Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 99.9999/ACCESS.9999.DOI

Accountable Clouds through Blockchain

MIRKO ZICHICHI[1], GABRIELE D'ANGELO[2,5], STEFANO FERRETTI[3] (Member, IEEE), MORENO MARZOLLA[4,5]

[1]Ontology Engineering Group, Universidad Politécnica de Madrid, ETSIINF Campus de Montegancedo s/n, Boadilla de Monte, 28660 Madrid, Spain (e-mail: mirko.zichichi@upm.es)
[2]Department of Computer Science and Engineering (DISI), University of Bologna, Mura Anteo Zamboni 7, 40126 Bologna, Italy (e-mail: g.dangelo@unibo.it)
[3]Dipartimento di Scienze Pure e Applicate (DiSPeA), University of Urbino Carlo Bo, Piazza della Repubblica 13, 61029 Urbino, Italy (e-mail: stefano.ferretti@uniurb.it)
[4]Department of Computer Science and Engineering (DISI), University of Bologna, Mura Anteo Zamboni 7, 40126 Bologna, Italy (e-mail: moreno.marzolla@unibo.it)
[5]Center for Inter-Department Industrial Research ICT, University of Bologna, 40126 Bologna, Italy

Corresponding author: S. Ferretti (e-mail: stefano.ferretti@uniurb.it).

ABSTRACT We present a solution for accountability in Cloud infrastructures based on blockchain[a]. We show that, through smart contracts, it is possible to create an unforgeable log that can be used for auditing and automatic Service Level Agreement (SLA) verification. As a practical case study, we consider Cloud storage services and define interaction protocols for registering the outcome of each file operation in the blockchain. We developed a prototype implementation that runs on the GoQuorum, Hyperledger Besu, and Polygon blockchains, using different consensus protocols. Using a dedicated testbed, we discuss the performance of our implementation in terms of latencies, error rates and gas usage. Results demonstrate the viability of our approach over permissioned blockchains, with better performance for the Polygon and GoQuorum Raft decentralized systems. Our implementation enables interoperability, given that it is supported by the Ethereum Virtual Machine which currently is underlying several blockchain platforms.

[a]An early version of this work appeared in [1]. This paper is an extensively revised and extended version where more than 50% is new material.

INDEX TERMS Blockchain, Smart Contracts, Cloud Computing

I. INTRODUCTION

Cloud computing is a well-established paradigm for providing computation and storage resources according to a "pay as you go" model. In Cloud computing, *service providers* own computing resources and provide remote access to those resources to *customers* for a fee [2].

The level of abstraction at which a customer interacts with a Cloud infrastructure is defined by the *service model*. In a Software as a Service (SaaS) Cloud, customers are provided with application services running in the Cloud infrastructure. "Google Workspace" and "Microsoft Office Online" are examples of widely used SaaS Clouds. A Platform as a Service (PaaS) Cloud provides programming languages, tools, and a hosting environment for applications developed by the customer. Examples of PaaS solutions are AppEngine by Google, Force.com from SalesForce, Microsoft's Azure, and Amazon's Elastic Beanstalk. Finally, an Infrastructure as a Service (IaaS) Cloud provides low-level computing capabilities such as processing, storage, and networks where the customer can run arbitrary software, including operating systems and applications. Amazon EC2 is an example of IaaS Cloud.

The mode of operation of a Cloud defines its *deployment model*. A *Private Cloud* is operated exclusively for a customer organization; it might be managed or owned by that organization, although this is not required. A *Community Cloud* is shared by several organizations and supports a specific community with common concerns (e.g., regulatory requirements). A *Public Cloud* is made available to the general public and is owned by an organization selling Cloud services. Finally, a *Hybrid Cloud* is built upon a combination of private, public, and community Clouds.

Cloud computing allows separation between construction and operation of the infrastructure and providing end-user services. This opportunity enables the existence of at least three categories of providers and users: (i) the provider of Cloud resources, (ii) the provider of services implemented upon these resources, and (iii) the customer of these services. Although providers and customers could be the same (e.g., in the case of Private Clouds), in most cases, they are different

entities. This implies that customers have no control over the resources they use: a common joke is to replace "Cloud computing" with "other people's computers" so that the sentence "storing data in the Cloud" becomes "storing data on other people's computers".

The lack of direct control over the Cloud infrastructure is a serious concern for some users, e.g., those subject to regulatory obligations or handling sensitive information. The legal implications of sending data and computation to a third party located in a different country with a different data protection legislation are still a gray area [3]. It is essential to recognize that these are old problems arising in a new context: for example, moving the production of goods to other countries may require the transfer of valuable information (e.g., chip design, special production techniques) to a legal context that may differ strikingly from that of the owner company.

The separation between resource providers and customers introduces a problem that is not unique to Clouds, as every service provider faces it: if something goes wrong (e.g., data is lost, or the computation returns an incorrect result), how do we determine whether the customer or the provider caused the problem? As an example, consider the following scenarios:

Scenario 1

Company *A* offloads its customer-facing application to a Cloud provider *B*. Suddenly, *A*'s application crashes, and customers complain with *A*, asking for compensation. In turn, *A* accuses the Cloud provider *B* of the caused service unavailability. However, *B* asserts that its infrastructure was operating correctly during service unavailability, thus suggesting that the problem was at the software level, i.e., *A*'s application fault.

Scenario 2

Company *A* stores important data on a Cloud operated by *B*. At some point, some data is found to be missing. *A* blames *B*, who claims that the missing data have never been uploaded. (An alternative scenario is that *B* asserts that the data have been deleted upon explicit request by *A*).

Cloud providers offer services on an as-is and as-available basis, subject to terms and conditions that disclaim any responsibility no matter what. For example, Microsoft's Service Level Agreements (SLAs) for online services contain a clause according to which the entity that decides on client-initiated disputes is the service provider (i.e., Microsoft) itself[1]:

> We [Microsoft] will evaluate all information reasonably available to us and make a good faith determination of whether a Service Credit is owed. We will use commercially reasonable efforts to process claims during the subsequent month and within forty-five (45) days of receipt. You must be in compliance with the Agreement in order to be eligible for a Service Credit. If we determine that a Service Credit is owed to you, we will apply the Service Credit to your Applicable Monthly Service Fees.

Having the service provider make "a good faith determination" about SLA violations is far from satisfactory. Amazon Web Services is even more dismissive: their general customer agreement[2] denies any compensation for a broad range of failures, no matter what, so that there is no need to decide who's to blame:

> [...] Neither we nor any of our affiliates or licensors will be responsible for any compensation, reimbursement, or damages arising in connection with: [...] d) any unauthorized access to, alteration of, or the deletion, destruction, damage, loss, or failure to store any of your content or other data.

Clauses like those above are common in the Information Technology world, since they favor the party that defines them (i.e., the service providers). There is, however, an objective problem in resolving disputes in the absence of solid evidence, so it is no surprise that SLA are as forgiving as possible, ultimately limiting the adoption of Cloud technologies.

All these issues might be addressed by adding accountability to Cloud services [4]–[8]. Indeed, an accountable Cloud would be capable of attributing actions and transactions to specific entities, thus adding responsibility to the functionalities and behavior of all actors involved in the Cloud service applications.

So far, accountability in distributed systems has relied on a trusted third party or to tamper-proof hardware devices [9]. Neither of these is desirable because, in both cases, trust is assumed rather than derived from verifiable system properties.

