DevOps for Ethereum Blockchain Smart Contracts

Maximilian Wöhrer, Uwe Zdun
University of Vienna, Faculty of Computer Science, Research Group Software Architecture
Vienna, Austria
{maximilian.woehrer,uwe.zdun}@univie.ac.at

***Abstract*—With the evolution and proliferation of blockchain, the technology is becoming more prevalent in enterprise software development. Using the already proven DevOps approach in this setting makes sense, as it can accelerate the general pace of software development and delivery, improve software quality, and increase overall productivity. However, there is currently a lack of guidance on a structured DevOps approach and a breakdown of the specifics in the context of blockchain-based software development. Therefore, we combined gray literature and DevOps application studies from pertinent GitHub projects to systematically investigate current practices and solution approaches for an efficient blockchain-oriented DevOps procedure. In this process, we elaborated procedural steps and related activities according to the main stages of Continuous Integration and Continuous Delivery. Our research shows that core DevOps concepts and activities are similar to other areas and are entirely possible with already established CI/CD solutions that orchestrate the right tools, with the difference that more rigorous testing and differentiated deployment practices are required due to the inherent immutability of blockchain.**

***Index Terms*—blockchain, DevOps, software engineering, smart contract**

I. Introduction

DevOps and blockchains are two hype terms of the recent past. DevOps is a multi-layered concept that is not easy to grasp and can be defined in many ways [1]. In its broadest sense, DevOps refers to the combination of software development (Dev) and operations (Ops) with a focus on cross-organizational integration to bridge the gap between different stages of the software life cycle. Two core aspects of DevOps are Continuous Integration (CI) and Continuous Delivery/Deployment (CD), which support the DevOps principle of interlocking the two underlying disciplines through a high degree of automation. CI usually refers to integrating, building, and testing code, whilst CD is primarily about automating the deployment and release engineering process.

Blockchains, on the other hand, combine various computational and economic concepts to provide a fraud-free intermediary platform for efficiently settling transactions between different parties. In this context, shared business processes can be realized through application code running autonomously on the blockchain to digitally facilitate, verify, and enforce the execution of arbitrary terms via smart contracts. As a special feature, smart contracts are usually not subject to a normal software life cycle, in which a new code version may add features or fix bugs. This circumstance means that software quality and reliability are important pillars in development. In this and various other respects, DevOps can provide valuable support, be it through test automation or the provision of stable operating environments. However, at the moment there is a lack of a structured approach and breakdown of the specifics regarding DevOps usage in this area. To address this gap, we explore DevOps approaches and methods by gathering data from multiple sources and applying grounded theory (GT) techniques to extract and identify common practices.

In order to concretize the research objectives, we ask the following research questions: *RQ1)* What are typical stages and activities in a DevOps approach for blockchain-based applications? *RQ2)* What are the particularities of using DevOps in blockchain-based software development?

For illustration purposes, this paper describes DevOps in the context of Ethereum, a popular smart contract platform, and Solidity, the platform's leading programming language for smart contracts. However, it can be assumed that the presented concepts and basic practices are in principle transferable to other platforms as well.

The paper is structured as follows: First, we discuss related work in Section II and our research methodology in Section III. Then, we elaborate DevOps for blockchain-based solutions as main contribution in Section IV. Finally, we discuss findings in Section V and conclusions in Section VI.

II. Related Work

According to our research, there is currently no academic literature that specifically addresses DevOps in the context of blockchain-based software development. There are some works that deal with (different types of) testing such software that can be considered as extended literature (see [2] and referenced literature therein). That aside, here are some papers that at least decidedly mention DevOps in the context of blockchains and smart contracts. Koul [3] discusses challenges faced in testing blockchain-based applications. The paper describes different approaches to testing and acknowledges the need to devise specialized tools and techniques for this purpose to ensure quality standards. Continuous testing in the course of DevOps is also mentioned, but not described in more detail. Li *et al.* [4] examine the challenges of developing and operating consortium blockchain solutions. Within their work, they discuss eight pairs of challenges and solutions for different phases of developing and operating such systems. One of the implications identified in their study is that applying DevOps culture and practices can be beneficial to overcome several challenges. Unfortunately there are no details on how to practically address this. Yussupov *et al.* [5]

analyze how blockchain technology and smart contracts fit into the serverless architectural style of developing cloud-native applications. The authors picture and derive a set of scenarios in which blockchains act as different component types in serverless architectures. Moreover, implementation requirements that have to be fulfilled to successfully use blockchains and smart contracts in these scenarios are formulated. In the course of this, DevOps requirements are also discussed, more specifically under the aspects to support the development of smart contracts and deployment automation, but not in sufficient detail. Other work in the broader context can also be cited that uses blockchain technology to improve DevOps and software development processes, particularly with respect to integrity and auditability. These include papers by Yilmaz *et al.* [6] to enhance development through a distributed record of software development events and Beller and Hejderup [7] to address trust issues through democratized build services or package repositories.

To our best knowledge, no academic work exists to date that addresses DevOps with a focus on blockchain-based software engineering. It is the goal of our work to make a first contribution in this respect in order to remedy this lack.

