Blockchain-based Collaborative Development of Application Deployment Models

Ghareeb Falazi, Uwe Breitenbücher, Michael Falkenthal, Lukas Harzenetter, Frank Leymann, Vladimir Yussupov

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{falazi, breitenbuecher, falkenthal, harzenetter, leymann, yussupov}@iaas.uni-stuttgart.de

BIBTEX:

```
@InProceedings{Falazi2018_BlockchainCollaborativeDevelopment,
  author    = {Falazi, Ghareeb and Breitenb{\"u}cher, Uwe and Falkenthal, Michael
               and Harzenetter, Lukas and Leymann, Frank and Yussupov, Vladimir},
  title     = {{Blockchain-based Collaborative Development of Application
                Deployment Models}},
  booktitle = {On the Move to Meaningful Internet Systems.
               OTM 2018 Conferences (CoopIS 2018)},
  publisher = {Springer},
  series    = {Lecture Notes in Computer Science},
  volume    = {11229},
  pages     = {40--60},
  year      = {2018},
  isbn      = {978-3-030-02610-3},
  doi       = {10.1007/978-3-030-02610-3_3}
}
```

Blockchain-based Collaborative Development of Application Deployment Models

Ghareeb Falazi, Uwe Breitenbücher, Michael Falkenthal, Lukas Harzenetter, Frank Leymann, and Vladimir Yussupov

Institute of Architecture of Application Systems, University of Stuttgart, Germany
{`lastname`}`@iaas.uni-stuttgart.de`

Abstract. The automation of application deployment is vital today as the manual alternative is too slow and error-prone. For this reason, many technologies for deploying applications automatically based on deployment models have been developed. However, in many scenarios, these models have to be created in collaborative processes involving multiple participants that belong to independent organizations. However, the potential competing interests of these organizations hinder the degree of trust they have in each other. Thus, without a guarantee of accountability, iterative collaborative deployment modeling is not possible in such domains. In this paper, we propose a decentralized deployment modeling approach that achieves accountability by utilizing public blockchains and decentralized storage systems to store intermediate states of the collaborative deployment model. The approach guarantees integrity of deployment models and allows obtaining the history of changes they went through while ensuring participants' authenticity.

Keywords: Declarative deployment models, blockchains, accountability

1 Introduction

The automation of application deployment is vital in many modern scenarios because manual deployment is typically too slow and error-prone—especially in cloud-based environments that are completely designed for automation. For this reason, the idea of deployment models was introduced [16]. These models describe the application to be deployed which includes all the required components as well as their relationships. Furthermore, many deployment automation technologies, such as Chef [2] and Kubernetes [3], have been developed to automate the entire deployment process based on the aforementioned models.

However, the creation of such models is becoming more difficult as many scenarios nowadays require different companies and experts to participate in th process. For example, the creation of a deployment model for a data analytics application would likely involve multiple participants that belong to independent organizations. Moreover, such a collaborative development process is likely iterative which makes it fairly complex, and even worse, the various participants of such a distributed scenario may have competing interests in the market, with a low level of mutual trust. This raises the importance of enforcing accountability in the collaborative deployment modeling process, so that the changes made to the deployment model over the course of the creation process are immutably documented and linked with their authors. However, existing version control technologies such as Git and Subversion (SVN) are not enough to support the desired process as their main aim is aiding collaboration rather than accountability. For example, both approaches do not prevent that the history of log changes on the server are altered secretly by a malicious party, which destroys accountability.

In this paper, we tackle these issues by presenting a decentralized collaboration approach that allows developing deployment models in business-critical scenarios where accountability is of high importance. The approach we present in this paper is based on blockchains which we use as an immutable store for the various states a deployment model goes through. Participants use a modeling tool, like Winery [22], to enhance a shared deployment model, and before forwarding it to other participants, they register the version they have produced in the blockchain. Through the application of the approach, we achieve process integrity which means that we guarantee having an irrefutable proof of who is responsible for which changes. Furthermore, we also guarantee maintaining the provenance of the model and its sub-components in the form of an immutable history of changes that allows us to inspect how the model evolved, and who is responsible for potential malicious acts. We prove the technical feasibility of the approach based on a prototype integrated into Winery.

The remainder of this paper is structured as follows: in Sect. 2, we introduce the basics of deployment modeling, and motivate our work. In Sect. 3, we provide a formal description of the problem, whereas in Sect. 4 we present our blockchain-based approach. We prove the feasibility of the approach in Sect. 5, and evaluate the prototype in Sect. 6. Finally, we discuss the related work, and provide concluding remarks with a glance at the future work.

2 Motivation and Background

In this section, we provide some basic domain knowledge and emphasize the problem we aim to solve in our work, as well as framing its scope.

2.1 Deployment Automation Technologies and Models

Cloud computing environments nowadays support a high degree of service provisioning automation which allows customers to acquire cloud resources required for deploying their applications in a dynamic manner. Such automatic provisioning reduces the number of potential errors and accelerates the process. A provisioning engine is usually responsible for deployment automation, and supports describing the deployment configuration in the form of a *deployment model* [16]. A *declarative* deployment model describes the structure of the application to be deployed including all components it consists of as well as their configuration.

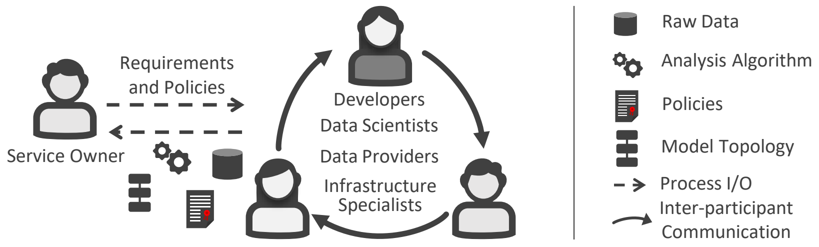

Fig. 1. Collaborative development of application deployment models.

For example, a declarative deployment model may specify that a certain java application has to be deployed on a Tomcat webserver listening on port 80, which shall be installed on a virtual machine running on Amazon EC. Thus, it describes only the desired goal, not how to execute the actual deployment technically. In contrast, *imperative* models explicitly describe *how* provisioning should take place in the form of a process describing each task to be executed. In this work, we focus on declarative deployment models, as they are less intrusive to developers, and because they are more widely accepted by most deployment automation technologies [11]. Developing such deployment models is often done by a single company as part of a regular DevOps process without external participants; however, in the following, we show how the development of specific kinds of applications requires collaborating with external parties.

2.2 Motivation: Business-Critical Deployments

The development of business-critical applications often requires the collaboration of multiple companies and experts. For example, a data-analytics application requires critical data, which a specialized data-owning company stores, to be accessed and processed by algorithms provided by data analysts. However, for such algorithm developments that require a significant level of expertise, often external companies are engaged [8,18], and the final results of the analysis could be consumed by a different company, such as an advertisement agency. If we want to deploy such an application to a cloud environment using the aforementioned declarative deployment models, we need a collaborative process in which each of the relevant companies participates by providing certain portions of the model.

