applied sciences

MDPI

Communication

Towards Continuous Deployment for Blockchain

Tomasz Górski

Department of Computer Science, Polish Naval Academy of the Heroes of Westerplatte (PNA), Śmidowicza 69, 81-127 Gdynia, Poland; t.gorski@amw.gdynia.pl

Abstract: Ensuring a production-ready state of the application under development is the immanent feature of the continuous delivery approach. In a blockchain network, nodes communicate, storing data in a decentralized manner. Each node executes the same business application but operates in a distinct execution environment. The literature lacks research, focusing on continuous practices for blockchain and distributed ledger technology. In particular, such works with support for both software development disciplines of design and deployment. Artifacts from considered disciplines have been placed in the 1 + 5 architectural views model. The approach aims to ensure the continuous deployment of containerized blockchain distributed applications. The solution has been divided into two independent components: Delivery and deployment. They interact through Git distributed version control. Dedicated GitHub repositories should store the business application and deployment configurations for nodes. The delivery component has to ensure the deployment package in the actual version of the business application with the node-specific up-to-date version of deployment configuration files. The deployment component is responsible for providing running distributed applications in containers for all blockchain nodes. The approach uses Jenkins and Kubernetes frameworks. For the sake of verification, preliminary tests have been conducted for the Electricity Consumption and Supply Management blockchain-based system for prosumers of renewable energy.

Keywords: blockchain; continuous deployment; 1 + 5 architectural views model; model-driven development

Citation: Górski, T. Towards Continuous Deployment for Blockchain. *Appl. Sci.* **2021**, *11*, 11745. https://doi.org/10.3390/app112411745

Academic Editor: Jason K. Levy

Received: 17 November 2021
Accepted: 9 December 2021
Published: 10 December 2021

1. Introduction

The primary principle of the Agile Manifesto underlines the importance of early and continuous delivery of software that meets the customer needs [1]. Abbreviations have been provided for the following notions: Continuous integration (CI), continuous delivery (CD), and continuous deployment (CDT). The first practice involves that software is integrated continuously during development. The practice encompasses automation of software builds and testing. The backbone of a CI is version control. The most popular GitHub service ensures the distributed version control of source code using Git. The new or changed code is incorporated into a build and checked by automated tests. Automation of testing ensures checking that the application works correctly in case new commit is merged with the release branch. The CD approach goes even further in software development automation. It aims to enable on-demand software release. CD employs a set of stages, including the acceptance tests and release process. Automated acceptance tests and the release process allow on deployment of application under development on demand. Humble and Farley [2] present a comprehensive description of the continuous delivery process. They have defined the notion of the deployment pipeline as an automated process of tasks, which is responsible for producing a release. CD practice requires specific governance to act properly, i.e., infrastructure, data, and configuration management. The CDT approach elevates automation on an even higher level. It automatically deploys every release to users' acceptance tests environments or even a production one. CDT is a push-based practice. Conversely, CD approach is a pull-based one.

Figure 1 presents loop of steps in continuous integration and continuous delivery approaches. The loop makes a complete CDT approach.

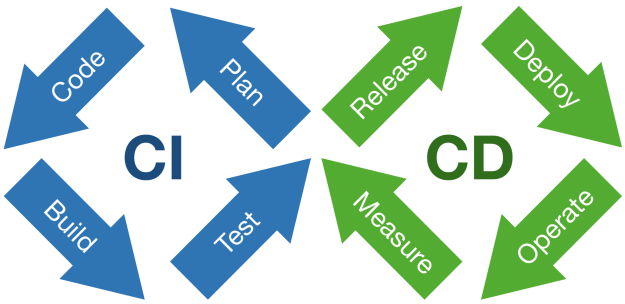

Figure 1. The Möbius strip in continuous software engineering.