III. Research Study Design

Given the fact that our research objective is strongly linked to field applications of blockchain and that practical knowledge is often conveyed in practitioner reports, we decided to conduct a research methodology that is guided by the pattern derivation approach [8], where we define a pattern as the conceptual equivalent of (best) practices. In accordance with this scheme, we applied Grounded Theory (GT) techniques [9] [10] for theory building where patterns are discovered ("mined") and codified ("written"). Driven by our research questions and known practices from our own experience, we searched the major search engines (e.g., Google, Bing) for the following search string ("Blockchain" OR "Smart Contracts") AND (CI/CD OR "Continuous Integration" OR "Continuous Delivery" OR "Continuous Deployment" OR "DevOps" OR IAC OR "Infrastructure as Code") in order to gather a number of technically in-depth sources from the so-called "gray" literature [11] (e.g., practitioner reports/blogs). In addition, we searched GitHub for typical CI/CD configuration files (e.g., .travis.yml, .gitlab-ci.yml) which contain smart contract development frameworks (e.g., Truffle, Hardhat [formerly Buidler]) to study their configuration. The resulting source pool [12] was then analyzed using GT techniques. This included a close examination and labeling of materials with labels ("codes") and optional memos explaining important aspects of the findings while establishing conceptual relationships among the codes ("axial coding") to identify candidate patterns. In this process, pattern discovery and validation occurred stepwise in several iterative phases, using new sources (inspired by previous iterations) to constantly compare, revise, and contrast patterns. The primary stopping criterion, as is common in GT-based studies, was theoretical saturation, i.e., a state in which adding new sources no longer yields new insights.

IV. DevOps for Blockchain Smart Contracts

The core DevOps concepts and activities in the blockchain domain may not be very different from traditional software development. Developers work in a local branch on the source code for smart contracts and dependent applications, add new features or apply corrections to that code, test those changes, and submit their work to a source control management system from which a solution can be build. A release pipeline then deploys the smart contracts or dependent applications to one or more system environments. However, some inherent blockchain peculiarities cause specific constraints that need to be considered when adopting DevOps principles. In the following, we look at core aspects of DevOps for smart contracts and blockchain-based solutions. Since it is useful to divide CI/CD processes into stages, the content on these topics has been organized accordingly by key stages.

A. Continuous Integration

CI is a DevOps practice for automating the integration of code changes made by multiple developers. More specifically, it allows developers to frequently integrate changes into a central repository, where each integration is validated by an automated build, including tests, to catch integration errors. The main goals of CI are to optimize software quality by detecting and fixing bugs faster, and to minimize the time needed to validate and deploy software updates. The general CI flow is as follows: Developers have a local copy of the code on which they make changes and run local tests, once tests are successful they commit their changes and then submit a merge request. This request to merge code changes into a shared code repository is then reviewed through an approval process and depends on the success of a series of automated tests included in the build pipeline. Whereby that pipeline is typically triggered on every merge request that targets the main branch as well as whenever a commit is pushed to that branch.

These basic principles and action steps can also be applied to blockchain-based development. In our research, we found that established CI solutions (e.g., Jenkis, Travis CI, Gitlab CI/CD, GitHub Actions) provide sufficient means to build and test smart contracts and are also practically used for these purposes. These integrations usually consist of a dedicated CI environment (CI server) that monitors a code repository and performs automated actions in a (dockerized) shell environment when changes occur to check the state of that code along with the change that occurred. In general, there are a number of frameworks for smart contract development that support the management and automation of recurring development tasks. As such, these frameworks are also essential in a CI approach. Corresponding tools (e.g., Truffle, Hardhat, Embark, Brownie, Waffle) aim to provide a comprehensive development solution with an integrated testing blockchain to facilitate compiling, deploying, testing, and debugging smart contracts. These tools are usually available as CLI, either as pre-built Docker images, or they can be easily installed and run in a (dockerized) shell environment. In this manifestation, they can also be easily applied in a CI pipeline during various processing steps.

1) Code: The code phase focuses on core development tasks within IDEs supported by appropriate plugins and frameworks. Solidity code is typically written either in the web-based Remix IDE with integrated compiler and Solidity runtime environment or locally in a code editor of choice. In the Remix IDE, plugins can perform a variety of tasks such as verifying contracts, linting, generating documentation, compiling, debugging, deploying, and much more to support a rapid development cycle. When developing in a local IDE supported by a version control system (VCS), such focused integrated IDE support and abundance of development tools as in Remix is not yet present. As a way out, there is currently either the possibility to integrate the local file system into Remix, or conversely, there is the embryonic option to integrate Remix plugins into a local IDE (e.g., remix-vscode for Visual Studio Code).

The general design principle in software engineering of reducing complexity is especially true in the design of smart contracts. Smart contracts should be focused on a single task or capability (preferring many simpler smart contracts over a few larger ones) and be designed to minimize the number/size of on-chain transactions/writes (to reduce costs) as well as the dependencies required for testing. For general concerns (e.g., access control), production-tested library contracts (e.g., OpenZeppelin) and standardized contract implementations (e.g., ERC-20) should be used. Furthermore, contracts should be developed and tested by locking pragmas with a fixed compiler version to avoid the impact and risk of undiscovered bugs in newer compiler versions. In addition all public contract interfaces should be fully annotated with specially tagged comments in the so-called NatSpec format to provide documentation for functions, return variables, etc.