Figure 1 shows an exemplary scenario to develop the deployment model of a cloud-based data-analytics application. In this scenario a company, which could be, e. g., an advertisement agency, is interested in the analysis of data from a certain domain and requests the creation of a deployment model for a cloud application that fulfills its needs. The company, which we denote as a *Service Owner*, outsources the development process and associates its request with the set of requirements and policies it wants to enforce. The creation of a deployment model for this application would likely involve multiple participants that belong to independent organizations: a (i) *Data Scientist* provides the respective algorithms, and indicates the requirements they have for execution, such as the appropriate

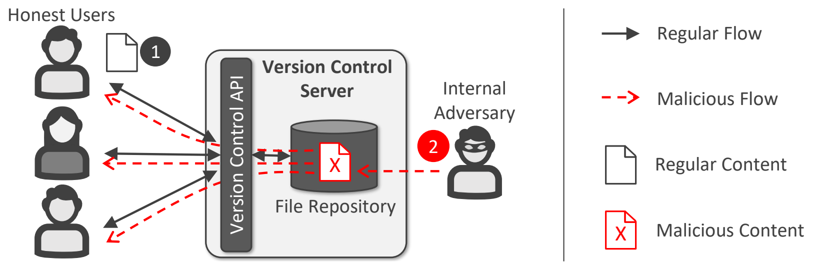

Fig. 2. Honest interaction with a centralized VCS (1) vs. malicious interaction performed by an internal adversary (2).

runtimes, the used libraries, etc. Furthermore, a (ii) *Data-owning Company* supplies the possibly business-critical data on which the algorithms operate, and specifies privacy and security requirements for the deployment, which could be, for example, that the deployed algorithms are only allowed to access specific databases of the company, and that these algorithms must not run on a shared IT resource. However, certain security requirements, e. g., that the deployed analysis algorithm must not access the Internet as otherwise business-critical data may leave the company without permission, have the potential of interfering with needs of the algorithm. Thus, a (iii) *Infrastructure Specialist* is required that refines the deployment model based on (1) the data scientists' technical requirements and the (2) data-owning company's security and privacy policies.

Moreover, such a collaborative development process can easily become iterative, e. g., if the final analysis results are not satisfying and changes need to be made to the provided algorithms by tuning their parameters, or to the set of accessible databases by allowing access to more of them. This makes the whole process fairly complex, and to make things worse, the various participants of such a distributed scenario may have competing interest in the market, and thus the degree of trust they have in each other is limited. For example, if the *Infrastructure Specialist* was malicious, they could alter the analysis algorithms to produce misleading results and then the blame could be falsely pointed at the *Data Scientist*, or they can alter the security policies provided by the *Data-owning Company* to allow illegal exportation of business-critical data.

This means that accountability plays a vital role in facilitating collaborative development of business-critical deployment models as it enables tracking the various changes made throughout the process and associating them with their authors. This allows the blame for malicious acts to be correctly directed.

2.3 Problem Statement: Collaborative Deployment Modeling

The absence of trust is considered one of the most important risk factors that can affect collaborative software development and lead it to failure [24]. Accountability, which we define as the ability to detect the actor(s) who maliciously alter the deployment model under development, can increase the level of trust in such processes as it allows implementing punishing measures to adversaries, which has the potential of reducing the probability of malicious acts in the first place.

However, existing collaboration approaches such as Version Control Systems (VCS) do not facilitate accountability. Centralized VCSs such as Github or SVN constitute a single-point of failure for collaborative processes. For example, a large DDoS attack stalled Github for five days in 2015 [20]. Furthermore, participants of collaborative processes need to trust the impartiality of parties operating these systems, and that they are well-protected against adversaries. Figure 2 shows that, although regular users interact with such centralized systems through their exposed APIs only, an adversary who has access to the internals of the server can directly alter the contents of the repository holding the shared files without having to use the API. These malicious edits would later propagate to the honest participants with the next pull from the server without them noticing the attack.

On the other hand, the Git protocol, which is the basis of Github and similar services, allows working in a peer-to-peer style without the need for a centralized origin server. However, this usage is discouraged as it increases the potential of confusing who is doing what, and it requires peers to be always online to provide access to their local repositories [15, p. 101]. Furthermore, both centralized and decentralized VCSs allow changing various aspects of the recorded history such as the author or the message of a commit [5, 29], and although Git, as an example, allows using cryptographic signatures to associate commits to their authors, it does not enforce it [15, pp. 233-237], and even if a set of rules that enforces signing all commits were introduced on top of Git, this would not prevent unilaterally removing commits completely from the history log by one or more partners which cannot be proven by others, especially if a trusted intermediary is not used.

In this paper, we propose a blockchain-based approach that enforces accountability in the collaborative development process of deployment models while avoiding the need for trust in third-parties. Here, we do not aim at replacing the role of VCSs, but finding a way to ensure accountability regardless of the underlying technologies used to exchange and version files. So our approach may be combined with VCSs in a complementary way. For example, each participating organization can have a local VCS to organize the local development process.

3 Assumptions and Formal Definitions

The sensitive nature of certain collaborative processes makes it crucial to have the ability of tracking changes of the evolving deployment model even when the level of trust among the participants performing these changes is low. In this regard, we can identify two general aspects of change traceability that should be considered in order to achieve the desired accountability: (i) the **integrity** of the collarborative deployment modeling process, and (ii) the **provenance** of all artifacts contained in the deployment model. Before defining these two terms, we explain the assumptions we build upon.

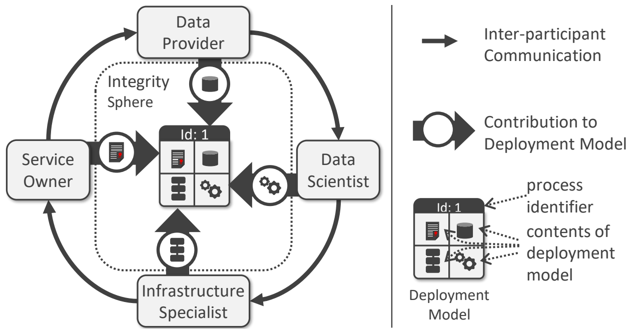
Data Provider
Integrity Sphere
Id: 1
Service Owner
Data Scientist
Infrastructure Specialist
Inter-participant Communication
Contribution to Deployment Model
Id: 1
process identifier
contents of deployment model
Deployment Model

Fig. 3. Maintaining accountability in the collaborative deployment model development.