The author has observed increasing interest in continuous approaches. Shahin et al. [3] have found the following important topics: Reducing build and test time, automation of tests, raising the scalability of deployment pipelines, and elevating the deployment process reliability. They have also enumerated essential elements in implementing continuous approaches. They have underlined suitable infrastructure, testing, highly skilled programmers, and a proven design process. Debroy and Miller [4] show actions to overcome challenges in implementing continuous practices. They use custom images for building agents to handle micro-service dependencies. They also apply orchestration to manage resources in order to keep infrastructure costs low. Keeping build and release times short requires employing an orchestrator, such as Kubernetes (K8s), to handle scaling. Recently, IEEE Standard for DevOps has been approved and published [5]. The standard provides requirements and guidance on the implementation of DevOps to define, control, and improve software life cycle processes. It applies within an organization or a project for building and deploying software reliably. The efficiency of the continuous integration process has been improved by Abdalkareem et al. [6] through identifying commits that can be skipped. The proposed prototype tool works with *Git* repositories. Continuous integration and delivery tools have been analyzed by Prado Lima et al. [7] in the view of test case prioritization. They have analyzed the following environments: BuildBot, GoCD, Integrity, Jenkins, and Travis CI. For this work, GoCD and Jenkins were considered. Both are open-source, written in Java, and integrate well with both K8s and GitHub. However, Jenkins offers better support for continuous integration. These environments do not provide ready-made functions for blockchain. However, they offer an application programming interface (API) and thus the possibility of extending their functionality. A newly published paper by Leite et al. [8] touches on continuous delivery practice. They have analyzed the structure of DevOps teams and communication between them. So, the subject seems to be timely.

The underlying technology for blockchain is distributed ledger. Xu et al. [9] provide the following definition of a distributed ledger: "A distributed ledger is an append-only store of transactions which is distributed across many machines." A consensus algorithm is a fundamental component of a distributed ledger and blockchain that ensures synchronization among multiple peers. There are two main well-established consensus algorithms: Proof-of-work (PoW), and proof-of-stake (PoS). The author would like to draw attention to the fact that there are works on alternative approaches [10,11]. Blockchain is a disruptive technology. Casino et al. [12] have done a thorough literature review of blockchain-based applications. They enumerate many uses, but emphasize the vast opportunities in the energy sector. Researchers and practitioners use various blockchain frameworks. An extensive comparison of permissioned (private) and permissionless (public) blockchain frameworks have been done by Chowdhury et al. [13]. They have chosen, e.g., Hyperledger Fabric, Ethereum, IOTA, Multichain, and R3 Corda. The latter is a private ledger where consensus involves two DLT nodes that participate in the transaction, which is signed by the Notary node. It has a vast impact on scalability. In the ledger, there is also a Network Map node and Oracle nodes. In a transaction, two DLT nodes and one

Notary node take part. The R3 Corda's block creation time is 0.5–2.0 [s], which places the framework among the fastest. Moreover, the usage of energy by the framework is almost negligible. A DLT node hosts distributed applications (CorDapps) and services. The main services are oracle, notary, network map, and permissioning. A Corda network is a fully connected graph. The communication among DLT nodes and the Notary node is done via an Advanced Message Queuing Protocol over Transport Layer Security (AMQP/TLS). Moreover, it uses a Hypertext Transfer Protocol Secure (HTTPS) for the communication of DLT nodes with the Network Map and Oracle nodes. In the case of energy systems, permissioned blockchain networks fit best. They fulfill privacy requirements and can facilitate peer-to-peer energy exchange. The approach uses the R3 Corda blockchain-based Electricity Consumption and Supply Management System (ECSM) that enables energy exchange between prosumers.

The contribution is the continuous deployment approach for generating complete node deployment packages for blockchain nodes and running the blockchain network in a containerized environment. Figure 2 shows a scheme of the solution.