2) Build: The build phase includes all the steps required to generate the artifacts needed for execution from the source code. Regarding Solidity, this is the compiled bytecode for the Ethereum Virtual Machine (EVM) and the associated Application Binary Interface (ABI) as the interface required to interact with the EVM bytecode. To generate these artifacts there are two compilers, solc, written in C++, and solc-js, which uses Emscripten to cross-compile from solc C++ source code to JavaScript, thus both use the same compiler source code. The recommended way to interface with the Solidity compiler especially for more complex and automated setups is the so-called JSON-input-output interface. The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output. The compilers can be used either directly or via development frameworks mentioned earlier, which allow easier handling of compiler versions and compiler configuration. In the latter option, the compilation output format may vary depending on the framework used, but is usually represented as a JSON bundle containing information related to compiler input/settings (e.g. compiler name/version) and output (e.g. bytecode, ABI, SourceMap, etc.).

In some cases it may be necessary to use a preprocessor (e.g., solpp) before compilation, e.g. to reduce the source files by merging referenced imports from the file system, Node.js modules or URLs and their dependencies into a single file. Another reason would be, if the source files contain symbols or macros that should be extended, or if they contain proprietary operations that are only useful during development (e.g. the *console.log()* command from Hardhat) and should be removed. For CI, it is best to keep raw source files in a separate directory and run the preprocessor to output the code to the pipeline's source directory before compiling a project.

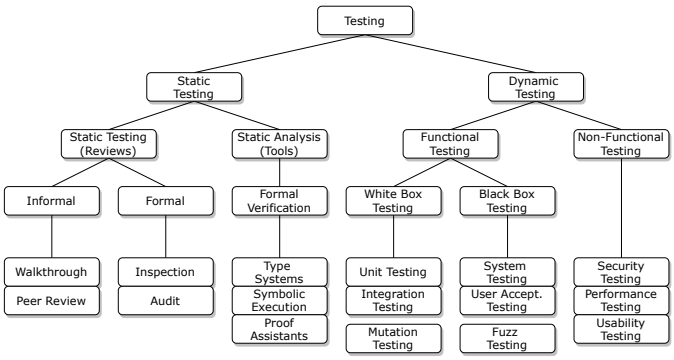

Fig. 1. Overview of test types for smart contracts and blockchain-based software.

3) Test: Once a build is successful, it is automatically deployed for review in a test environment where a series of automated tests are run. There are numerous ways to perform tests, and this also applies to blockchain-based software. A basic division according to separable components of the software to be tested and the test purpose is useful. For components, a subdivision according to architecture layers (i.e., application, smart contract, data, consensus, network) is suitable [2]. Regarding test purposes, these can be diverse and categorized in different ways, e.g., by the type of execution or focus. In terms of test complexity, a chronological order of unit, integration, system, and acceptance testing is generally used. Figure 1 provides an overview and aid to orientation with respect to possible test types. A detailed discussion of all presented test types is beyond the scope of this paper. Therefore, we mainly focus on smart contract testing with currently established methods and practical tools considered useful in the context of CI.

a) Testing Environment: For testing, smart contracts need to be deployed in a blockchain environment. This environment can be either an existing permanent or a purpose-built ephemeral environment that is specifically provisioned. In order to achieve reproducible results and avoid undue delays, the latter option is usually resorted to by utilizing a (temporary) local (in-memory) blockchain for testing and development purposes (e.g., Ganache) that simulates the characteristics of a real blockchain network. Unlocked and funded accounts are provided and new transactions are mined instantly, making automated tests much faster and cheaper to run. It is also easier to manipulate the blockchain environment, such as changing the gas price, mining speed, and time flow in general.

b) Test Data: Generally speaking, there are three ways to equip a blockchain for testing. In the simplest case, one

uses an empty blockchain without any transaction history. This approach is a viable option before initial deployment and is suitable for local testing of transaction history-independent logic and is usually also the starting point for many test cases. Another possibility is to fill a blockchain synthetically. Transactions are grouped within blocks, so prepopulating a blockchain could be done with custom tooling that forms valid blocks, though it requires careful coordination and sequencing of transactions also in the context of multiple parties. A more practical option in this context, as is common when setting up tests, is to run specific sequences of transactions and save the resulting state using a snapshot. This approach can be promising in complicated test cases to speed up tests by setting up so-called test-fixtures, i.e. consistent test environments with all preconditions a system shall have. With this approach, a snapshot of the blockchain state at the current block is saved and the blockchain can be restored to this state again and again, which simplifies automated tests. The last way to equip a blockchain for testing is to fork a production blockchain. With this approach, one can simulate the same state as the production blockchain within a local development blockchain. This is achieved by forking the production blockchain from a node endpoint at a given block into the local chain. Tasks related to new blocks are processed by the local chain, while tasks for older blocks are processed by the forked chain.

c) Unit and Integration Tests: Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended [13]. According to a survey of Chakraborty *et al.* [14] regarding blockchain software, the most common technique to check correctness is unit tests followed by manual code reviews.