3.1 Identity Establishment Assumptions

The basic assumption we have regarding the deployment model and its sub-resources, such as algorithm implementations, data files, policies, etc., is that they can be uniquely identified during the collaboration process. Figure 3 shows an exemplary process in which multiple participants operate on a deployment model depicted as a single entity consisting of sub-resources which get added or modified in an iterative manner. The entity, shown as a square containing multiple artifacts, is associated with an identity seen by all participants. To achieve this, we assume that the participant who initiated the development processes creates an initial version of the deployment model associated with a unique identifier. Then, all other participants operate on this initial model while always referring to it using the same identifier. We refer to the set of tasks that allow participants to have a common view on the deployment model as the *integrity sphere*. Finally, we assume that all involved participants can also be uniquely identified. Later in this paper, we demonstrate how our approach fulfills these assumptions.

3.2 Formal Problem Definition

Now, we formally describe a data structure that captures the development of a deployment model and show why all participants should have access to it in order to ensure integrity and provenance, the two pillars of accountability.

Assuming the following notations: (i) $\mathcal{M}$ refers to all potential deployment models, (ii) $\mathcal{I}_m$ refers to all potential identifiers of deployment models, (iii) $\mathcal{S}$ refers to all potential sub-resources, (iv) $\mathcal{I}_s$ refers to all potential identifiers of sub-resources, (v) $\mathcal{D}$ refers to all potential contents of sub-resources, and (vi) $\mathcal{P}$ refers to all potential participants of the process, we define the **state of the deployment model** at the point in time $t \in \mathcal{T}$ as a tuple $m_t = (author, S) \in \mathcal{M}$ where: (i) $author \in \mathcal{P}$ is the participant who authored this specific version, and (ii) $S \subseteq \mathcal{S}$ is the set of all sub-resources contained within the main resource at time t. Moreover, we define each **sub-resource** $s \in \mathcal{S}$ as a tuple $s = (id_s, data)$ where $id_s \in \mathcal{I}_s$ is the unique identifier of s, and $data \in \mathcal{D}$ is its actual content.

A sub-resource could be, e. g., an implementation of an algorithm, a web server configuration file, a license file, etc. Whereas the model is a single archive that encapsulates all of these artifacts. Furthermore, we define the **development of a deployment model** with an identifier $id_m \in \mathcal{I}_m$ as an acyclic digraph $G_{id_m} = (V, E)$ whose vertices $V = \{m_{start}, \ldots, m_{end}\} \subseteq \mathcal{M}$ represent the various states the model goes through, and whose edges $E \subseteq V \times V$ represent causal dependencies between states. The identifier of the model development graph, id_m, is set by the *Service Owner*, and distributed with the deployment model. Further participants have to use the same identifier to refer to the same process.

A new state is added to the graph when a participant $p \in \mathcal{P}$ at time t shares a new version of the deployment model with one or more other participants, thus, our notion of time is discrete, and new time points are added only when new states emerge in the model development graph. Our approach, as explained in Sect. 4.1, guarantees that no two distinct deployment model states can share the exact same point in time of creation. Furthermore, we define $\mathcal{A} : \mathcal{I}_m \to \wp(\mathcal{P}_{bc})$ as a mapping that, given a deployment model identifier, returns the set of participants authorized to take part in the process. For convenience and to enhance readability, we define the projection $\pi_e(tuple)$, which returns the element labeled e of a given $tuple := (.., e, ..)$. Based on these notations, we define the notion of integrity:

Definition 1 (Integrity of Collaboration Process). *Integrity is the fact that the author of any deployment model state of a given development graph* G_{id_m} *is an authorized participant:* $\forall m_t \in \pi_V(G_{id_m}) : \pi_{author}(m_t) \in \mathcal{A}(id_m)$ $\square$

This means that if the deployment model transitions to a new state which we cannot associate to any of the authorized participants of the process, we should consider this state as invalid, and refuse to operate on it. Furthermore, integrity guarantees the non-repudiation of participants' actions meaning that a participant causing the state of the deployment model to change will not be able to deny their responsibility for this action.

On the other hand, provenance refers to the ability to identify all states a sub-resource goes through, which includes knowing the set of participants who operated on it, and when state changes occurred:

Definition 2 (Provenance of a Sub-Resource). *Given a model development graph* G_{id_m}*, provenance is a mapping:* $p : \mathcal{S} \to \wp(\mathcal{T} \times \mathcal{P} \times \mathcal{D})$ *which is defined as:*

$$p(s) := \{(t, \pi_{author}(m_t), \pi_{data}(\sigma)) \in \mathcal{T} \times \mathcal{P} \times \mathcal{D} \mid m_t \in \pi_V(G_{id_m}) \wedge \sigma \in \pi_S(m_t) \wedge \pi_{id_s}(\sigma) = \pi_{id_s}(s)\} \quad \square$$

We notice that sub-resources are always transmitted as part of the deployment model, so their evolution is coupled with its evolution.

4 Blockchain-based Approach to Ensuring Accountability

As seen in the previous section, guaranteeing accountability requires having the development graph of the deployment model accessible to all participants so they can add to it when producing a new version, or read from it when validating the integrity of a given one, and even discover how it evolved over time. Our approach, which aims at storing and managing this graph so that accountability is achieved, works by combining three complementing aspects: (i) using blockchains to create an immutable trace of all state changes of the deployment model, (ii) storing the detailed differences between resource states in a decentralized storage system to facilitate a fine level of accountability, and finally (iii) building a tree-of-trust that allows authorizing participants and maintaining their identities. Before giving more details about the approach, we first summarize why we have chosen blockchains as the core technology to implement it.

4.1 Suitability of Public Blockchains

Maintaining the globally accessible model development graph is straight forward in a centralized setup. However, such an approach requires the involvement of a trusted third-party, which is sometimes an unrealistic assumption [32]. Consequently, we aim to find a decentralized approach that does not have such a requirement by delegating the responsibility of managing the model development graph to the set of network peers collectively. However, the task is more challenging in a decentralized setup as we do not trust any specific peer to behave according to the protocol we set, nor do we trust the transportation channels among peers. This problem is known as the Byzantine Fault Tolerance (BFT) problem [23], and public blockchains provide a probabilistic approach to solve it [25].

Blockchain systems are usually used in situations were several parties need to share a common state regardless of the fact that they do not trust each other nor do they trust a third-party. They started as the backbone of crypto-currencies such as Bitcoin [25], but later involved use-cases in other fields, such as finance [28], and supply chains [13]. Moreover, looking at the high-level properties of the public blockchains, we can judge their suitability to our use-case. Public blockchain consensus protocols, such as Proof-of-Work [25] and Proof-of-Stake [21], periodically produce blocks of transactions which are broadcast to the peers of the network. These transactions are ordered within the blocks, and each of them alters the world state represented by the blockchain. For example, in the case of Bitcoin, the state is the set of balances of all accounts, and transactions represent currency transfer operations. However, if we apply this concept to our use-case, the state would be the deployment model itself, and each transaction would represent the creation of a new version of it. The aforementioned consensus protocols also ensure that altering an existing portion of the blockchain is practically infeasible without controlling either a large portion of the network's computing power (in the case of Proof-of-Work), or a large portion of the network's stakes (in the cased of Proof-of-Stake). This ensures that the blockchain provides not only the current state of the shared resource, but also an immutable history of it.