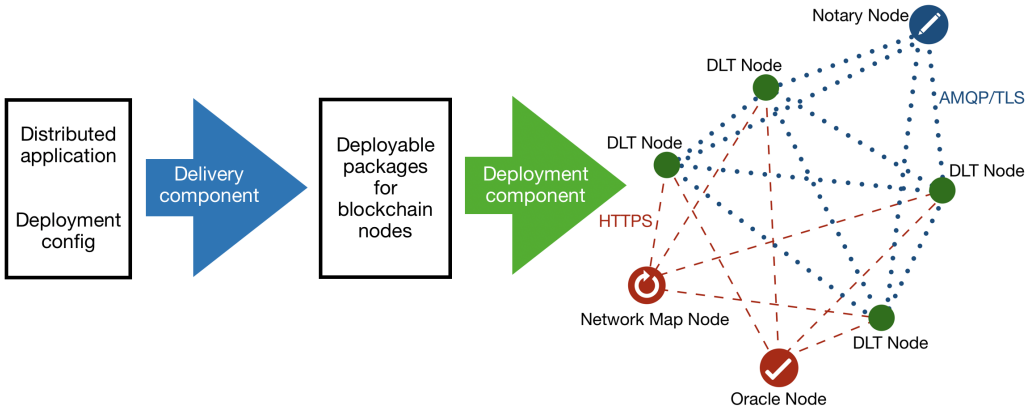

Figure 2. The approach overview.

Blockchain introduces an additional level of difficulty for continuous practices. In addition to ensuring the proper functioning of the business application, there are a variety of deployment configurations for blockchain nodes. Ozkaya et al. [14] have found that the functional and information views are the most popular ones in software architecture modeling. Zou et al. [15] have conducted an analysis to discover the actual obstacles that developers have to overcome while developing smart contracts. Results revealed that the source code of smart contracts is compromised as far as security is concerned. Besides, they claim that the development support of blockchain applications in existing tools is still incomplete. The approach offers design support for the *Deployment* view and uses the Unified Modeling Language (UML) Profile for Distributed Ledger Deployment. The key element in a blockchain is a smart contract. The approach offers a unique *Contracts* view to describe smart contracts within the 1 + 5 architectural views model. Górski in [16] has shown the *Smart Contract Design Pattern* that offers a flexible manner for designing smart contracts in a permissioned distributed ledger. The pattern has been incorporated into the delivery component. That component uses the Visual Paradigm modeling tool and automates build release tasks with the Jenkins server. The continuous deployment component, still under design, uses the Kubernetes platform for automating the deployment of blockchain distributed applications in containers.

The following part of the communication is arranged as follows. Section 2 outlines the design of the delivery component. Section 3 discusses the preliminary design of the deployment component. Section 4 introduces the method of validation of the delivery component that allows for checking the consistency between generated deployment scripts and UML models. The section also presents the validation method of the business application. Section 5 concludes the work done and shows already planned tasks.

2. The Design of the Delivery Component

Various architectural principles have been applied to the design of the continuous delivery component. Firstly, there have been imposed *Modularity* and *Separation of responsibility* principles on the approach design. The following layers have been identified: Design & Development, Version control, and Build automation pipelines. The *Design & Development* layer consists of the *UML Deployment model* of the distributed ledger solution and the model-to-code transformation. The model uses the *UML Profile for Distributed Ledger Deployment* profile. The second module is the *Java distributed application*, which realizes a smart contract. The first module is designed in Visual Paradigm whereas the second one is developed in IntelliJ IDEA. Both tools have a community edition. The IntelliJ IDEA works smoothly with GitHub repositories. Java has been chosen for portability reasons but has two uses. It has been used to implement a distributed blockchain application. Moreover, it is the language of developing the transformation application and plug-in for Visual Paradigm. The *Version control* layer comprises Git repositories. The first one encompasses the source code of the smart contract application and Corda execution environment. The second repository consists of deployment configuration files for DLT nodes. The *Build automation pipelines* layer encompasses both Jenkins pipelines. The first one automates the generation of the smart contract application and Corda runtime JAR files. The second pipeline automates the generation of the complete ZIP file that consists of the application, Corda runtime, and deployment configuration files for nodes. Deployment packages for nodes are stored by the open source Jenkins automation server. The delivery component is still under construction. Figure 3 depicts the overview of the delivery component for generating blockchain deployment packages.