In the context of smart contracts, unit tests should ideally cover all contract methods, or at least those that are publicly exposed (i.e., public, external). As part of this, it should be ensured that method return values are as expected and invalid input parameters are rejected. Further, expected execution of reverts and event emitting should be checked, which is a bit more complex as this requires processing of transaction receipts/logs, but there are tools to help with this (e.g. truffle-assertions, OpenZeppelin Test Helpers). Since most smart contracts introduce some form of role-based access control, access privileges should also be verified. With respect to the interdependence of test cases, each test case should be executable in isolation without relying on the state imposed by other test cases. Although it is possible to reduce test execution time by writing cascading test cases, this should be avoided to clearly communicate the intent of test cases (to others) and avoid dependency on the execution of other test cases to minimize complexity.

Integration tests as the next level, are more complex than unit tests, as the behavior of different parts as a whole is tested. For smart contract testing, this can mean interactions and complex scenarios with multiple calls between different dependencies (i.e., users/contracts) of a single contract or across multiple contracts, as well as on all types of oracles and front-end client applications. Subsequently, one can assume two different areas for integration testing. One refers to inter-blockchain interaction between cooperating smart contracts, the other to interaction between smart contracts and dependent client applications. The first aspect can be covered with blockchain development frameworks, for the latter other tools for integration testing might be more efficient.

When it comes to writing automated tests for Ethereum, developers have basically two main options: Solidity and JavaScript/Typescript. Solidity tests can basically test every single function in a contract in a bare-to-the-metal style, as the tests are written in the language of the components under test. When writing unit tests in Solidity it is necessary to create mock dependencies for oracles or dependent contracts. This approach can be cumbersome for several reasons: Another contract is created for each individual mock, thus the test setup is more complex and slower as multiple contracts need to be deployed and put into a specific state for each test. Further, test flexibility is limited to a predefined mock functionality. JavaScript tests, unlike Solidity tests, test contract behavior from an external client viewpoint (using contract abstractions and web3) and therefore can cover external and public, but not internal or private Solidity functions. Under the hood, JavaScript tests usually rely on established testing utilities such as the Mocha testing framework paired with Chai as an assertion library to test smart contracts asynchronously. This usually makes tests easier to implement and setting up a desired contract state less tedious. Some frameworks have also addressed the cumbersome mock situation of Solidity tests, and there are efforts to create mocks dynamically within the test code (e.g., Waffle, MockContract). Some frameworks also allow to execute tests in parallel (e.g., OpenZeppelin, Truffle), to speed up testing, if the tests are split across multiple files. Altogether, to make an analogy to the distinction between Solidity and JavaScript tests, the testing of an API can be used, whereby the logic implementation can be tested externally (including the transmission path) or directly internally. Figure 2 illustrates this aspect and the structure for Solidity- and JavaScript-based testing.

d) Static/Dynamic Analysis: While unit and integration tests verify that smart contracts behave as desired according to implemented test cases, they do not uncover potential vulnerabilities in the code itself. For this purpose, it is common to perform static analysis checks on smart contracts. Static code analysis is a debugging technique that examines code with heuristics without actually executing it. A linter can be understood as most basic form of static analysis and can help to improve the code quality and remove minor issues by e.g., checking syntax errors, structural problems, conformance against best practices, and code style guideline violations. A linter tool is typically one of the first applied measures to verify smart contracts. As a best practice, execution before each commit is a good idea, which can be realized with VCS hooks (pre-commit hooks for git). However, there are also more advanced tools beyond a linter that extend on static analysis and are commonly used to detect security vulnerabili-

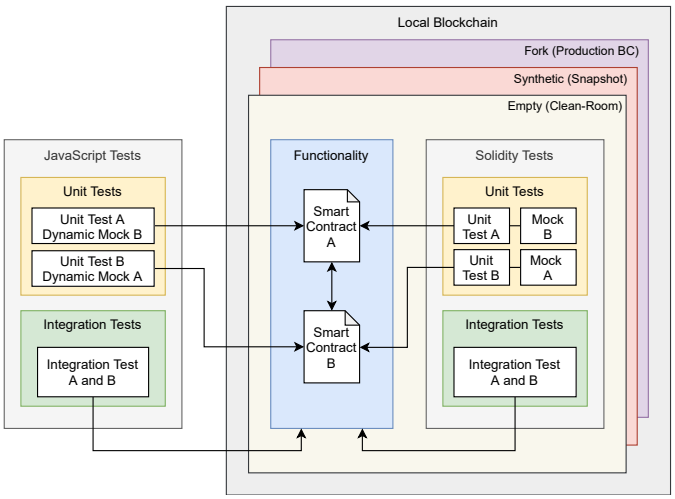

Fig. 2. Unit and integration test structure for Solidity and JavaScript tests.

ties. These use source code or generated bytecode to examine potential code behavior, vulnerable patterns, and errors that may occur during a program's runtime. Today, many such tools exist (e.g., MythX, Securify, SmartCheck, Slither, Manticore, Mythril; see [15] [16]) some of which incorporate a suite of vulnerability detectors that build on analysis techniques such as dynamic analysis, symbolic execution, SMT solving, to name a few. According to a survey by Ayman *et al.* [17] on the frequency of mentioning such tools on Medium and Stack Exchange, Mythril was mentioned most often.