This means that each participant will be able to locally construct the model development graph G_{id_m} by traversing the blockchain and applying all relevant transactions in order. What makes blockchains even more suitable for the use case at hand is that they use public-key cryptography to sign all submitted

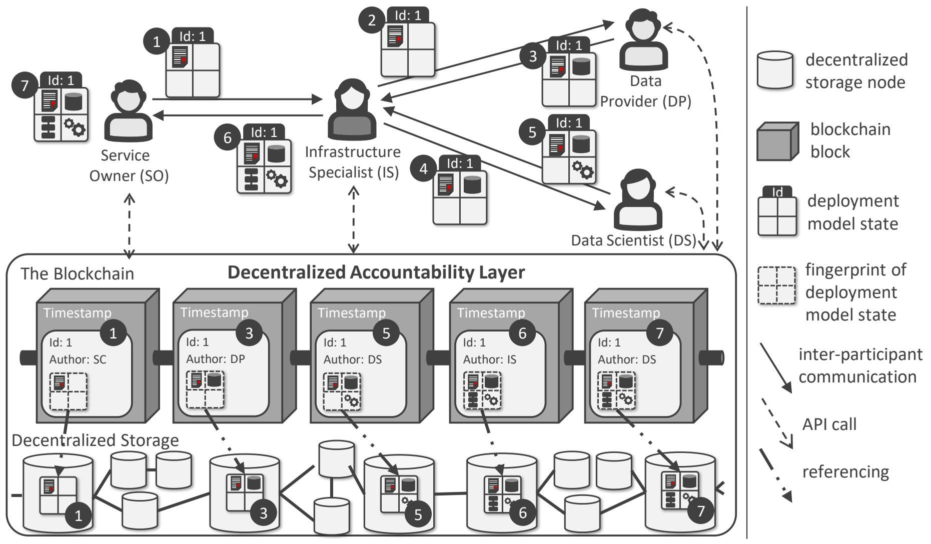

Fig. 4. Blockchain-based accountability approach applied to an exemplary use-case

transactions, which ensures their authenticity, i. e., that the author of each transaction is actually who they claim to be. Being a decentralized and an immutable ledger of discrete events that ensures authenticity of its records, makes the public blockchain suitable for our use-case.

4.2 Maintaining an Immutable History of State Changes

In this section, we explain the first angle of the approach. The basic idea here is demonstrated in Fig. 4 which shows its application to the motivational scenario we presented earlier in Sect. 2.2. The collaboration process starts when the *Service Owner* creates an initial version of the deployment model, which contains a set of policies, and gives it its unique identifier. Before forwarding the resource to the next participant (the *Infrastructure Specialist* in this scenario), the he registers it in the blockchain by issuing a transaction which contains enough information about this specific version of the model and the associated sub-resources to allow their unique identification. We call this information a fingerprint, and in Sect. 4.3 we explain it in more details. After making sure that the transaction containing the fingerprint is durably persisted in the blockchain, the *Service Owner* sends the model to the *Infrastructure Specialist*. It is irrelevant to our approach how the model is sent to the next participant as long as it was registered in the blockchain beforehand. When the model is received, the *Infrastructure Specialist* first checks the blockchain for stored versions of the same model and compares them to the received one. If the model is found to be registered beforehand in the blockchain, then its integrity is guaranteed; otherwise, the ownership of the received model cannot be proven, which means that it lacks integrity, and in such a case she should refuse working on it. We call this check integrity verification, and if it is successful, the *Infrastructure Specialist* can operate on the model. In this case, she needs the inputs of the *Data Provider*, and the *Data Scientist* before she can decide on the appropriate infrastructure, so she just forwards it to them without modifying it, and since no new version of the model is generated, she does not need to store anything in the blockchain. Of course this is just an exemplary scenario; in other cases a common default infrastructure may be specified before the others enhance it with their artifacts. The process continues similarly: whenever participants receive a new version of the deployment model they: (i) verify its integrity by checking its stored versions in the blockchain, and if the check passes, they (ii) optionally operate on it by, e. g., adding new sub-resources to it, or altering the content of existing ones. If such changes were performed, (iii) they store a fingerprint of the new version in the blockchain, and then (iv) they further forward the model.

In Fig. 4, the solid arrows represent sending versions of the deployment model directly from one participant to the other, whereas the dashed ones represent communication between participants and the decentralized accountability layer. Furthermore, the numbered circles represent the order in which events happen in this specific scenario. The figure also shows how the blockchain gives total order to its transactions, and consequently to the stored deployment model states. This allows the reconstruction of the provenance of any given sub-resource by traversing the transactions in order, and collecting the partial fingerprints belonging to the sub-resource along with the author and the timestamp of each version. But how can we find out details about the exact changes that happened to each sub-resource? We answer this question in the following section.

4.3 Storage of deployment model States

The storage of arbitrary data in public blockchains is generally very expensive because all peers of the protocol need to store the entire blockchain locally. This led Ethtereum, for example, to set high fees for data storage. The general approach to address this issue is putting only a digest of the actual content in the blockchain rather than storing it entirely there. However, our scenario requires identifying the exact changes made by each participant, but storing merely the hash of the deployment model only helps detecting that something has changed from one state to another, but not exactly what. We present two approaches that ensure capturing enough details about the deployment model state changes to facilitate accountability while not storing the entire model in the blockchain.

In the first approach we try to identify the data items of the deployment model that correspond to changes exclusive to one participant. For example, in a certain scenario we could expect that a configuration file of an application server is expected to be created and edited by a single participant only, so storing its hash in the blockchain allows us to identify if some other participant breaks this rule. However, in other cases we could expect that such a configuration file can be altered by multiple participants, but that each entry of it, e. g., the listening port of the server, can be modified by a single participant only. In this case we store a map of the hashes of these entries in the blockchain instead. When we identify the correct level of data items that corresponds to changes exclusive to a single participant for each of the sub-resource types we have, i. e., algorithm implementations, configuration files, etc., we create a collection of the hashes of these data items and store it in the blockchain for every state change we go through. We call this collection the **fingerprint** of the state. This fingerprint allows us to detect malicious participants that edit data items that are not expected to be altered by them. However, if the level of exclusive changes is too low, e. g., at the level of single characters, storing the fingerprint in the blockchain becomes even more expensive than storing the whole state there.

In the second approach we store the whole deployment model in a content-addressable decentralized file storage accessible by all participants, such as Swarm [6], or Inter Planetary File System (IPFS) [9]. A decentralized storage provides at least the following two functions for storing and retrieving arbitrary files: $store : \mathcal{D} \rightarrow \mathcal{H}$ and $retrieve : \mathcal{H} \rightarrow \mathcal{D}$ where $\mathcal{H}$ is the range of some hash function h. At the same time, we store only the hash in the blockchain, which serves both as the address of the content in the decentralized storage and as a guarantee that it has not been altered there. However, in this approach encryption needs to be utilized, or else the privacy of stored content is lost.