In this paper, we argue that blockchain technology can address the accountability problem in Cloud infrastructures. To support this claim, we have developed a prototype component responsible for logging events in a distributed, unforgeable event log. The log contains the sequence of interactions between a customer and the service provider and can be used to settle disputes if problems arise. Additionally, the blockchain allows the implementation of *smart contracts*through which it might be possible to write programs that can negotiate and verify the fulfillment of SLAs. Moreover, an effective use of a blockchain-based system, such as the one we present in this work, can help auditors implement an integrated and automated audit framework that enhances the efficiency, effectiveness, and quality of Cloud operations. Indeed, such a system would enhance auditing with COSO, COBIT, and ISO Control Frameworks [10].

The main contribution of this paper is as follows:

[1]https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services, Accessed on 2022-11-25

[2]https://aws.amazon.com/agreement/ Section 11; Accessed on 2022-11-25

We provide a protocol that, based on blockchain technologies, allows us to build an unforgeable log for Cloud accountability. The blockchain allows tamper-proof logging of events to verify if Cloud Service Level Agreements are violated.

We implement our protocol through a smart contract set in Solidity. Our solution, concerning state of the art, is supported by the Ethereum Virtual Machine, which is currently used by several blockchain-based systems. Thus, this design choice enables interoperability over multiple blockchain platforms.

We deploy and test our implementation over different blockchain platforms, i.e., GoQuorum, Hyperledger Besu, and Polygon, and different consensus protocols. Results demonstrate the viability of our approach, with better performance for Polygon and GoQuorum Raft.

This paper is organized as follows. In Section II, we provide some background on Cloud computing, blockchain, and smart contracts. In Section III, we highlight some of the challenges and requirements of accountable Clouds. Section IV investigates how blockchain-based technologies can enforce accountability in a case study dealing with a cloud-based storage service. Section V describes an actual implementation of the proposed system, whose performance is experimentally evaluated in Section VI. Finally, conclusions and future research directions are discussed in Section VIII.

II. BACKGROUND

To make this paper self-contained, we provide some background on Cloud computing infrastructures, accountability, blockchain technology, and smart contracts.

A. CLOUD COMPUTING

The main characteristics of a Cloud environment are [11]:

On-demand self-service: the ability to provide resources (e.g., CPU time, network storage) as needed [12], [13];
Broad network access: resources can be accessed through the network [12];
Resource pooling: virtual and physical resources can be pooled and assigned dynamically to consumers using a multi-tenant model [13];
Elasticity: dynamic provision of resources to enable customer applications to scale up and down [12], [13];
Measured service: resource and service usages are optimized through a pay-per-use model [5], [14].

B. ACCOUNTABILITY IN CLOUD COMPUTING

The importance of accountability in distributed systems in general [15], [16] and Cloud computing in particular [5], [17]–[22] has already been recognized. In [5] the author discusses the requirements for achieving accountability in clouds through tamper-evident logs: *completeness* (all SLA violations are eventually reported), *accuracy* (no violations are reported if the SLA is not violated), and *verifiability* (a third party can independently verify all reported violations). To realize an accountable Cloud based on trusted logs, it is necessary to decide what to log and how to log. We consider "how" first. Logging must guarantee fairness and non-repudiation, ensuring that the misbehavior of others does not disadvantage well-behaved parties and that no party can subsequently deny their participation. It should enable tracing back the causes of an "incident" (i.e., a behavior that is not SLA compliant) after it has occurred. Cloud providers and customers require protection for each other's actions, with all assurances rooted on an independent source of trust. For example, there should be a user-verifiable assurance that the data, applications, and services they deploy in the Cloud are secure even against impairment by Cloud system administrators. As concerns "what" to record, Cloud computing creates new relationships between an organization and third-party Cloud service providers. The general scenario is that Cloud services could be arbitrarily complex. Providers will offer their services to consumers with specific Quality of Service (QoS) attributes, such as reliability and security, under specific terms and conditions [14]. Most of the existing research on SLA management focuses on computational and algorithmic aspects of QoS monitoring and provisioning. Specifically, considerable effort has been spent developing proactive or reactive algorithms for allocating the appropriate number and resources needed to meet a set of QoS requirements. However, SLA violations happen in practice, and it is necessary to deal with them. Currently, the handling of SLA violations is entirely based on "out of band" negotiations between service providers and customers since the systems being monitored cannot provide legal evidence of malfunctions (or lack of). What is needed is a framework or a set of technologies that enable the creation of SLA clauses in a machine-readable form, such that users can be assured of their effective enforcement in the event of a violation. The blockchain's transparency and immutability and the smart contracts' automation have already been proposed to deal with SLA violations in Cloud services. However, such solutions still need to be thoroughly studied and evaluated [23].

In [1], the authors introduced the problem and proposed an initial blockchain-based solution that needed to be implemented and evaluated in terms of performance (e.g., scalability). A partially overlapping problem is discussed in [24], in which the authors explore the usage of blockchains to support cloud exchanges (i.e., marketplaces for cloud services). QoS and SLA violations are relevant topics for cloud exchanges that need specific and trusted solutions. On the other hand, in [25] the authors present an accountable cloud data storage that, similarly to our work, is implemented using Ethereum smart contracts. The difference with our work is that their evaluation focuses only on off-chain operations related to data storage, and only the gas usage is measured regarding on-chain operations. Additionally, we test the performance of three different blockchain implementations.

Another partially overlapping problem is data accountability, in which the goal is to obtain unified control and assign responsibilities to the operation on data hosted on a cloud

infrastruture. Also, in this case, a solution based on cloud-blockchain fusion [26] can be implemented. Worth of notice is also the work presented in [27], since it is one of the first works that integrates a blockchain with a cloud system to provide accountability. However, both solutions [26], [27] are limited to the Hyperledger Fabric blockchain and are not interoperable with blockchains based on the Ethereum Virtual Machine.

C. BLOCKCHAIN AND SMART CONTRACTS

Distributed Ledger Technologies (DLT) consist of networks of nodes that maintain a single ledger and follow the same protocol for appending information to it. The blockchain is a type of DLT where the ledger is organized into blocks, and each block is sequentially linked to the previous one [28]. The execution of the same protocol, i.e., source code, guarantees (most of the time) the property of being tamper-proof and not forgeable. This allows a trust mechanism to be created without the need for third-parties [29]. The untampered data availability makes DLT a promising tool for developing new types of applications where immutability and transparency are requiremed. Examples of these applications can be found in general-purpose blockchains [30]–[32].

There are different implementations of DLTs, each with its pros and cons. Permissionless DLTs are systems in which anyone can participate in the consensus mechanism. Permissioned DLTs, on the other hand, have a privileged set of nodes that are authorized to execute the consensus mechanism. In both cases, the full ledger can be either private or accessible by anyone, i.e., public. Another distinction lies in the support for smart contracts, a feature that quite often has a negative impact on the system scalability and responsiveness [30]. In fact, DLTs that are believed to provide better scalability often lack support for smart contracts. To address this issue, IOTA [33] implements a more scalable solution for distributing the ledger.

A smart contract is a program, in compiled or source form, that is deployed in a DLT environment [34]. The program is executed deterministically by different participants in the DLT with the same inputs, and therefore must produce the same results. When a smart contract is deployed on the DLT and the issuer is confident (e.g., by reviewing the code) that the code embodies the intended behavior, then transactions originating from that contract can be considered "trusted" without requiring the presence of a third party. This principle is based on the assumption that most DLT nodes are honest and follow the same protocol.