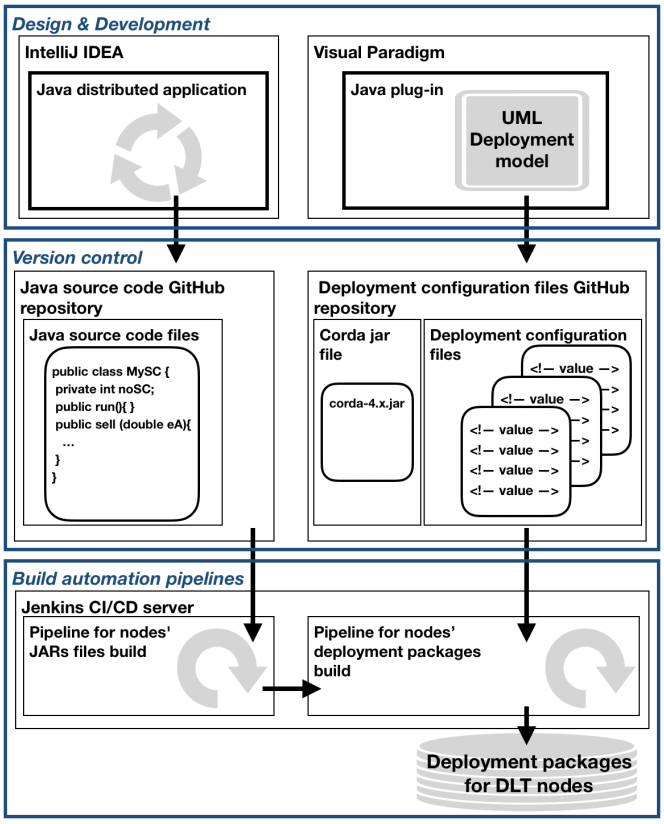

Figure 3. The delivery component overview.

2.1. UML Profile for DLT Deployment

The approach concentrates on the Platform Specific Model (PSM) to express the precise deployment configuration of the R3 Corda framework in version 4.6. Stereotypes and tagged values have been used for defining the profile with the needed semantic enrichment

for *Deployment view* modeling. Stereotypes have been applied to represent nodes, services, and communication protocols. Tagged values have been used to define deployment configuration parameters for nodes. First, tagged values have been identified that describe parameters common for all types of nodes and placed in the ≪CordaNode≫ stereotype. Then, there have also been defined tagged values for the notary node and placed in the ≪NotaryNode≫ stereotype. In the current version of the profile, the deployment parameters of the notary node operating in the high availability mode has been taken into account [16,17]. All stereotypes for Corda network nodes have a common set of tagged values because they inherit from the ≪CordaNode≫ stereotype.

2.2. The Model-to-Code Transformation

The delivery component incorporates model-to-code transformation for generating blockchain deployment scripts [18]. The source of the transformation is a UML Deployment model. The current version of the UML profile has been used. The second vital change in the design of the transformation is the ability to store generated deployment scripts at GitHub under Git version control. The transformation ensures consistency of the *UML Deployment model* with deployment configuration files of blockchain nodes.

2.3. Delivery Pipelines

The open-source Jenkins automation server has been used. The main notion in the Jenkins automation server is a pipeline. A pipeline can be described as an automated process of generating a releasable package on the basis of software stored under version control. A pipeline defines your entire build process, which typically includes stages. A stage block defines a conceptually distinct subset of tasks. A single task tells Jenkins what to do at a particular step in the process. Correct node deployment package involves both business logic and deployment configuration details. Thus two Jenkins pipelines have been introduced. The *Build node deployment package* pipeline is responsible for completing the full node deployment package using CorAapps jar files and deployment configuration scripts. The *Build Cordapps jar files* pipeline is responsible for building an up-to-date version of CorDapps JAR files and is triggered when a new source code of business logic appears in the repository. After finishing execution, the pipeline triggers the *Build node deployment package* pipeline. Such division allows for the separation of the Java source code of business logic from the node deployment configuration. The pipeline *Build node deployment package* is triggered when a new deployment configuration is generated from the UML Deployment model.

Figure 4 depicts the UML activity diagram with designed pipelines.

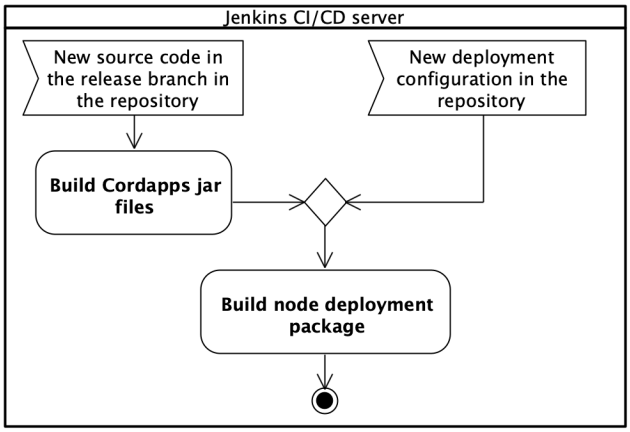

Figure 4. Pipelines of the delivery component.