In our work here, we use a hybrid of these two approaches: in the blockchain, we store a fingerprint listing all sub-resources identifiers and their hashes, whereas in the decentralized storage, we store the actual artifacts. In order to generate the fingerprint of a deployment model state m_t, we use the following function: $fingerprint(m_t) := \{(id_s, h(data)) | s = (id_s, data) \in \pi_S(m_t)\}$, then the blockchain transaction itself would be: $tx(m_t) := (id_m, author, fingerprint(m_t))$. To reconstruct the actual sub-resources of m_t, the fingerprint can be used to locate content in the decentralized storage as demonstrated in Fig. 4.

4.4 Establishing Identity and Authenticity

Public blockchains allow public access to their content; anyone can run a peer node and submit new transactions to it or read existing transactions. Consequently, anyone can pretend to be a participant in the development process, and inject fake or altered fingerprints of the deployment model. Thus, we need an additional mechanism to ensure the authenticity of participants without imposing the requirement of knowing all participants at the beginning of the process which is unrealistic in large-scale collaborations. Moreover, to identify their users, public blockchains use pseudonyms, which are hashes of the public key of each participant [25, 34], thus guaranteed to be unique while hiding the true identity. However, regarding accountability, it might be beneficial to expose actual identities.

A potential way to approach these issues is the usage of permissioned blockchains like Hyperledger Fabric [12], which establishes the identity of all participants and implements access-control to its content. However, the usage of such an architecture increases the centralization of the system. For example, a Membership Service Provider (MSP) is needed for the purposes of certificate issuing, validation, and user authentication, and its configuration is stored in the genesis block of the ledger [4]. MSPs are centralized services whose setup and operation need to be agreed upon by the participants beforehand. This has the same drawbacks as needing a trusted third-party, and further constitutes a single point of failure for the system. We propose a simple alternative approach to solving these problems which also builds upon public blockchains.

The basic idea is building a tree of trusted participants rooted at the *Service Owner*, which can be augmented with the Real-World Identity (RWI) of each one of them if needed by the use case. This tree is built with the assumption that if a participant A forwards the deployment model to another participant B, then A trusts B and knows their RWI. Based on this assumption, the approach simply requires A to insert a new node in the tree including this trust information before actually forwarding the model to B (providing that B is not already part of the tree). If all participants follow this rule, then we build a tree of trust rooted at the *Service Owner*, who, by transitivity, would trust all included participants.

Formally, based on the previous notations, and providing that $\mathcal{P}_{bc}$ refers to all potential blockchain-based participant identities (pseudonyms), referred to earlier as $\mathcal{P}$, and that $\mathcal{P}_{rwi}$ refers to all potential RWIs of participants, e. g., their names or email addresses, the tree of trust for a given deployment model id_m is identified as a tuple $T_{id_m} := (ownerId, V_T, E_T)$ where: (i) $ownerId$ is the blockchain-identity of the *Service Owner*, (ii) $V_T \subseteq \mathcal{P}_{bc} \times \mathcal{P}_{rwi}$ is the set of tree nodes each of which, $v := (bcid, rwid)$, represents the blockchain-identity and, optionally, the RWI of a trusted participant, and (iii) $E_T \subseteq V_T \times V_T$ is the set of tree edges that represent the inclusion of new participants by an existing one. In order to build this tree, we also use a public blockchain; when A wants to include B in the tree they submit a blockchain transaction $tx := (id_m, authorizer, authorized, rwi)$ where: (i) id_m: the identifier of the deployment model, (ii) $authorizer, authorized$: the blockchain identities of A and B, and (iii) rwi: the RWI of B. Reconstructing the tree of trust based on the set of all relevant transactions R_{id}, which can be obtained by traversing the blockchain, happens as follows:

$$\begin{aligned}
&ownerId := \pi_{authorizer}(tx) :\ \nexists tx_1 \in R_{id}\ s.t.\ tx_1 < tx \\
&V_T := \{(ownerId, \_)\} \bigcup \{(\pi_{authorized}(tx), \pi_{rwi}(tx))|\ tx \in R_{id}\} \\
&E_T := \{(v_1, v_2) \in V_T \times V_T|\ tx \in R_{id}\ s.t. \\
&\qquad\qquad \pi_{authorizer}(tx) = \pi_{bcid}(v_1)\ \wedge\ \pi_{authorized}(tx) = \pi_{bcid}(v_2)\}
\end{aligned}$$

where $<$ uses the total order of transactions guaranteed by the blockchain, and ($\_$) refers to an empty value (as the RWI of the Service Owner is not important). Now, we can show how we implement the mapping $\mathcal{A}$ that describes the authorized participants using the tree-of-trust (cf. Sect. 3.2).

Definition 3 (Authorized Participants Mapping). *Given a tree-of-trust of a collaboration process T_{id_m}, the mapping $\mathcal{A} : \mathcal{I}_m \rightarrow \wp(\mathcal{P}_{bc})$ is defined as:*

$$\begin{aligned}
\mathcal{A}(id_m) := \{p \in \mathcal{P}_{bc}|\ \exists path = \langle v_{start}, ..., v_{end}\rangle\ in\ T_{id_m}\ s.t.& \\
\pi_{bcid}(v_{start}) = ownerId \wedge \pi_{bcid}(v_{end}) = p\}& \quad \square
\end{aligned}$$

Here, we simply find the participants reachable form the root with a path.

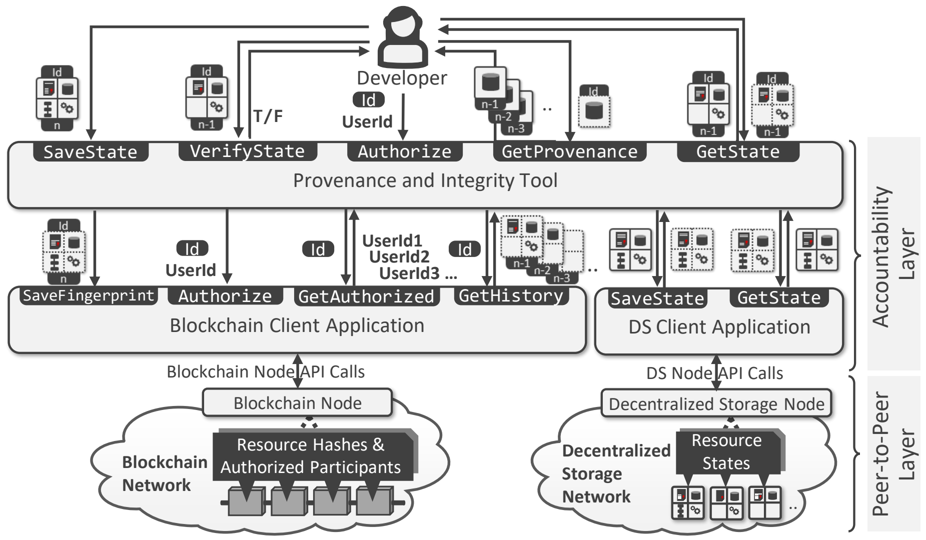

Fig. 5. The system architecture

5 Proof of Concept

The goal of this chapter is demonstrating how we realized the proposed approach by introducing a system architecture that supports it and by providing a prototypical implementation of this architecture that proves its feasibility.