In addition to these tools, there are also special test tools for smart contracts that rely on techniques already used in other software areas. One example is fuzz testing (e.g., Echidna), an automated testing technique in which software is fed invalid, unexpected, or random data as input and monitored for vulnerable program states and exceptions (e.g., crashes, memory leaks). Another example is (code) mutation testing (e.g., Vertigo), where certain components in a source code are intentionally changed to cause errors and verify that a test suite is able to detect the changes. This technique can be used to evaluate the quality of existing tests and to develop new ones.

Overall, analysis tools have different capabilities and detect different types of problems, but they are not perfect, so one has to expect false positives and false negatives. In this regard, a best practice is a combination of different tools to have a better protection and safeguard against potential problems.

e) Reports: An important metric when running tests is test coverage, which is a measure (usually given in percentage) that describes the extent to which the source code of a program is executed when a particular test suite runs. The idea is that tests should execute all code paths of the code under test. If this premise is (largely) fulfilled and the test results are as expected, the code is less likely to contain unforeseen errors. Code coverage tools also exist for Solidity (e.g., solidity-coverage, sol-coverage). Since coverage generation tracks which lines are hit during test execution by instrumenting contracts with specific Solidity statements and detecting their execution in a coverage-enabled EVM, coverage detection distorts gas consumption and slows testing. Thus, it is best practice to run coverage as a separate CI job rather than assume its equivalence to an ordinary test procedure. When run in a CI system, test coverage can be generated when developers push commits or merge branches.

Another important type of metric is tracking gas consumption. It can be useful to track gas usage per unit test and analyze gas metrics for method calls and deployments. In this context, a gas reporter tool (e.g., eth-gas-reporter) can help to get an overview of the gas costs associated with a smart contract. In a CI environment, the automatic generation of gas reports can be useful to show differences in gas consumption between code iterations.

B. Continuous Delivery

CD is a DevOps practice where software is built in such a way that it can be released to production at any time. For non-blockchain solutions, deployment pipelines deliver updated configurations and code to hosting environments, e.g., as a virtual machine, container, or a serverless function, or provision these environments with an infrastructure as code (IaC) approach. The core concepts are the same for blockchain solutions. A proper release pipeline would deploy needed environments and the smart contracts along with dependent applications to one or more system environments.

1) Release: The release phase is the point at which a build is ready for deployment to the production environment. At this stage, every code change has gone through a series of manual and automated tests, so it can be assumed that problems and regressions are unlikely. Since deploying smart contracts is a rather infrequent and delicate undertaking, it is desirable to have control over when builds are released to production. To this end, a manual approval process can be implemented in the release phase that allows only certain individuals within an organization to authorize a release for production. Before doing so, however, it may be advisable to take further precautions to minimize the risk of undetected issues prior to deployment. This includes conducting independent smart contracts security audits, preferably at least two from different organizations (e.g., Consensys, OpenZeppelin), which is especially important for safety-critical areas that manage large amounts of capital. Another advisable aspect in the release phase is to collect all artifacts generated for the deployment and store them in a shared environment to ensure that collaborating parties are fully aligned. Information such as metadata, ABI, bytecode, etc. can be stored centrally for each release. Based on this central archiving it is possible to provide useful services, e.g. an API for client applications to retrieve the version specific ABI for a smart contract.

2) Deploy: The deploy phase handles the process of pushing release builds into a production environment. There are several measures to automate the process to make release procedures more reliable and less cumbersome.

a) IaC: IaC automates the deployment of system environments to achieve consistency of components, topology, and

configuration in order to mitigate discrepancies that can result from the direct application of manual changes to a system. This approach is particularly beneficial for the provisioning process of permissioned blockchains, which is usually complicated and should therefore be automated to avoid errors and further save time and resources. There are some IaC utilities on the market to automate the provisioning process (e.g., Terraform, Ansible, Puppet, Chef), which can also be configured appropriately for this purpose, but we have hardly come across this approach in our research. Blockchain-as-a-Service (BaaS) offerings to ease provisioning are far more common, but are subject to vendor-specific limitations in terms of supported blockchain platforms and hardware infrastructure, furthermore dovetailing with DevOps is more difficult.

b) Smart Contract Deployment: When deploying smart contracts to test and production networks, automated solutions are needed to ensure proper deployment in the respective environments. Contracts need to be initialized in a certain order with certain parameters and possibly put to a certain state by calling functions (e.g., to set permissions), depending on the respective deployment target. Specifically, this means that in order to deploy a smart contract, required libraries and dependent contracts must first be deployed. To achieve flexibility with respect to different environments, especially for testing, dependency information is typically passed into the contract via the constructor and is not hard-coded in the contract. One can imagine that performing the described deployment steps manually for multiple environments, is not only more time consuming, but also more error prone. Fortunately, suitable tools (e.g., Truffle Migrations, Hardhat Ignition) can be used to reliably automate the necessary steps for linking contracts to other contracts and populating contracts with initial data.