However, smart contracts are usually isolated from the outside world, e.g., they cannot contact a website, in order to ensure that execution is more resistant to attacks with a higher degree of certainty [31]. This limits the possibilities of using these technologies, given that many applications require real-time information from the outside world. In this context, oracles assist DLTs in enabling smart contracts to operate in the real world by flowing data from services external to the DLT [35]. They act as a bridge, providing the ability to retrieve, verify and digest data into smart contracts. Their off-chain execution can be centralized, i.e., from a single source, or decentralized, based on the consensus of a multitude of sources.

An exciting aspect of smart contracts is their ability to be self-enforcing in verifying the fulfillment of SLA agreements. Smart contracts allow the formulation of sets of machine-readable rules from service contracts, therefore transforming rules that are typically written in "legal-ese" into software programs. In our scenario, smart contracts might contain two kinds of contractual clauses: (i) terms and conditions and (ii) SLAs. Terms and conditions are concerned with rights, obligations, and prohibitions to perform a particular action; whereas SLAs are concerned with rights, obligations, and prohibitions to maintain a given service in a particular state. Smart contracts allow the definition of computational procedures for monitoring and detecting rule violations. This can be accomplished by recording service interactions at a granularity that is sufficient for checking if they comply with the rights (permissions), obligations, and prohibitions stipulated in contract clauses and tracing the causes of violations.

III. OVERVIEW OF CLOUDSLA

This section presents CloudSLA, a protocol that builds a trusted, tamper-proof log of actions executed in a cloud service through blockchain technologies. These interactions are represented as transactions recorded in the blockchain. Through smart contracts, all parties can check the trusted log and find and resolve disputes arising from SLA violations.

When API calls involve transferring a large amount of data (e.g., a file upload), cryptographic hashes are used in order not to store too much information into the blockchain. There are no particular constraints on the Cloud APIs under consideration: instantiating a virtual machine, uploading, deletion, or modification of files, and accessing a given resource, are all examples of events that can be recorded. All these events are notified in the blockchain by the entity invoking the request (the end user or a delegate) and/or by the entity receiving the request (the Cloud provider). The rationale is that recording all the involved parties' activities can help reveal the causes of a SLA violation.

In the rest of this paper, we consider a specific use case for implementing a Cloud storage service for data archival and backup, similar to Amazon Glacier. The service exposes the following Application Programming Interface (API):

- `Upload(f, c)`: Upload a file with unique identifier f and content c; if a file with id f already exists, its content is overwritten;
- `Delete(f)`: Delete a file with id f; if the file does not exist, this operation does nothing;
- c `Read(f)`: Return the content c of the file whose id is f; if the file does not exist, return the special value Nil.

We assume the existence of some authorization/authentication mechanism that allows users to access only the files that they

are allowed to. We also assume that the User encrypts each file before uploading it to the Cloud. The Cloud should not be able to decrypt any user-generated content, to prevent a class of insider threats.

The blockchain can be used following a notary scheme: let us assume that the provider fails to deliver a data block x requested by the user, or that the delivered data is different from what is expected. In this case, inspection on the blockchain can reveal whether the provider lost x or some updates to x, or the user has deleted or never uploaded x (or some modifications to x).

Another classic SLA example is: "99% of transactions during a daily activity must have a response time below a certain threshold t". This SLA can be safely monitored if we assume the presence of a (third) trusted component that logs response times and can audit (virtualized) resource usage [36]. In practice, self-enforcing smart contracts should be coupled with specific oracles to monitor response times and, based on the SLA, pay the damaged entity (more often, the customer) accordingly.

A similar strategy could work as well to monitor SLAs stipulated in terms of adequate resource capacities provided by the Cloud, rather than applications-specific performance metrics [37]. Thus, CloudSLA would allow checking if the Cloud provider allocated the proper amount of resources, e.g. processing and storage capacities, RAM, and middleware software.

As a concrete example, let us consider a customer who uploads some content on a Cloud storage service. For simplicity, let us assume that the content is a file (similar reasoning would apply to data chunks or other kinds of information). In the following, the customer wants to be sure that the uploaded files are not removed or altered by the Cloud provider. This can be obtained by using different execution architectures:

1) blockchain-based double signed transactions;
2) blockchain-based logging without smart contracts;
3) blockchain-based logging and smart contracts.

Double-signed transactions are signed by multiple parties, and can be used to certify that a transaction has been agreed upon by both the customer and the cloud provider. Double-signed transactions are simple to use and require a low overhead since they can be realized with few interactions. On the other hand, this approach would provide a coarse-grained representation of the interactions between the user and the cloud, because double-signed transactions certify whether the parties agree on something. This all-or-nothing result can be quite limiting since it relies on the two parties to agree.

Blockchain-based logging (without smart contracts) allows the recording of all interactions between the User and the Cloud. In the case of SLA violations, each party can trigger a verification by a third entity (e.g., an arbitrator) to identify who is responsible for such a violation. It is worth noting that, in this architecture, the arbitrator is not required to have been involved in any previous interaction with either the User or the Cloud since it can use the information publicly provided by the blockchain to determine the responsibilities.

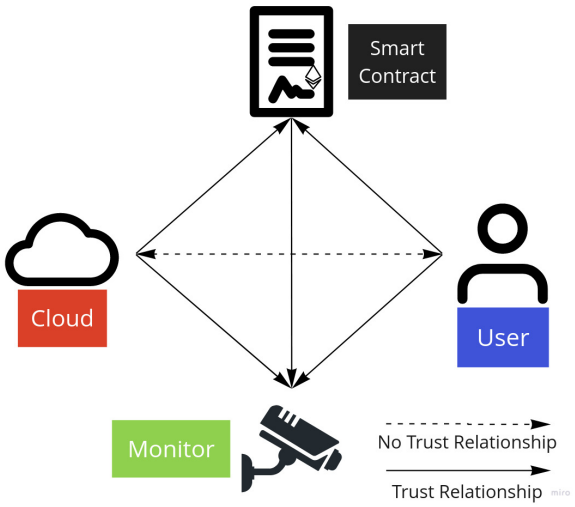

FIGURE 1. Cloud Service Level Agreement environment interactions.

Another option is to employ a smart contract acting as the arbitrator. In this approach, the smart contract verifies all events stored in the blockchain, identifying SLA violations, and calculates compensations if necessary. The main advantage of this approach is that no third party needs to be involved in resolving disputes. In particular, since the content of the smart contract can be accessed by both parties, they can verify its correctness before agreeing to its terms. In other words, the trust of the User and the Cloud provider is on the smart contract (that can be inspected and verified), following the notion that "code is law".

IV. SMART CONTRACTS FOR CLOUD SERVICE LEVEL AGREEMENTS

In this section we describe a smart contract that leverages blockchain-based logging for monitoring SLA violations of functional requirements of the file storage case study. The CloudSLA smart contract can help attribute SLA violations to the appropriate party for the following three operations: upload, delete, and read. We assume that the following active entities are involved (see Figure 1): *User*, *Cloud provider*, *CloudSLA smart contract*, and *Monitor* (i.e., an oracle).

For simplicity, we consider the blockchain as a passive entity that receives and stores events generated by active entities (User and Cloud). However, in the following discussion, we might state that an entity, say the Cloud, receives a transaction from the blockchain. This is a simplification to state that the blockchain network nodes reached a consensus on a transaction that includes a smart contract event triggered by the execution of a method while the Cloud was listening for such events.