Both elements, the redesigned model-to-code transformation and delivery pipelines are still under construction. Figure 5 shows results of the single execution of the *Build Cordapps jar files* pipeline. The Jenkins automation server collects values of the following

metrics: *Average stage time* and *average full run time*. The pipeline execution time for the blockchain network of 5 business nodes is under 50 [s]. It looks promising in view of processing larger networks.

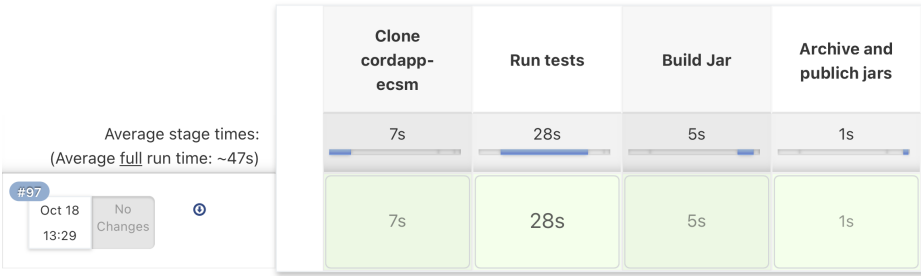

Figure 5. Results of the pipeline execution.

The Jenkins server with implemented pipelines has been exposed in the domain of the statutory project: *model.amw.gdynia.pl*.

3. The Design of the Deployment Component

Containers have a lot in common with virtual machines. They are considered lightweight because they use the same operating system layer. The purpose of the deployment component is to provide a fully working container infrastructure for the business application. The component is designed in form model-to-code transformation. It uses the UML Deployment model and generates YAML files. To be specific, the transformation uses XML files of UML models created in Visual Paradigm. The transformation creates the internal data model of the Kubernetes application and processes it into the working set of YAML files. That set can be further sent to the K8s cluster. The transformation is written in the JavaScript programming language for Node.JS runtime. It has no system-level dependencies, thus it can be run on hosts with various operating systems, e.g., Windows, Linux, and macOS. Moreover, the transformation can be run directly in the K8s Pod as an application/service. Figure 6 depicts a model of K8s application. The deployment component is still under construction.