c) Upgradeable Smart Contracts: The only way to upgrade a contract is to deploy a new version of that contract. This procedure requires manually migrating all state information of the old contract and propagating the new contract address to users. To avoid this, there are upgrade mechanisms that can be used to replace contract implementations while preserving their address, state, and balance. Most commonly, a proxy pattern (see Contract Relay in [18]) is used for this purpose in combination with the *delegatecall* mechanism, which allows a function from another contract to be executed as if the function were from the calling contract. Based on these concepts, it is possible to develop a solution where users interact directly with a proxy that is responsible for handling state information and delegating transactions (via *delegatecall*) to and from other exchangeable (upgradeable) contracts that contain the associated logic. To avoid requiring the proxy to expose the entire interface of logic contracts, which would be difficult to maintain and make the interface itself not upgradeable, a dynamic forwarding mechanism can be used. In this case, the proxy can forward any call of any function with any set of parameters (with the *fallback* function) to the logic contract; depending on the caller address calls to manage the proxy can also be handled directly. One drawback of the proxy approach is that the proxy contract and its delegate/logic contracts use the same storage layout. Therefore, when handling state variables, care must be taken to avoid scoping and storage collisions between the proxy and logic contracts or between different versions of the latter. In this context, three patterns are helpful: Inherited, Eternal, and Unstructured Storage (see [19], [20] for more details).

After all, smart contracts can be updated under additional effort and increased complexity. To support developers in this regard, efforts are being made to develop and establish standards and frameworks based on the concepts described above. For example, the EIP-2535 Ethereum Improvement Proposal, titled "Diamonds, Multi-Facet Proxy", formulates a standard for building modular smart contract systems that can be extended in production. Another example is OpenZeppelin's Upgrades Plugins, which can be integrated into existing development environments and workflows to support the deployment and management of upgradeable smart contracts.

d) Testnet: It is common practice to test contracts on a public test network (aka testnet) before deployment on the mainnet. In this context, the organization of releases in stages (alpha, beta) through testnet and mainnet and the tendering of bug bounties should be considered. There are several public testnets for Ethereum, which differ mainly in the consensus algorithm, block time, and supported clients. The main testnets are Goerli, a cross-client Proof-of-Authority (PoA) testnet with a block time of 15 seconds, Rinkeby and Kovan also PoA based testnets with fewer client support and a block time of 15 respectively 4 seconds, and Ropsten, the most similar testnet to the Ethereum mainnet with a Proof-of-Work (PoW) consensus and a block time of under 30 seconds. In general, it is advantageous to test the behavior of smart contracts first with a PoA and later with a PoW test network, as the former are usually more stable and the latter can have unpredictable block times and frequent chain reorganizations.

3) Operate: When a build is deployed to production, it is important to make sure that everything is running as intended. In this context, when deploying smart contracts on a permissionless blockchain, it may be necessary to publicly verify the deployed contract to establish trust with others. This involves using a recognized service (e.g., Etherscan, Sourcify) to confirm that an uploaded and publicly viewable source code is the same as the code compiled on the blockchain. This creates transparency as users know exactly what is being deployed on the blockchain, and allows the public to audit and verify the code to ensure it is actually doing what it is supposed to do. This task can be automated as part of CD with the right tooling (e.g., truffle-plugin-verify, hardhat-etherscan).

4) Monitor: To ensure the health, performance, and reliability of smart contracts and dependent applications, it is necessary to monitor their operation. Monitoring and analyzing the behavior of smart contracts can be done based on various metrics or events to detect erroneous or suspicious behavior. For simple checking purposes, one can use a block explorer (e.g., Etherscan, Etherchain, Blockchair; see [21] for more details), which acts as an analytics platform or search engine that allows users to look up real-time data on

blocks, transactions, miners, accounts, balances, and other on-chain activities for both the main Ethereum network and the testnets. In addition to these closed services, which cannot be independently verified, there is also the possibility of operating one's own blockchain explorer (e.g. BlockScout, Expedition), for which implementations for private EVM-based networks also exist (e.g. Ethernal). In addition, it is in principle possible to run a dedicated blockchain node and implement a smart contract activity tracker that sends JSON RPC requests to that node to request transactions, blocks, and logs and scan these for specific characteristics. Following this principle, there are also service providers (e.g. Tenderly, OpenZeppelin Sentinel, Parsiq) that do the heavy lifting of such a solution and offer a convenient setup of complex events to be monitored. This makes it possible to easily monitor e.g. state variables, function calls, function arguments, function reverts, emitted events, gas consumption deviations and send alarms in case of critical behavior via different means (e.g., Webhooks).

In the event that own infrastructure is deployed, it should be monitored as well. There are several monitoring solutions (e.g., Prometheus, cAdvisor) that make it possible to observe the activity of network nodes by collecting statistics that can be used to analyze and optimize resource usage and for a better overall understanding of system operation.