It is important to remark that the assumptions above are only intended to simplify the discussion. In other words, they are not needed for guaranteeing the correctness of the proposed approach. The proposed system can operate correctly by adhering to nothing more than the usual development

practices employed in common DLT-based systems.

A. SMART CONTRACT OPERATION

The CloudSLA smart contract is the on-chain representation of a SLA contract. CloudSLA includes most of the information that builds up the agreement between a User and a Cloud provider. The operations enabled by this smart contract are described below.

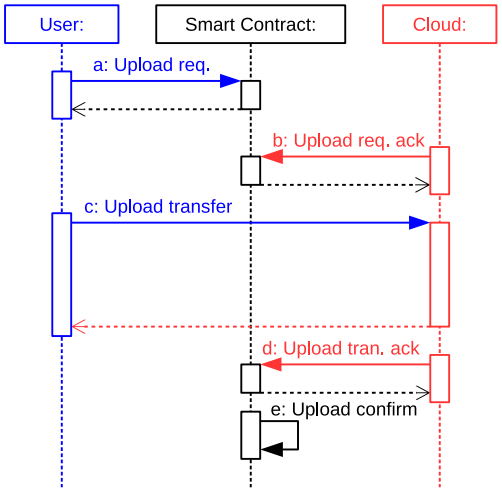

FIGURE 2. UML sequence diagram for the interactions involved in the file upload operation.

1) Upload

Figure 2 shows the behavior of the involved entities when the User uploads a file. What follows is the description of the Upload operation execution.

(a) Before transmitting the file to the Cloud, the User starts the upload operation using the smart contract method UploadRequest(). This method registers the upload request in the blockchain (arrow *a* in the diagram). It takes as input a *filepath*, i.e., a unique identifier by which the Cloud identifies a user's file in its storage. The identifier will be stored in the blockchain ledger, so it should not convey any information that could reveal its content or true location. The UploadRequest() function also takes as input a challenge [38], which consists of the hash digest of the file's hash digest, i.e., the result of executing the hash function twice, one after the other, on the file. This is done to hide the hash of the content so that it can be verified (by the smart contract) once the Cloud has uploaded the file. This is sometimes called "hash masking".

(b) Once the Cloud receives the transaction, including the upload request event from the blockchain, it can accept such a request by issuing an upload ACK message to the User through the invocation of the method UploadRequestAck().

(c) Once the User receives the transaction, including the upload ACK event, it can start the data upload to the Cloud.

(d) Once the upload finishes, the Cloud logs the success of this operation with a new transaction; in this transaction, the Cloud invokes the UploadTransferAck() method that stores the file's hash digest in the blockchain.

(e) Finally, the previous method invokes the UploadConfirm() method, (arrow *e*), which uses the file's hash digest provided by the Cloud as a response to the challenge set previously by the User. This method executes the hash function on the file's digest and checks the result with the data provided by the User. This operation confirms or rejects the hash digest published by the Cloud. If rejected, then the Cloud should delete the received file.

Through these steps, anyone can verify the correctness of the uploaded file by checking the digest provided by the Cloud and the related confirmation by the User. Recall that anyone who has access to the blockchain can check what the Cloud and the User stored, and thus he/she can understand if one of the two parties did not behave correctly. Moreover, the automatic execution of the smart contract challenge-response method enables file integrity validation and the possible rejection of the upload operation. This might not be considered a violation since there could have been some transmission issues not due to the Cloud.

Analysis.

Upon registration of message *e* (Upload Confirm), the following properties are guaranteed:

1) Neither the User nor the Cloud provider can claim that no upload was requested; indeed, the User stored a publicly-visible upload request on the blockchain (request *a*), that the Cloud acknowledged with message *b*; if the Cloud was unavailable right after message *a*, e.g., because it was down, it can nevertheless see the request from the blockchain as soon as it is operational again;

2) Neither the User nor the Cloud provider can claim that no file was transferred; indeed, an explicit upload transfer ack *d*, containing the file hash digest, is stored in the blockchain. The User can verify the correctness of the hash and repeat the upload in case of mismatch; the has must match the one from the initial upload request *a*.

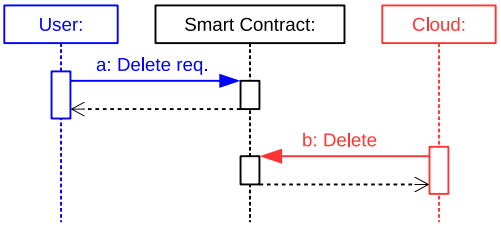

FIGURE 3. UML sequence diagram for the interactions involved in the file delete operation.

2) Delete

The interactions required to delete a file from the Cloud are shown in Figure 3. What follows is the description of the

Delete operation execution.

(a) The User issues a delete request by invoking the DeleteRequest() method with a *filepath* parameter.
(b) The Cloud receives the delete request event and deletes the file. The Cloud acknowledges the completion of the request by registering the event into the blockchain through the method Delete().

After these operations, future disputes on the presence or absence of a file can be resolved by looking at the log. If the User requests a file not present in the Cloud, the smart contract automatically verifies whether the User previously requested deletion for that file. If such a request is present, the Cloud correctly deleted that file; if no delete request was logged for that file, there is a SLA violation. Furthermore, if the Cloud is found to have a copy of a file for which a successful and legit delete request is in the blockchain, then the Cloud would again be in violation of SLA since it did not correctly remove the file as requested.

Analysis.

Upon registration of message *b* (Delete), neither the User nor the Cloud can claim that no delete operation was actually requested, since a publicly visible request *a* was stored in the blockchain. Furthermore, neither the User nor the Cloud can claim that the delete operation failed, due to the acknowledgement *b* being stored in the blockchain as well.

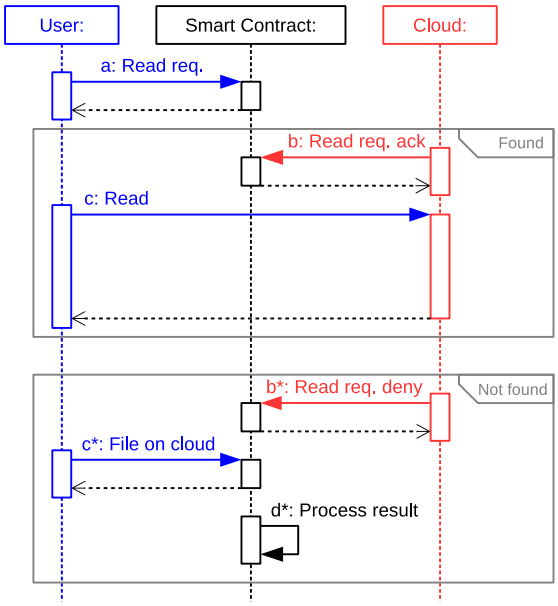

FIGURE 4. UML sequence diagram for the interactions involved in the read request operation. Note the two branches that describe the behavior depending on the file's existence.

3) Read

Figure 4 shows the interactions required to read a file stored in the Cloud. What follows is the description of the Read operation execution.

insane

(a) In order to access a file, the User issues a transaction in order to invoke the ReadRequest() method, providing as input the *filepath*.
(b) To give access to the file, the Cloud invokes ReadRequestAck() and inserts into the blockchain an URL, where the file can be retrieved[3]. This procedure is required to witness that the Cloud has granted access to the User and that the file is valid.
(c) Finally, the User can read the file through the URL provided by the Cloud.