5.1 System Architecture

Figure 5 depicts the suggested system architecture. The architecture is divided into two layers: the first layer, i. e., the **Peer-to-peer Layer** is located at the bottom of the architecture, and its components are not part of the client application itself, but rather allow it to access both the blockchain network, and the decentralized storage network. The layer represents the point of view in which a peer sees these networks: in order for peers to communicate with a public blockchain network, which is collectively responsible for storing the blockchain data, they need to run a local node implementing the corresponding blockchain protocol, and exposing an API that allows utilizing it. The same applies to the decentralized storage network; a local node is also needed here.

On the other hand, our approach relies on smart contracts, which are decentralized applications that provide an easy way to store and manage arbitrary data on the blockchain, to create two necessary repositories: (i) a repository for storing and managing the various states of the deployment model, and (ii) a repository for storing and managing the tree-of-trust for the collaboration process. The tasks of reading from these repositories and writing to them are part of the node's functionality, and are exposed via its API. The second layer, i. e., the

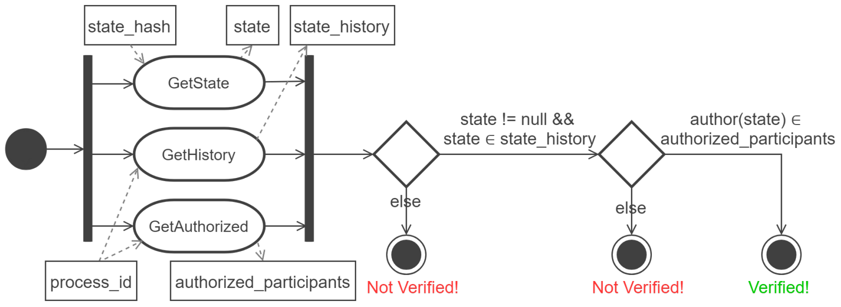

Fig. 6. UML activity diagram showing the state verification process

Accountability Layer is responsible for implementing the logic of the proposed approach. It is divided into two sub-layers: The one at the bottom consists of two client applications that communicate with the local peer-to-peer nodes on one hand, and simplify interacting with them on the other hand by exposing easy-to-use domain-specific operations to the sub-layer above it. The operations `SaveState`, and `SaveFingerprint` are responsible respectively for storing the state of a deployment model in the decentralized storage, and for storing its fingerprint in the blockchain. Besides its original purpose as a means to establish integrity of the deployment model, the fingerprint stored in the blockchain allows us to retrieve the state from the content-addressable decentralized storage using the `GetState` operation. On the other hand, the operation `GetHistory` gets the history of fingerprints for a given collaboration process, whereas `Authorize`, and `GetAuthorized` are used to authorize a new participant, and to read the list of all authorized participants respectively. The role of this sub-layer is similar to the role of the Data Source Layer of the 3-layer architecture [19].

The sub-layer at the top has the role of providing the "business-logic", and it divides its functionality into a set of operations: `GetState`, and `Authorize` simply expose the corresponding operations of the sub-layer below it. However, the other three operations are more sophisticated; `GetProvenance` gets the provenance of a specific sub-resource of the whole deployment model. To this end, it utilizes the `GetHistory` operation and uses its results to retrieve a complete history using the `GetState` operation, and finally analyzes the results. Moreover, `SaveState` is called before sending a deployment model to the next participant and it stores the current state of the resource in the decentralized storage using the `SaveState` operation at the lower level, and it also stores a fingerprint of the state in the blockchain by invoking the `SaveFingerprint` operation. On the other hand, `VerifyState` is invoked when receiving a new version of the deployment model in order to determine its integrity. The execution of this operation is depicted in Fig. 6 which shows that it uses the `GetHistory`, `GetState` and `GetAuthorized` operations of the sub-layer below to achieve its tasks.

5.2 Prototype

To validate the feasibility of the architecture, we have implemented a prototype that realizes it which is publicly accessible via Github[1]. To implement the bottom-most layer of the architecture, we have chosen Ethereum [34] as the underlying blockchain protocol since it is the most mature public blockchain that supports smart contracts and because it can be easily integrated with other peer-to-peer protocols from the Ethereum Foundation [17]. Smart contracts are programmed using Solidity, a specialized Turing-complete language. Listing 1 shows the smart contract responsible for managing the repository of the deployment model states:

```
contract Provenance {
  event NewState(string indexed _id, address indexed _author, bytes _fp);
  function addState(string _id, bytes _fp) public {
     emit NewState(_id, msg.sender, _fp);
  }
}
```

Listing 1. Ethereum smart contract for storing deployment model states

The contract has a single event and a single function that only emits it when invoked via a transaction. Emitting an event adds a log entry to the transaction's permanent storage with the parameters passed to the event [34]. In this case, we pass a string representing id_m, an address representing $author$, and a compressed version of $fingerprint(m_t)$ (c.f. Sect. 4.3). By storing data via logs, which are only readable via external applications, instead of state variables, we save costs as the price for storing them is cheaper [34]. Moreover, adding the keyword *indexed* to an event parameter speeds up queries that retrieves log entries with a filter based on this parameter. Finally, the smart contract for managing the tree-of-trust is formulated similarly. We connect our local node to a testnet instead of the mainnet as it allows us to run experiments free of charge.

On the other hand, we have chosen Swarm [6] as the implementation of the decentralized storage as it is meant to be self-sustainable in the sense that network peers are motivated to keep copies of the shared resources on their local nodes because they get paid for this with Ethereum tokens (Ethers), which allows peers to operate in an upload and disconnect mode [31].

Furthermore, we have integrated the *Accountability Layer* as a reusable sub-module into Winery [22], a web-based environment that allows the graphical modeling of TOSCA [1] topologies. Winery sub-modules are written in Java, and packaged using Maven. The prototype supports Cloud Service ARchives (CSARs) as deployment models. These archives are part of the TOSCA standard, and they contain the topology, management plans, and the various other software artifacts required for the correct provisioning, and operation of a cloud application. An important part of the CSAR is the *TOSCA Meta File* which allows to interpret its various components properly. This file is divided into blocks that describe each of the artifacts contained in the CSAR.