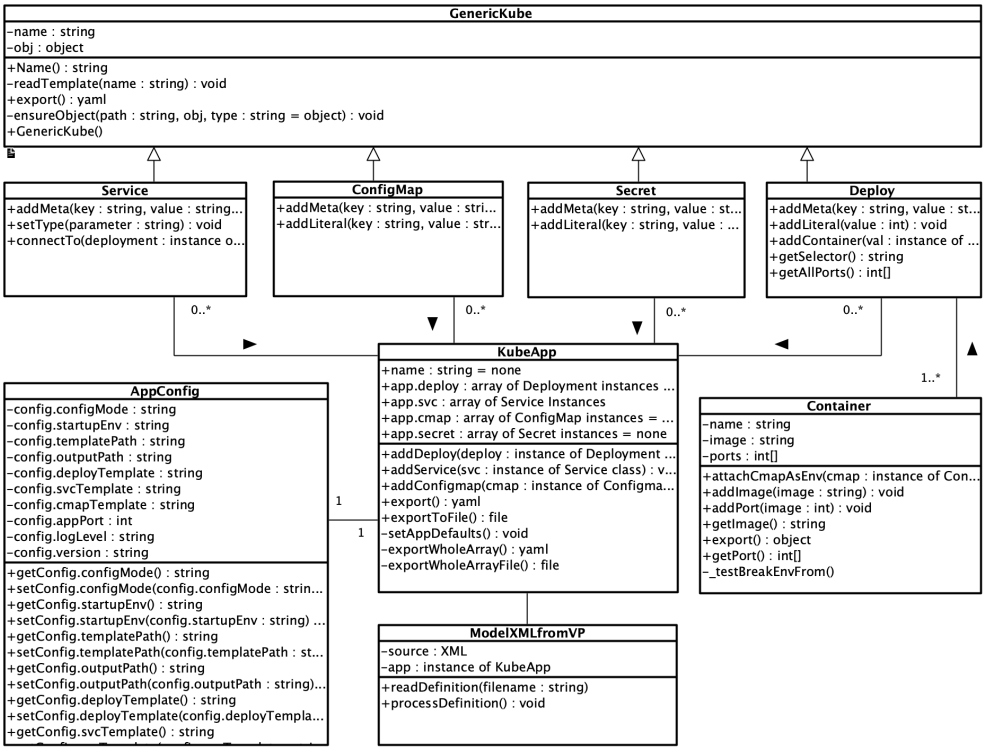

Figure 6. The UML class diagram that shows a model of the K8s application.

4. Validation

Validation determines whether a system or component satisfies requirements specified by the user. Validation of such a solution includes two stages: Delivery and deployment. At both stages, the tests must concern the business application and deployment configurations of the blockchain network nodes. At the delivery stage, unit tests are designed for individual methods in the business application and integration tests verify the operation of the business application in blockchain nodes running on the test environment. The same set of tests must be passed by the business application at containerized environments e.g., user acceptance tests and staging.

As far as deployment configurations are concerned, the consistency of deployment scripts with the UML Deployment model should be verified. A single deployment configuration file and the corresponding UML node are considered. The UML node comprises tagged values, $t \in T$. The script contains deployment configuration parameters, $d \in D$. Intersection of two sets D and T is denoted by $D \cap T$, and is the set containing all elements of D that also belong to T or similarly, all elements of T that also belong to D. It means checking that the intersection meets the following Equation (1):

$$D \cap T = D = T. \tag{1}$$

The cardinality of both sets should be the same. The deployment configuration parameter d is an ordered pair, $d = (n^d, v^d)$, where: n^d is the name, and v^d is the value of d. The tagged value t is an ordered pair, $t = (n^t, v^t)$, where: n^t is the name, and v^t is the value of t. For each $d \in D$, there must be $t \in T$ with the same name and value (2).

$$\bigwedge_{d \in D} \bigvee_{t \in T} (n^d = n^t) \wedge (v^d = v^t). \tag{2}$$

Similarly, for each $t \in T$, there must be $d \in D$ with the same name and value (3).

$$\bigwedge_{t \in T} \bigvee_{d \in D} (n^t = n^d) \wedge (v^t = v^d). \tag{3}$$

Apart from verifying deployment configuration files, there are designed tests for the business application. A dedicated test class has been designed for testing verification rules in the smart contract. Figure 7 depicts the test case for one of the smart contract verification rules. The preliminary tests have been conducted for the ECSM system [19].

```
@Test
    public void transactionMustHaveOneOutput_pass() {
        ledger(ledgerServices, (ledger -> {
            ledger.transaction(tx -> {
                tx.output(IOUContract.ID, new IOUState(iouValue,
                gdyniaB.getParty(), gdyniaA.getParty()));
                tx.command(ImmutableList.of(gdyniaA.getPublicKey(),
                gdyniaB.getPublicKey()),
                new IOUContract.Commands.Create());
                tx.verifies();
                return null;
            });
            return null;
        }));
    }
```

Figure 7. The source code of the positive test case.

Tests have confirmed that the delivery component works correctly.

5. Conclusions

The paper introduces the continuous deployment framework for generating the distributed application for blockchain nodes. Nodes are deployed and operated in containers. By applying the *Loosely coupling* architectural rule, it is possible to implement both components independently. The delivery component offers UML modeling support for the

deployment architectural view. The component also delivers flexibly designed smart contracts in distributed blockchain applications. The Jenkins pipelines read smart contract applications and deployment configuration files stored in Git repositories and provide complete deployment packages. The deployment component reads the UML deployment model and generates YAML files needed to run the distributed application in containers. Containerized environments ensure the reconfigurability of the blockchain network. But each permissioned blockchain community should be deployed and operated on a separate physical machine. Another goal is to elevate the design level of smart contracts and deployment configurations. The research also encompasses security algorithms for registering a new node in the blockchain network and making transactions. The main aim is to achieve automation of the deployment of reconfigurable blockchain networks with updatable and extensible smart contracts at runtime.