(b) When the file specified in the request is missing, the Cloud responds with a missing message by invoking ReadRequestDeny().
(c) To assess if there is a SLA violation the User executes the FileOnCloud() operation. This method analyzes the smart contract storage that keeps track of the operations and determines if a SLA violation has occurred
(d) The output of this process is stored on the blockchain.

Analysis.

After message *a* (Read request) is stored in the blockchain, the expected outcome can be anticipated by checking the log of all operations involving the same file that have been previously stored there. In fact, the User can not held the Cloud responsible for the unavailability of a file that the User did not upload, since the Cloud can simply point to the lack of a matching Upload operation on the publicly readable log. Similarly, the Cloud can not held the User responsible for the unavailability of a file that must indeed be present, since the User can point to a previous Upload operation not followed by any deletion request on the public ledger. The basic principle is that the history of all interactions is permanently stored in the blockchain in a tamper-resistant way, so anyone can verify the availability (or lack of) of every file at any point in time by simply "playing back" the log.

B. OFF-CHAIN MONITORING

At each execution of a method in the CloudSLA smart contract, some operational and file integrity checks are autonomously executed before the method can go through with its implemented behavior. For instance, ReadRequest() checks that a file has already been uploaded before instantiating the request, and UploadTransferAck() checks the file integrity. However, the CloudSLA smart contract cannot directly read a file from the Cloud using the URL inserted into the blockchain. This represents an off-chain operation where the Monitor comes into play. The Monitor is an oracle fetching data from, and providing data to, the blockchain. We focus on three different scenarios where the Monitor is needed:

Case 1 After a delete operation, the Monitor can check if the Cloud still has a copy of the file that was requested for deletion;

[3] Also in this case, hash masking can be used to provide privacy.

Case 2 After a read operation, the Monitor can check if the Cloud has corrupted the content of the file, i.e., if the digest stored during the upload is different from the digest of the data read back using a read operation;

Case 3 After a read operation, the Monitor can check if the Cloud replies with a "file not found" error, whereas the file should be present because no deletion has been requested.

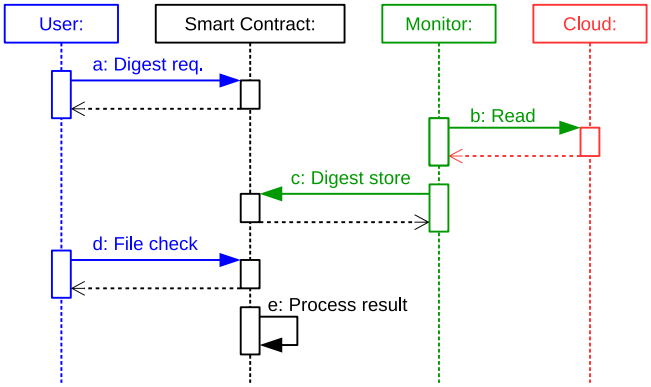

FIGURE 5. UML sequence diagram for the interactions involved with a monitoring request operation.

Figure 5 shows the behavior of the involved entities when the User asks the Monitor to check a file. What follows is the description of the Monitor check operation.

(a) First of all, the User requests to start a check operation using the smart contract method DigestRequest(). This operation registers the request to fetch a file from the Cloud and obtain its digest in the blockchain. It takes as input the URL of the file.

(b) Once the Monitor receives the transaction, including the digest request event from the blockchain, it can accept such a request by reading the file from the Cloud using the provided URL.

(c) Once the Monitor reads the file and obtains its hash digest, it can store the digest in the blockchain through the DigestStore() method.

(d) Once the User receives the transaction, including the Monitor's digest store event from the blockchain, it can invoke FileCheck() with a new transaction.

(e) Such FileCheck() method triggers some processing in which the Monitor's file digest is retrieved and compared with the hash digest of the original file, stored earlier by the Cloud during an upload operation. If the two digests are different, there has been a violation of the SLA. This violation falls into one of the three cases presented above. Consequently, depending on the case, it may incur into a sanction, automatically paid by the Cloud provider through the smart contract (more details in the next subsection).

Analysis.

Similarly to the Upload and Delete request, the expected result of a monitoring operation can be computed by anyone that has read access to the blockchain by playing back the sequence of operations involving the file under consideration. The Cloud can not fabricate a fictitious Upload or Delete operation on behalf of the User, since it is assumed that the User's credentials required to sign a transaction on the blockchain are private. For the same reason, the User can not fabricate a file Upload or Delete operation that did not happen (the User can not sign a blockchain transaction without the Cloud credentials). Finally, the Monitor can not alter the result of a Digest Store operation (message c), e.g., by asserting that the content of a file is different from the expected one, or that a file that should be there is not available, since any deviation from the expected result can be independently verified by anyone from the file hash that is included within the previous file operations stored on the blockchain.

C. AUTOMATIC CREDITS SETTLEMENT

The distributed computation feature embodied by smart contracts enables us to include an automatic credits settlement mechanism into each operation in response to SLA violations. In particular, several blockchain implementations enable the creation of multiple second-layer assets. These assets might represent different means of value exchange between blockchain Users (an example being ERC20 tokens [32]). In the case of a SLA, these assets can automatically handle situations in which the Cloud is liable to the User due to a violation. Some "credits" can be moved to the User account directly on the chain and then redeemed for paying Cloud services. Thus, upon SLA violations, credits are directly moved to the User. For instance, when, after a read operation, the Cloud has lost the file (Figure 4), then after the process is stored on the blockchain (arrow d in the same figure), another smart contract method is invoked to move credits from the Cloud to the User. For each type of violation, the number of credits can be set up through the SLA during the initialization phase.

D. THREAT MODEL AND LIMITATIONS

The solution above has been designed to fulfill the requirements of a specific threat model that will be briefly described in this section. As usual, each threat model has limitations that need to be carefully considered.

The entities that we consider in our security analysis are 1) the User, 2) the Cloud, 3) the CloudSLA smart contract, and 4) the Monitor (i.e., the oracle).

In the proposed solution, the User and the Cloud can be malicious. They may operate in a way that does not conform to the SLA without paying for the prescribed compensations. However, this malicious behavior is cushioned by automatic penalty payments in smart contracts. On the other hand, both the CloudSLA smart contract and the Monitor must be trusted by both the User and the Cloud provider. Trusting the smart contract should not be a problem since its source code is available for review, and the contract itself runs on a blockchain that both parties can assess. Vulnerabilities can

be found in the smart contract and on the blockchain, but this is common to all solutions that are based on smart contracts and blockchains.

The situation is different when we consider the oracle that, in the current proposal, is a centralized external service. This component is mandatory and needs to be trusted by the parties (without the ability to perform a preliminary validation such as the one that can be done for the smart contract). To address this concern, we are working on an extension of this solution that involves a decentralized oracle instead of a centralized one (see Section VII).

Even assuming the trust structure we described above, further limitations must be considered. First of all, the proposed architecture can not prevent the possibility for the Cloud to create unauthorized copies of files. This should not be an issue if the User adequately encrypts the content before uploading it to the Cloud (as mandated by the proposed solution) and securely manages the encryption keys. Our Monitor implementation can detect this behavior if the Cloud still exposes the copy to the public. Another relevant aspect is that the Cloud could obtain some information on the User behavior (or his data) by performing inference analysis on the stored data and the User interaction patterns. Again, the stored data is encrypted, and if extended privacy is required, many mitigation techniques can be employed [39].