C. CI/CD Stages Overview

To summarize the solution aspects and approaches presented so far in a simple visual manner, Figure 3 shows a representation of typical DevOps stages and associated activities that can be incorporated in a CI/CD pipeline. While there are many degrees of freedom in setting up these pipelines, such rough guidance can help in initial setups. A rough sequence for a smart contract project to be built, tested, and deployed might look like the following: First, the sources are preprocessed and compiled, then static/dynamic vulnerability testing and unitary/integration testing is performed, thereafter reports and a release are generated, and finally deployment to the staging/production environment is done followed by verification. To practically illustrate the process, we set up a smart contract sample project [22] based on the Hardhat development framework and various helpful development tools, and implemented a GitLab CI/CD pipeline to demonstrate a holistic DevOps approach. An interesting finding in the above setup is that while the order of DevOps stages is quite clear, the order of jobs within stages or the dependencies between different jobs is only to some extend pre-determined. For example, it is not always mandatory that the code is already compiled in order to perform static or dynamic code analysis jobs, since respective tools take care of the compilation or trigger it again in the course of their processing. In this case, care should be taken to ensure that the compiler versions used are consistent between different tasks/jobs. The fuzzy processing order can also be used to advantage, e.g. jobs can be executed in parallel, since the processing of upstream jobs is not always mandatory. Thus, the execution time of a pipeline can be significantly reduced.

V. Discussion

Developing smart contracts at scale is difficult, especially for a distributed team. Add in key management, various responsibilities, different testing strategies, varying computing and testing environments, etc., and this leads to disparate development experiences among developers. The remedy is a DevOps approach and an appropriate tool framework that allows teams to not only manage their development and deployment process, but also integrate threat analysis and release management, for example. While the core DevOps concepts and practices in this area are basically the same as for any other type of software project, there are some peculiarities due to the decentralized nature of blockchain and its immutability. These include a greater focus on testing, in particular the use of testing tools to detect vulnerabilities using static and dynamic code analysis. During our research we have repeatedly encountered the inclusion of various such tools. One can say that they are an essential part of testing smart contracts. Conventional (unit) testing certainly plays an important role as well, but is much more costly to implement in comparison. Here, the fact that functions of smart contracts can be called publicly means that testing can be compared with the testing of APIs. Therefore, when testing implementations, understanding interfaces and communication points is a key challenge to ensure consistency with defined processes and legacy code. In this sense, it can be said that UI/end-user based tests play a rather minor role, while integration tests (API) should make up the bulk of the tests.

Another peculiarity is the differentiated approach to otherwise standard software deployment practices, as smart contracts can be hardly updated. It is worth mentioning here that during our research we found that deployments to production systems are usually done manually, either by deploying locally from a developer's machine or by manually triggering a deployment job defined in a pipeline. When deployment is triggered manually in a DevOps pipeline, it is referred to as Continuous Delivery rather than Continuous Deployment, since the latter involves fully automated deployment of new releases to production. Continuous Deployment for smart contracts would also be conceivable in principle, also with regard to the aforementioned proposals for upgrade mechanisms, but is hardly an issue at present because deployment is currently a rather rare and delicate undertaking over which developers want to have tight control. Therefore, mechanisms associated with CD such as Canary Releases (gradual roll-out of releases to a subset of users), feature toggles (turning functionality on or off at runtime of the software), A/B testing (evaluation of two variants of a system), are currently also uncommon. It is likely that future improvements in the ecosystem and recent initiatives to build modular smart contract systems that can be extended in production will make updating smart contracts a matter of course, and thus Continuous Deployment will become more important alongside CI. This is also a logical step given necessary code changes and bug fixes that are an integral part of software development.

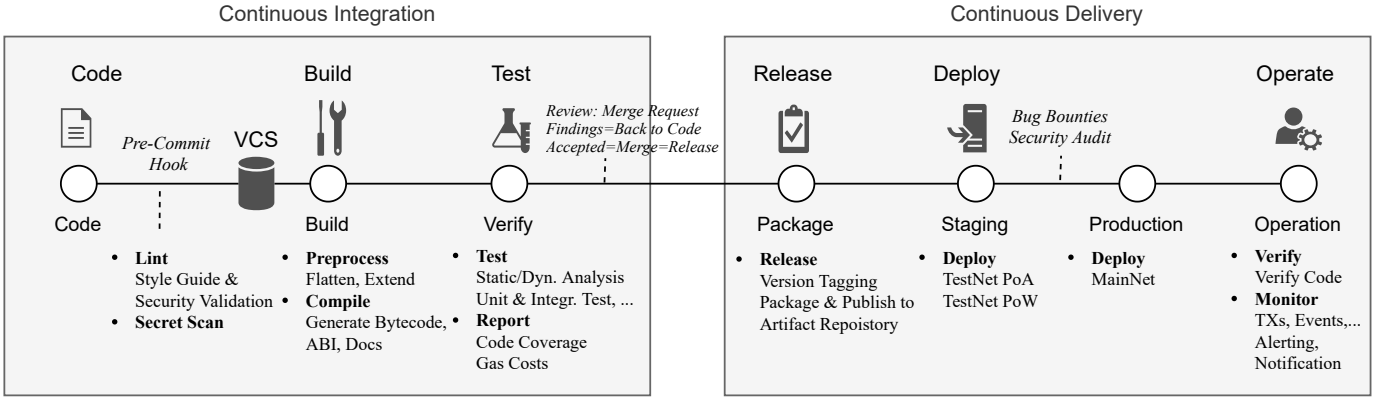

Fig. 3. An overview of DevOps stages for smart contracts.

