, A. Gyén, N. Pataki, C++ source code rejuvenation for an improved exception specification, in: 2022 IEEE 16th International Scientific Conference on Informatics (Informatics), 2022, pp. 94–99. doi:10.1109/Informatics57926.2022.10083493.

[8] D. Kolozsvári, N. Pataki, A static analysis approach for modern iterator development, Annales Mathematicae et Informaticae 59 (2023) 37–53. doi:10.33039/ami.2023.08.006.

[9] T. Brunner, Z. Porkoláb, Programming language history: Experiences based on the evolution of C++, in: Proceedings of the 10th International Conference on Applied Informatics, 2017, pp. 63–71. doi:10.14794/ICAI.10.2017.63.

[10] A. M. Gorelik, Statements, data types and intrinsic procedures in the Fortran standards (1966-2008) 33 (2014). URL: https://doi.org/10.1145/2701654.2701655.

[11] G. Dévai, N. Pataki, Towards verified usage of the C++ Standard Template Library, in: Proceedings of the 10th Symposium on Programming Languages and Software Tools, SPLST 2007, 2007, pp. 360–371.

[12] A. Sundelin, J. Gonzalez-Huerta, K. Wnuk, The hidden cost of backward compatibility: when deprecation turns into technical debt - an experience report, Association for Computing Machinery, New York, NY, USA, 2020. URL: https://doi.org/10.1145/3387906.3388629.

[13] M. Metcalf, J. Reid, M. Cohen, R. Bader, Modern Fortran Explained: Incorporating Fortran 2023, Oxford University Press, 2023.

[14] B. Stroustrup, The C++ Programming Language, 4th ed., Addison-Wesley Professional, 2013.

[15] B. Babati, N. Pataki, The role of implementation-specific static analysis, in: 2023 International Conference on Software and System Engineering (ICoSSE), 2023, pp. 1–6. doi:10.1109/ICoSSE58936.2023.00009.

[16] J. Malm, E. Enoiu, M. A. Naser, B. Lisper, Z. Porkoláb, S. Eldh, An evaluation of general-purpose static analysis tools on C/C++ test code, in: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2022, pp. 133–140. doi:10.1109/SEAA56994.2022.00029.

[17] B. Babati, G. Horváth, V. Májer, N. Pataki, Static analysis toolset with Clang, in: Proceedings of the 10th International Conference on Applied Informatics, 2017, pp. 23–29. doi:10.14794/ICAI.10.2017.23.

[18] M. Shahin, M. Ali Babar, L. Zhu, Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, IEEE Access 5 (2017) 3909–3943. doi:10.1109/ACCESS.2017.2685629.

[19] C. Singh, N. S. Gaba, M. Kaur, B. Kaur, Comparison of different CI/CD tools integrated with cloud platform, in: 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence), 2019, pp. 7–12. doi:10.1109/CONFLUENCE.2019.8776985.

[20] G. Horváth, N. Pataki, Transparent functors for the C++ Standard Template Library, in: E. Vatai (Ed.), Proceedings of the 11th Joint Conference on Mathematics and Computer Science, CEUR-WS, 2016, pp. 96–101.

[21] Á. Révész, N. Pataki, Containerized A/B testing, in: Z. Budimac (Ed.), Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, CEUR-WS.org, 2017, pp. 14:1–14:8. URL: http://ceur-ws.org/Vol-1938/paper-rev.pdf.

[22] Á. Révész, N. Pataki, A/B testing via Continuous Integration and Continuous Delivery, in: S. Bourennane, P. Kubicek (Eds.), Geoinformatics and Data Analysis, Springer International Publishing, Cham, 2022, pp. 165–174. doi:10.1007/978-3-031-08017-3_15.

[23] C. Vassallo, F. Palomba, H. C. Gall, Continuous refactoring in CI: A preliminary study on the perceived advantages and barriers, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 564–568. doi:10.1109/ICSME.2018.00068.

[24] M. Wyrich, J. Bogner, Towards an autonomous bot for automatic source code refactoring, in: 2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE), 2019, pp. 24–28. doi:10.1109/BotSE.2019.00015.