Finally, targeted attacks can be executed to reduce or nullify the availability of both the CloudSLA smart contract and the Monitor. Disrupting the availability of the smart contract would require attacking the whole blockchain. On the other hand, attacking the Monitor is again an issue in the presence of a centralized oracle. In this case, a distributed oracle is preferred. Another possible attack could aim to reduce the smart contract's ability (or the Monitor's ability) to complete some of the operations on the files to trigger improper SLA violations and gain from the related compensations. Many mitigation techniques can be implemented, but they are out of the scope of this paper that aims for a general validation of the proposed solution.

V. SOLIDITY SMART CONTRACT IMPLEMENTATION

This section discusses a prototype implementation of the smart contracts described in Section IV. The implementation is written in Solidity, a language compatible with the Ethereum Virtual Machine (EVM) that runs in the Ethereum public blockchain as well as other public and private blockchains. The source code is available on Github [40].

In the following, we first describe the software design pattern used in our development, the implementation of the main smart contract, and, finally, the implementation of the oracle.

A. PATTERN

In our implementation, we use the *factory pattern* [41], a software pattern for creating several smart contracts working with the same logic. In particular, a single Factory smart contract is deployed, which is in charge of instantiating several "child" contracts with the same code. This enables the creation of several SLAs between different Users and the Cloud. The following components make up our Factory contract (see the Factory contract in Figure 6):

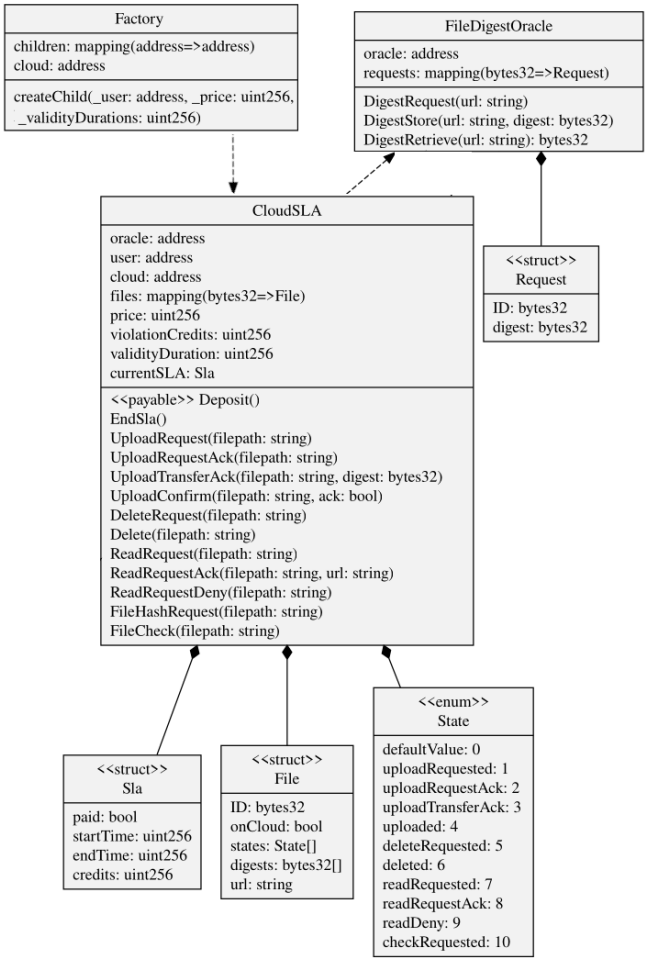

FIGURE 6. UML diagrams of the CloudSLA smart contract.

The Cloud address uniquely represents this entity on-chain.

A mapping of Users' addresses to contracts' addresses representing the "children" of the Factory.

The method createChild() for the creation of the mentioned child smart contracts. Each child contract (described in the next Sub-Section): (i) is attributed to a User using the User's address; (ii) has a price value indicating the User's payment for the Cloud service; (iii) has a SLA validity duration that starts when the child contract is instantiated.

B. SERVICE LEVEL AGREEMENT SMART CONTRACT

The Service Level Agreement Smart Contract implements a set of methods to execute the logging on-chain, namely the log operations for Upload, Delete and Read. Moreover, it implements a set of methods for checking the file integrity or deletion status through an oracle. The following components

make up our CloudSLA contract (see Figure 6):

The contract attributes:

- the addresses that uniquely represent (i) the operational oracle, (ii) the User, and (iii) the Cloud.
- the list of files uploaded to the Cloud storage after the SLA began (*files* mapping in Figure 6). In the smart contract, a File is represented by: (i) a unique filepath by which the Cloud identifies a file in the User's personal storage; (ii) an identifier obtained by hashing the filepath; (iii) its presence on the Cloud storage, i.e., if it has been deleted (*onCloud* boolean in Figure 6); (iv) a state indicating the actual request being executed on it, e.g., *uploadRequested*, *deleted*, *readRequestAck* (*states* array in Figure 6); (v) its hash digests, obtained using the SHA256 function and/or other functions; (vi) URL that is used to access the file externally through the Cloud service.
- the SLA information, namely (i) the price value the User pays for the service, (ii) the credits accumulated after a violation to be paid by the Cloud, (iii) the start time (in UNIX time) and (iv) the end time. Using the same smart contract, a structure named *Sla* is used for instantiating a new SLA contract when the previous one has been terminated.

The methods for performing the Upload operation logging are: (i) UploadRequest() to start the upload taking in input the filepath; (ii) UploadRequestAck() invoked by the Cloud; (iii) UploadTransferAck() invoked by the Cloud after the file has been uploaded and indicating the file digest; (iv) UploadConfirm() invoked by the User.
The methods for performing the Delete operation logging are: (i) the DeleteRequest() to delete the file indicating its path; (ii) Delete() to provide a confirmation by the Cloud.
The methods for performing the Read operation logging are: (i) ReadRequest() to read the file indicating its path; (ii) ReadRequestAck() invoked by the Cloud indicating the file URL; (iii) ReadRequestDeny() invoked by the Cloud indicating the file is not present.
The methods for invoking a file check through a Monitor are: (i) FileHashRequest() method requests a check to the oracle smart contract indicating the filepath and the file URL; (ii) the FileCheck() method executes a logic after the oracle replied to the request, i.e., increment the credit value if a violation, corrupted or undeleted file, has been detected.
Other methods for ending the SLA or enabling the User to deposit an amount of the blockchain currency equal to the price value for starting a new SLA. When a SLA is terminated, the credit amount of currency will be transferred to the User, while the remaining price amount will be transferred to the Cloud.

C. FILE DIGEST ORACLE

The Monitor operation is implemented using an inbound Oracle pattern. Generally, it can be seen as a unique smart contract that receives requests on-chain and an off-chain software component that listens to them and injects data into the blockchain. In this particular instance, the injected data consists of file hash digests. The following components make up our File Digest Oracle contract:

The Oracle provider (i.e., the Monitor) address uniquely represents this entity on-chain.
A list of requests, where each request includes a unique id (obtained by hashing the file URL) and the expected file hash digest.
The DigestRequest() method for making the request, invoked by the FileHashRequest() method on the CloudSLA contract.
The method DigestStore() invoked by the Monitor to store the file hash digest that has been obtained by reading the file at the URL indicated.
The method DigestRetrieve() for obtaining the digest stored by the Monitor, invoked by the FileCheck() method on the CloudSLA contract.

VI. PERFORMANCE EVALUATION

We now evaluate the performance of a prototype implementation of the smart contract described in Section V. All smart contracts have been deployed and tested on three different blockchain environments. The source code and the raw results for the experiments described below are available on GitHub [40].