Funding: The research has been conducted within the *Architectural views model of cooperating IT systems* project, financed by the statutory funds of the Department of Computer Science, PNA.

Conflicts of Interest: No conflict of interest to declare.

References

1. The Agile Manifesto. Principles behind the Agile Manifesto. Available online: agilemanifesto.org/principles.html (accessed on 7 November 2021).
2. Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed.; Addison-Wesley Professional: Crawfordsville, IN, USA, 2010.
3. Shahin, M.; Babar, M.A.; Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* **2017**, *5*, 3909–3943. [CrossRef]
4. Debroy, V.; Miller, S. Overcoming Challenges with Continuous Integration and Deployment Pipelines: An Experience Report From a Small Company. *IEEE Softw.* **2020**, *37*, 21–29. [CrossRef]
5. Std 2675-2021, IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment, 16 April 2021; pp. 1–91. Available online: https://ieeexplore.ieee.org/servlet/opac?punumber=9415474 (accessed on 7 November 2021). [CrossRef]
6. Abdalkareem, R.; Mujahid, S.; Shihab, E.; Rilling, J. Which Commits Can Be CI Skipped? *IEEE Trans. Softw. Eng.* **2021**, *47*, 448–463. [CrossRef]
7. Lima, J.A.P.; Vergilio, S.R. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Inf. Softw. Technol.* **2020**, *121*, 106268. [CrossRef]
8. Leite, L.; Pinto, G.; Kon, F.; Meirelles, P. The organization of software teams in the quest for continuous delivery: A grounded theory approach. *Inf. Softw. Technol.* **2021**, *139*, 106672. [CrossRef]
9. Xu, X.; Weber, I.; Staples, M. *Architecture for Blockchain Applications*; Springer: Cham, Switzerland, 2019; pp. 5–7. [CrossRef]
10. Oyinloye, D.P.; Damilare, P.; Teh, J.S.; Jamil, N.; Moatsum, A.M. Blockchain Consensus: An Overview of Alternative Protocols. *Symmetry* **2021**, *13*, 1363. [CrossRef]
11. Ma, J.; Jo, Y.; Park, C. PeerBFT: Making Hyperledger Fabric's Ordering Service Withstand Byzantine Faults. *IEEE Access* **2020**, *8*, 217255–217267. [CrossRef]
12. Casino, F.; Dasaklis, T.K.; Patsakis, C. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telemat. Inform.* **2019**, *36*, 55–81. [CrossRef]
13. Chowdhury, M.J.M.; Ferdous, M.S.; Biswas, K.; Chowdhury, N.; Kayes, A.S.M.; Alazab, M.; Watters, P. A Comparative Analysis of Distributed Ledger Technology Platforms. *IEEE Access* **2019**, *7*, 167930–167943. [CrossRef]
14. Ozkaya, M.; Erata, F. A survey on the practical use of UML for different software architecture viewpoints. *Inf. Softw. Technol.* **2020**, *121*, 106275. [CrossRef]
15. Zou, W.; Lo, D.; Kochhar, P.S.; Le, X.D.; Xia, X.; Feng, Y.; Chen, Z.; Xu, B. Smart Contract Development: Challenges and Opportunities. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2084–2106. [CrossRef]
16. Górski, T. The 1 + 5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions. *Symmetry* **2021** , *13*, 2000. [CrossRef]
17. GitHub Repository with the UML Profile for Distributed Ledger Deployment. Available online: https://github.com/drGorski/UMLProfileForDLT (accessed on 7 November 2021).
18. Górski, T.; Bednarski, J. Applying Model-Driven Engineering to Distributed Ledger Deployment. *IEEE Access* **2020**, *8*, 118245–118261. [CrossRef]
19. GitHub Repository with the ECSM Implementation. Available online: https://github.com/drGorski/renewableEnergyBlockchain (accessed on 7 November 2021).