[1] https://github.com/OpenTOSCA/winery/releases/tag/paper%2Fgf-accountability

We use the TOSCA meta file as a fingerprint of the whole CSAR, so instead of storing the actual archive in the blockchain, we store the meta file (cf. Sect. 4.3). To this end, we augment each block with the digest of the corresponding artifact (sub-resource), so we make sure we capture any changes made to it over the stored state versions. When we store these fingerprints in the blockchain, they become immutable, and providing that all blockchain transactions are signed by their creators, we can identify the author of each version of the CSAR, and we can detect whenever a contained artifact is changed and by whom. Furthermore, we compress the contents of the meta file before storing it due to cost reasons. Finally, to allow showing detailed difference between the various states of a given sub-resources, when we store the fingerprint in the blockchain, we also store the changed artifacts in the Swarm. These artifacts can be referenced using their hashes which are parts of the fingerprint stored in the blockchain. This allows us to dynamically retrieve them, e.g., when the user wants to visualize the provenance of a specific sub-resource. In a previous work [35], we have shown how to address the issue of securing parts of a CSARs in the context of collaborations. This can also be applied in the context of the files we store in the Swarm to guarantee sensitive data is not leaked.

6 Evaluation

In this section we show the applicability of the aforementioned prototypical implementation to real-world use cases by evaluating the costs and additional execution times incurred when using it.

6.1 Cost

The costs incurred by the approach are divided into: (i) infrastructure costs, (ii) Ethereum transaction costs, and (iii) Swarm storage costs.

To allow the prototype to access the Ethereum and Swarm networks, participants are advised to run and maintain local nodes. These nodes need to be in sync with other nodes in their networks before they are usable, so it is suggested that they are always online. An alternative is connecting to publicly accessible nodes instead, but that increases security risks. The costs incurred here are due to maintaining these nodes and providing them with Internet connection. On the other hand, each Ethereum transaction incurs certain costs [34]. A transaction that invokes a smart contract has some fixed base cost in addition to costs related to the complexity of the code executed and the size of data newly stored in the blockchain due to this execution. These costs are introduced to prevent malicious or buggy code from running indefinitely or for too long on Ethereum nodes and they are calculated in terms of *Gas*. The author of a transaction pays for the gas it consumes, and the node mining it receives this payment as a fee. The price of the unit of gas is determined by the author, and it affects how quickly the transaction is processed by the network since miners are motivated to pick transactions with higher fees first when formulating new blocks. In the following,

Table 1. Costs incurred by storing TOSCA metadata files of various sizes in Ethereum

CSAR Complexity	**Metadata File Size**	**Gas Consumed**	**Price (Ether/Euro)**
Low (73 files)	9958 B	461106	(0.010144332 / **2.41**)
Medium (111 files)	13786 B	639123	(0.014066706 / **3.34**)
High (133 files)	17234 B	795659	(0.017504498 / **4.16**)

we show an estimation of the costs of the transactions issued by the prototype assuming a price of 22×10^{-9} Ethers per unit of gas which usually results in processing the transaction in the next 1-2 blocks. Furthermore, we assume an Ether to Euro conversion rate of 237.49 which is valid at the time of writing. The prototype issues two types of transactions: (i) *authorization transactions* to add entries to the tree-of-trust (c.f. Sect. 4.4). These transactions incur a minor cost due to the low amount of data stored. On average, a transaction of this type consumes 31000 gas units which corresponds to ~0.0007 Ethers or ~**0.17 Euros**. (ii) *provenance transactions* that contribute to building the graph G_{id_m} (c.f. Sect. 3.2). Most of the gas consumed here is due to the TOSCA meta file which we store compressed in them as a model fingerprint (c.f. Sect. 5.2), thus the more complex the model is, the higher the cost it incurs in this context since the complexity of the model is reflected in a larger TOSCA meta file. Table 1 summarizes the costs incurred by sample CSAR files of various complexities. Here we measure the complexity by the number of associated sub-resources (files).

Finally, as mentioned earlier, a decentralized storage system requires a mechanism to incentivize peers to store content of other peers. The amount of the incentive would depend on the size of files stored and the degree of replication required. However, Swarm has not implemented such a mechanism yet (planned for 2019 [30]). Thus for now no additional costs are incurred by using Swarm apart from operating a local node.

6.2 Execution Time

The usage of the prototype in Winery increases the execution times of both exporting a CSAR and importing it. When exporting a CSAR, additional time is required to (i) hash the contents of the sub-resources in order to formulate the fingerprint, (ii) store the contents of the CSAR in a local Swarm node, and (iii) issue a blockchain transaction that contains the fingerprint. Whereas, when importing a CSAR additional time is required to validate its integrity by querying the local Ethereum node for relevant information (c.f. Fig. 6). Ethereum blocks are generated by the network at almost a constant rate of 15 seconds [34]. Thus, depending on the traffic and the gas price, time factor (iii) is usually 15-30 seconds pertaining to a delay of 1 to 2 blocks. On the other hand Table 2 shows the effect of the remaining factors on the import and export times of 5 exemplary CSAR files with different sizes and sub-resource counts. These measurements were performed on a computer running Windows 10 64-bits with a (Intel(R) Core(TM) I7-4710MQ CPU @ 2.5GHz) processor and 16 GB of RAM.

Table 2. Increase in CSAR import/export times when using the prototype

CSAR	**Export Time (mm:ss)**		**Import Time (mm:ss)**	
	Original	**Increase**	**Original**	**Increase**
model 1 (73 files, 19 MB)	02:12	00:09	00:02	00:04
model 2 (106 files, 27.8 MB)	02:37	00:08	00:03	00:04
model 3 (111 files, 20 MB)	02:48	00:08	00:02	00:05
model 4 (121 files, 11 MB)	02:49	00:06	00:03	00:05
model 5 (133 files, 27 MB)	03:01	00:11	00:04	00:06

7 Related Work

Web-of-Trust (WoT) (Zimmermann [26]) provides a distributed alternative to the centralized Public-Key Infrastructure (PKI). Our identity and authenticity approach shares common features with WoT; the act of inserting another participant's address in the blockchain is comparable to signing another person's certificate in the WoT because all transactions in the blockchain are signed by their authors, and a participant's address is derived from their public key. On the other hand, our approach is simple and suitable specifically for collaborations initiated by a Service Owner, whereas, WoT is a general-purpose scheme which addresses further aspects irrelevant to the use case at hand.

Furthermore, several centralized approaches were suggested to support collaborative development processes in various domains [14, 33]. However, these approaches share common properties such as centralized access-control mechanisms or strong assumptions about the underlying data structures which make them not suitable in the context of business-critical collaborative deployment modeling. Furthermore, they do not focuses on maintaining model provenance.