We assume that the system would be deployed onto a permissioned blockchain. The reason is that public blockchains, such as Ethereum, have known scalability issues [42] that make them unlikely to handle the high request rate that a Cloud demands. However, any EVM-based blockchain can run our implementation, even if the blockchain is permissionless.

A. CONFIGURATION SETUP

Our tests have been executed on three permissioned blockchain environments, all supporting the Ethereum protocol and thus allowing the execution of smart contracts compiled using Solidity. The first two environments are based on the ConsenSys Quorum permissioned blockchain, while the third is based on the Polygon framework for building Ethereum-compatible blockchains. In the following we present the configuration setup.

The general configuration setup of the three permissioned blockchains used in the performance evaluation is as follows:

Each blockchain network is independently run on a server with an Intel i7 CPU (12 physical execution cores) and equipped with 16 GB of DDR4 RAM.
Four validator blockchain nodes are deployed to create a base network. Each node executes one of the consensus mechanisms described above. The parameters for such

a mechanism are configured using the recommended values (see [43]).
One non-validator node is used to expose the APIs for external clients to interact with the blockchain.
Several user nodes are created to interact with the network.
The blockchain "gas limit" is set to 16 234 336. In Ethereum, the *gas* is a unit that measures the amount of computational effort needed to execute operations. The gas limit indicates the maximum amount of gas for a block [42].

In the following, we describe the configuration setup for the three permissioned blockchain environments.

1) GoQuorum

ConsenSys Quorum[4] is an open-source protocol for building Ethereum-compatible environments for enterprises. It is composed of a suite of different technologies that include GoQuorum[5]. This software is a fork of the Ethereum node implementation written in the Go programming language [44], with some enhancements in terms of (i) privacy, i.e., private transactions and private contracts; (ii) consensus mechanisms, i.e., QBFT, Raft, and others; (iii) peer authorization, i.e., access to the network; (iv) account management; and (v) performance. As far as consensus mechanisms are concerned, we tested the following:

Istanbul BFT (IBFT) [45], a Byzantine Fault-Tolerant (BFT) consensus algorithm in which each block requires multiple rounds of voting by a set of validators (> 66%), recorded as a collection of signatures on the block;
QBFT [46], a BFT consensus algorithm similar to IBFT. The key difference between QBFT and IBFT is that the validators take turns creating the next block within a non-randomized dynamic validator set. This is considered an improvement on IBFT's security properties;
Raft [47], a Crash Fault Tolerant (CFT) consensus mechanism in which the leader is always assumed to act correctly (honestly) and, whenever the leader crashes, the rest of the network automatically elects a new one.

2) Hyperledger Besu

Hyperledger Besu[6] is an open-source Ethereum client written in Java. It is included in the suite of technologies of Quorum for building permissioned networks; however, it can be run on the Ethereum public blockchain too. Besu includes several consensus algorithms and has comprehensive authorization schemes designed for enterprise environments. As far as consensus mechanisms are concerned, these are the ones tested:

IBFT, described above;
QBFT, described above;
Clique [48] a consensus algorithm that, similarly to IBFT, uses digital signatures to seal the blocks but sacrifices consistency (a fork can happen) for better availability and faster block generation.

3) Polygon

Polygon[7] is a protocol and a framework for building second-layer solutions on top of the Ethereum blockchain. Polygon is used to bootstrap new blockchains while providing full compatibility with Ethereum smart contracts and transactions. The difference with the previous two Quorum solutions is that Polygon blockchains also support communication with multiple blockchain networks, enabling the transfer of ERC-20 and ERC-721 tokens through a bridge. As far as consensus mechanisms are concerned, these are the ones tested:

IBFT, described above;
Proof of Stake (PoS) [49], a consensus algorithm in which each block validator is required to prove possession of a certain amount of cryptocurrency.

B. EXPERIMENTATION PROCEDURE

The experimentation procedure consists of the repeated execution of each smart contract operation described in Section IV-A, so that average performance measures can be computed. For the Read operation, we considered both the case where the requested file is found and the case in which the file is not found.

Specifically, we considered the following interactions:

Read_found: a Read operation that successfully finds the file invokes the ReadRequest() and ReadRequestAck() methods, in that order.
Read_not_found: a Read operation that does **not** find the file invokes the ReadRequest() and ReadRequestDeny() methods.
Upload: the Upload operation invokes the UploadRequest(), UploadRequestAck() and UploadTransferAck() in order.
Delete: the Delete operation invokes the DeleteRequest() and Delete() methods.
Monitor_check: the Monitor's operation for checking a corrupted file invokes the FileHashRequest(), (Oracle smart contract) DigestStore() and FileCheck() methods.

A steady-state simulation consisting of two phases was performed for each consensus mechanism of each blockchain, during testing:

Transient phase: To compute the distribution mean, we conducted tests on the *Upload* operation. We measured the *mean response time* of each *Upload* request, which includes all the interactions with the blockchain, as described in Section VI-B. We generated a stream of *Upload* requests with an exponentially distributed inter-request time with a rate of $\lambda = \{0.5, 1, 2\}$ req/s. The

[4]https://consensys.net/quorum/, accessed 2022-10-04
[5]https://github.com/ConsenSys/quorum, accessed 2022-10-04
[6]https://besu.hyperledger.org/en/stable/, accessed 2022-10-04
[7]https://polygon.technology/, accessed 2022-10-04

response time of the n-th operation is shown in Figure 7, where each data point is the average of 5 independent executions of the entire sequence, which lasted for 200s. *Steady-state phase*: The steady-state phase involved testing all operations for the computation of metrics over subsequent runs. Each operation was considered in isolation over a period of 600s with a request rate of $\in \{0.5, 1, 2\}$ req/s. Each test was repeated 5 times for each parameter combination, and each data point represents the average response time of all executions. The metrics of interest included (i) response time, (ii) throughput, and (iii) error rate, which is the fraction (percentage) of operations that could not be completed successfully. The most common causes of errors were (HTTP) 503 Service Unavailable errors, indicating that some services were overloaded.

To evaluate the performance of different blockchain/consensus combinations, we generate a stream of requests with exponentially distributed inter-request time, that is, the time between successive requests has probability density function $f(x) = e^{-x}$. The average time between successive requests is $1/$; we select $\in \{0.5, 1, 2\}$, where $= 2$ req/s is near the maximum load that our testbed can sustain (see below).

C. RESULTS

1) Transient analysis.

We start by analyzing the transient phase, i.e., the warm-up period where a stream of *Upload* operations is injected into an initially idle system. The upload operation was chosen because it involves several interactions and is more likely to stress the system. All blockchain/consensus and arrival rate combinations reach a steady-state behavior after about 30 operations. Polygon shows a lower response time (2.2s) than both Besu (12s) and GoQuorum (13s). We also observe that the response time is not significantly influenced by the request rate .

2) Steady-state analysis.

We now analyze the steady-state performance measures for each operation listed in Section VI-B and blockchain/consensus combination. Figures 8, 9, 10, 11, and 12 show the results for each operation. We use the "box and whisker" plot to report the minimum and maximum values (lower and upper whisker), the lower and upper quartile (box), and the average over 5 runs (orange line) for the metric taken into consideration. Each column of histograms plots the results for the increasing values of the request rate $\in \{0.5, 1, 2\}$.

Discussion. We observe that the response time is more or less independent from the arrival rate of requests; interestingly, the response time is also more or less the same for all types of requests once the blockchain/consensus combination has been chosen. The latter is somewhat surprising since we know from Section V that different types of operation trigger different interactions among the involved entities.