VI. Conclusion

Over the past decade, DevOps principles have been applied to a wide range of software development industries and disciplines. Essentially, the same principles can be applied to the development and operation of blockchain-based applications. However, applying DevOps in this domain requires a structured approach and corresponding guidelines. To this end, we studied practitioner reports and existing solution approaches, which we analyzed with GT techniques, to infer typical DevOps practices. In the process, we elaborated procedural steps according to the main stages of Continuous Integration and Continuous Delivery. Based on our findings, we compiled a typical DevOps approach highlighting possible stages and associated activities.

The use of DevOps in the blockchain environment is quite possible today with already established CI/CD solutions and is also lived in practice to make the development process faster, more pleasant, and more controlled. The development tools for smart contracts are sufficiently mature, also with regard to process automation and aggressive testing. The challenge is rather to specify the necessary test and deployment requirements and to select and orchestrate the appropriate tools from a constantly changing set of available development utilities and frameworks.

As blockchain technology has grown, so have the requirements around building and testing related applications. In this regard, in addition to formulating best practices for development, future research can be devoted to devising testing strategies that meaningfully combine the variety of techniques and tools in the field to integrate with DevOps and address outstanding challenges in ensuring reliable blockchain-based applications.

References

[1] W. de Kort, "What Is DevOps?" *DevOps on the Microsoft Stack*, pp. 3–8, 2016.

[2] C. Lal and D. Marijan, "Blockchain Testing: Challenges, Techniques, and Research Directions," 2021.

[3] R. Koul, "Blockchain Oriented Software Testing - Challenges and Approaches," *2018 3rd Int. Conf. Converg. Technol. I2CT 2018*, pp. 1–6, 2018.

[4] S. Li, Q. Xu, P. Hou, *et al.*, "Exploring the Challenges of Developing and Operating Consortium Blockchains: A Case Study," *ACM Int. Conf. Proceeding Ser.*, pp. 398–404, 2020.

[5] V. Yussupov, G. Falazi, U. Breitenbücher, *et al.*, "On the serverless nature of blockchains and smart contracts," *arXiv*, 2020.

[6] M. Yilmaz, S. Tasel, E. Tuzun, *et al.*, *Applying Blockchain to Improve the Integrity of the Software Development Process*. Springer International Publishing, 2019, vol. 1060, pp. 260–271.

[7] M. Beller and J. Hejderup, "Blockchain-based software engineering," *Proc. - 2019 IEEE/ACM 41st Int. Conf. Softw. Eng. New Ideas Emerg. Results, ICSE-NIER 2019*, pp. 53–56, 2019.

[8] D. Riehle, N. Harutyunyan, and A. Barcomb, "Pattern Discovery and Validation Using Scientific Research Methods," no. February, 2020.

[9] B. G. Glaser and A. L. Strauss, *Discovery of grounded theory: Strategies for qualitative research*. 2017.

[10] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qual. Sociol.*, 1990.

[11] V. Garousi, M. Felderer, M. V. Mäntylä, *et al.*, *Benefitting from the grey literature in software engineering research*, 2019.

[12] Maxwoe/sc-devops-knowledge-sources, [Online]. Available: https://github.com/maxwoe/sc-devops-knowledge-sources (visited on 09/02/2021).

[13] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development*, 2004.

[14] P. Chakraborty, R. Shahriyar, A. Iqbal, *et al.*, "Understanding the software development practices of blockchain projects: A survey," *Int. Symp. Empir. Softw. Eng. Meas.*, 2018.

[15] M. Di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," *Proc. - 2019 IEEE Int. Conf. Decentralized Appl. Infrastructures, DAPPCON 2019*, pp. 69–78, 2019.

[16] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart Contract: Attacks and Protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[17] A. Ayman, S. Roy, A. Alipour, *et al.*, "Smart contract development from the perspective of developers: Topics and issues discussed on social media," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 12063 LNCS, pp. 405–422, 2020.

[18] M. Wöhrer and U. Zdun, "Design Patterns for Smart Contracts in the Ethereum Ecosystem," in *2018 IEEE Int. Conf. Internet Things*, 2018, pp. 1513–1520.

[19] Proxy Patterns - OpenZeppelin blog, [Online]. Available: https://blog.openzeppelin.com/proxy-patterns/ (visited on 09/01/2021).

[20] P. Klinger, L. Nguyen, and F. Bodendorf, *Upgradeability Concept for Collaborative Blockchain-Based Business Process Execution Framework*. Springer International Publishing, 2020, vol. 12404 LNCS, pp. 127–141.

[21] G. A. Pierro, R. Tonelli, and M. Marchesi, "An organized repository of ethereum smart contracts' source codes and metrics," *Futur. Internet*, vol. 12, no. 11, pp. 1–15, 2020.

[22] Maxwoe/sc-devops, [Online]. Available: https://github.com/maxwoe/sc-devops (visited on 08/30/2021).