On the other hand, few decentralized approaches exist for supporting collaboration in software development scenarios. These approaches focus on providing a decentralized implementation of the Git protocol; Rashkovskii [27] proposes a Git implementation in which files are distributed on a Bitcoin-like proof-of-work-based blockchain network, which would ensure immutability of commit histories, and thus promote accountability of participants' actions. However, the approach requires a new proof-of-work blockchain, and since the security of proof-of-work depends mainly on the hash-rate of the network, many participants need to adopt it before it becomes usable, which reduces the trust in the approach altogether. Furthermore, Ball [7] shows how the file storage for Git can be implemented using the BitTorrent P2P protocol. The approach is completely decentralized, and uses Bitcoin as a name directory for repository addresses. However, the history of commits is not immutably persisted, and thus accountability is not achieved. Finally, Beregszaszi [10], also suggests a decentralized storage of Git objects, but with the help of IPFS [9] which is a content-addressed P2P filesystem. The approach further depends on Ethereum smart contracts to provide access-control and to manage pointers of the latest repository revisions. However, the history of commits is not stored in the blockchain, and thus resource provenance and process accountability cannot be guaranteed.

8 Concluding Remarks and Future Work

In this paper we presented a novel approach to enable accountability in the collaborative development processes of declarative deployment models for business-critical applications. Our approach is decentralized and based on blockchains. This allowed us to ensure integrity and provenance, the two aspects of accountability, without the need for a trusted third-party which could be difficult to agree upon, and would constitute a single point of failure. Our approach also provided a decentralized mechanism to maintain the identity and authenticity of all participants as part of the process integrity. Finally, as a future work, we plan to enhance the security of the approach via tackling the issue of an authorized participant losing their private key by making the tree-of-trust dynamic based on participants' behavior, or by adding a blacklisting mechanism to it.

Acknowledgments This research was funded by the Ministry of Science of Baden-Württemberg for the doctoral program "Services Computing", and by the project SePiA.Pro (01MD16013F) of the BMWi program Smart Service World.

References

1. Topology and orchestration specification for cloud applications (Nov 2012), http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html
2. Chef (May 2018), https://www.chef.io/
3. Kubernetes (May 2018), https://kubernetes.io
4. Membership service providers (MSP) (Jul 2018), http://hyperledger-fabric.readthedocs.io/en/release-1.1/msp.html
5. Rewriting history — Git commit –amend and other methods of rewriting history (2018), https://www.atlassian.com/git/tutorials/rewriting-history
6. Swarm (Jul 2018), https://github.com/ethersphere/swarm
7. Ball, C.: Announcing GitTorrent: a decentralized GitHub (May 2015), https://blog.printf.net/articles/2015/05/29/announcing-gittorrent-a-decentralized-github/
8. Baumann, F.W., Breitenbücher, U., Falkenthal, M., Grünert, G., Hudert, S.: Industrial data sharing with data access policy. In: CDVE. pp. 215–219 (2017)
9. Benet, J.: IPFS-content addressed, versioned, P2P file system. arXiv preprint arXiv:1407.3561 (2014)
10. Beregszaszi, A.: Mango (Jul 2016), https://medium.com/@alexberegszaszi/mango-git-completely-decentralised-7aef8bcbcfe6
11. Bergmayr, A., et al.: A systematic review of cloud modeling languages. ACM Computing Surveys (CSUR) **51**(1), 22 (2018)
12. Cachin, C.: Architecture of the Hyperledger Blockchain Fabric. Tech. rep., IBM Research - Zurich (2016)
13. Cecere, L.: Seven use cases for Hyperledger in supply chain (Jan 2017), http://www.supplychainshaman.com/big-data-supply-chains-2/10-use-cases-in-supply-chain-for-hyperledger/
14. Cera, C.D., et al.: Role-based viewing envelopes for information protection in collaborative modeling. Computer-Aided Design **36**(9), 873–886 (2004)
15. Chacon, S., Straub, B.: Pro Git. Apress (2014)

16. Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, F., Wettinger, J.: Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In: Proceedings of the 9th International Conference on Pervasive Patterns and Applications. pp. 22–27. Xpert Publishing Services (XPS) (2017)
17. Ethereum Foundation: Web3 base layer services (Aug 2018), http://ethdocs.org/en/latest/contracts-and-transactions/web3-base-layer-services.html
18. Falkenthal, M., et al.: Towards function and data shipping in manufacturing environments: how cloud technologies leverage the 4th industrial revolution. In: Proceedings of the 10th Advanced Summer School on Service Oriented Computing. pp. 16–25. IBM Research Division (2016)
19. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
20. Goodin, D.: Massive denial-of-service attack on GitHub tied to Chinese government (Mar 2015), https://arstechnica.com/information-technology/2015/03/massive-denial-of-service-attack-on-github-tied-to-chinese-government/
21. King, S., Nadal, S.: Ppcoin: peer-to-peer crypto-currency with proof-of-stake (Aug 2012), https://peercoin.net/assets/paper/peercoin-paper.pdf
22. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – A Modeling Tool for TOSCA-based Cloud Applications. In: ICSOC 2013. Springer (Dec 2013)
23. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (1982)
24. Mohtashami, M., Marlowe, T., Kirova, V., P. Deek, F.: Risk management for collaborative software development. Information Systems Management **23**(4), 20–30 (2006). https://doi.org/10.1201/1078.10580530/46352.23.4.20060901/95109.3
25. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
26. Philip, Z.: PGP user's guide, volume I: essential topics. Phil's Pretty Good Software, version 2.6.2 edn. (Oct 1994), https://web.pa.msu.edu/reference/pgpdoc1.html
27. Rashkovskii, Y.: Gitchain (Sep 2014), https://www.kickstarter.com/projects/612530753/gitchain/description
28. Schwartz, D., Youngs, N., Britto, A., et al.: The Ripple protocol consensus algorithm (2014), https://ripple.com/files/ripple_consensus_whitepaper.pdf
29. Steinbeis, G.: Change author of SVN commit (Jun 2011), https://blog.tinned-software.net/change-author-of-last-svn-commit/
30. Trón, V.: Announcing Swarm proof-of-concept release 3 (Jun 2018), https://blog.ethereum.org/2018/06/21/announcing-swarm-proof-of-concept-release-3/
31. Trón, V., Fischer, A., Nagy, D.A., Felföldi, Z., Johnson, N.: Swap, swear and swindle: incentive system for swarm (May 2016), https://swarm-gateways.net/bzz://theswarm.eth/ethersphere/orange-papers/1/sw%5E3.pdf
32. Viriyasitavat, W., Martin, A.: In the relation of workflow and trust characteristics, and requirements in service workflows. In: Informatics Engineering and Information Science. pp. 492–506. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
33. Wang, Y., et al.: Intellectual property protection in collaborative design through lean information modeling and sharing. Journal of computing and information science in engineering **6**(2), 149–159 (2006)
34. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger - Byzantium Version (2018), https://ethereum.github.io/yellowpaper/paper.pdf
35. Yussupov, V., Falkenthal, M., Kopp, O., Leymann, F., Zimmermann, M.: Secure collaborative development of cloud application deployment models. In: Proceedings of the 12th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE) (2018)