TABLE 1. Gas usage for different operations/Smart Contract combinations.

Smart Contract	Operation	Gas Usage
Factory	deploy()	4 051 597
	createChild()	3 403 924
CloudSLA	UploadRequest()	114 379
	UploadRequestAck()	45 152
	UploadTransferAck()	115 319
	ReadRequest()	47 148
	ReadRequestAck()	68 442
	ReadRequestDeny()	75 996
	DeleteRequest()	47 237
	Delete()	36 174
	FileHashRequest()	73 143
	FileCheck()	88 861
	FileCheck() (corrupted)	74 484
FileDigestOracle	deploy()	617 595
	DigestStore()	52 195

On the other hand, the error rate is greatly influenced by . In particular, few errors are reported when $= 0.5$, while with $= 2$, many blockchain/consensus combinations result in 2% and 8% operations failing. This is why we have chosen $= 2$ req/s as the maximum request rate our testbed can sustain. Again, the error rate is more or less independent of the operation.

As the middle row shows, although the throughput is the same, response times vary greatly depending on the blockchain used. Polygon IBFT, Polygon PoS, and GoQuorum Raft have a lower response time. Polygon's configurations have a latency between 1.7 and 2.8 for all operations, while GoQuorum's configuration always responds with an average latency of about 0.6s. The three Besu configurations have a fluctuating behavior, with latencies between 7.7 and 13 seconds, while GoQuorum's IBFT and QBFT have the worst performances with a maximum latency of 16.2 seconds.

As expected, the two operations that need more time to be completed are the *Upload* and *Monitor_check* since these involve three smart contract methods instead of only two, as the other operations do.

3) Gas usage.

Table 1 shows the gas usage of our implementation. The Factory contract has the methods with the highest gas usage; however, these methods are invoked rarely. The deploy() factory method is called only once with a gas usage of 4000k, while createChild() is invoked for each instance of a new CloudSLA contract with a gas usage of 3400k. The CloudSLA contract includes most of the methods invoked more frequently, with gas usage ranging from 36k to 115k. These methods have a more reasonable gas usage and favor a feasible usage by the User. The ReadRequest() method is likely to be the most frequently used in practice and has the lowest gas usage, which is desirable indeed. Finally, the deploy() method of FileDigestOracle is invoked only once per Oracle with 600k gas units, while DigestStore() gas

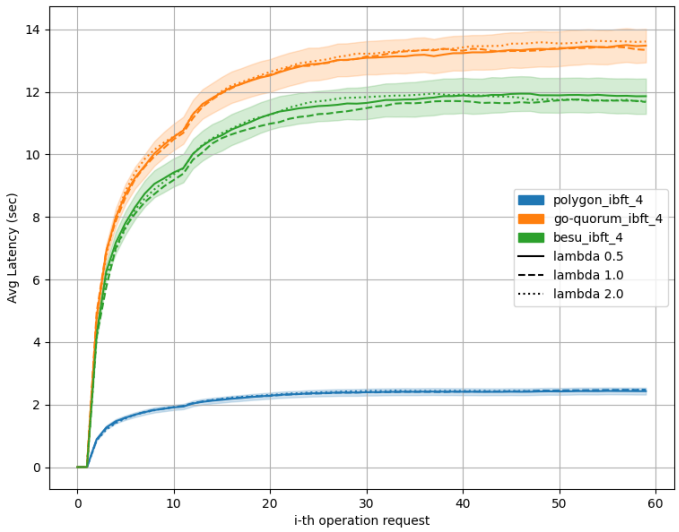

FIGURE 7. Performance of the transient phase; a stream of requests is injected into an initially idle system and the average latency of the i-th request is plotted. All combinations of blockchain/consensus and inter-arrival rate are considered. For $= 0{:}5$ curves are plot together with bands that indicate the confidence interval at 95%

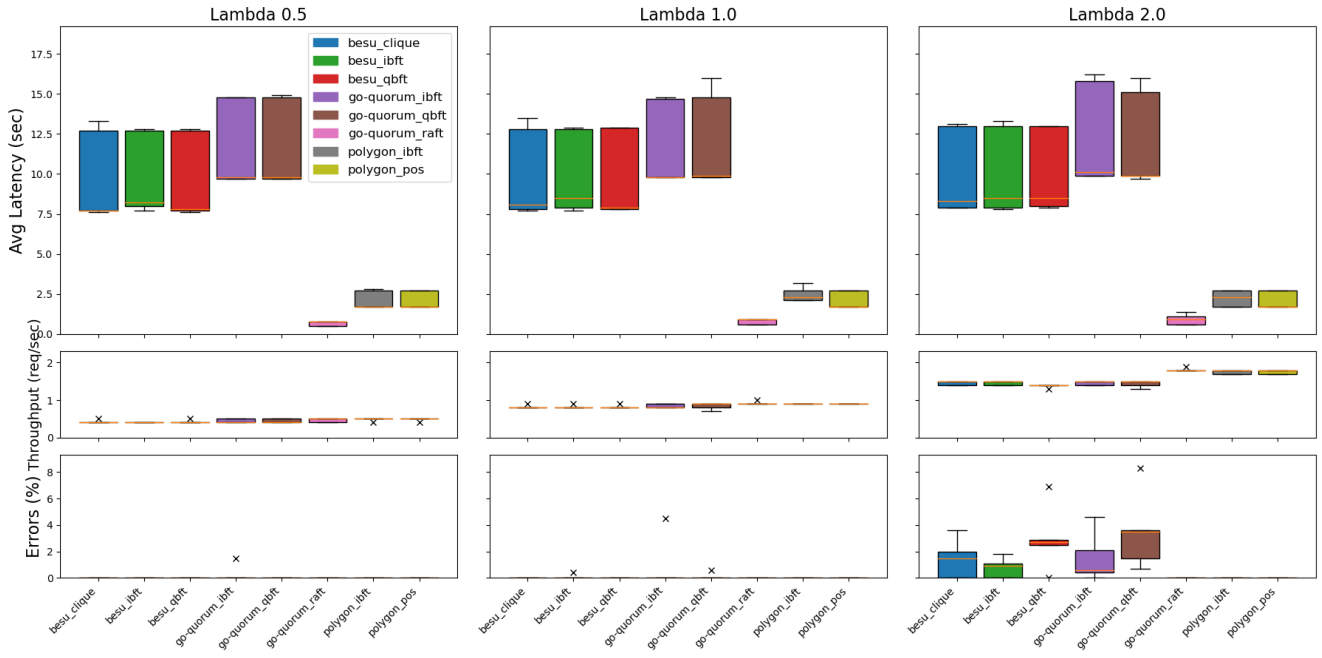

FIGURE 8. Steady-state performance measures for the *Read_found* operation.

usage is in the same order as gas usage of the CloudSLA contract methods.

VII. DISCUSSION

In this section we discuss two issues that are central to our proposal: (i) the issue of efficiency in a mixed cloud/blockchain environment; (ii) the issue of Monitor centralization (i.e., centralized vs. decentralized oracle).

Overall, the results discussed in Section VI show that Polygon and GoQuorum Raft perform better. Indeed, they achieve zero error rate and a reasonable latency, that seems not to be influenced by the inter-arrival rate , i.e., they are more scalable. However, evidence shows that current incarnations of blockchain technologies might not provide response times low enough to support a large number of concurrent customers. Additionally, transaction fees might

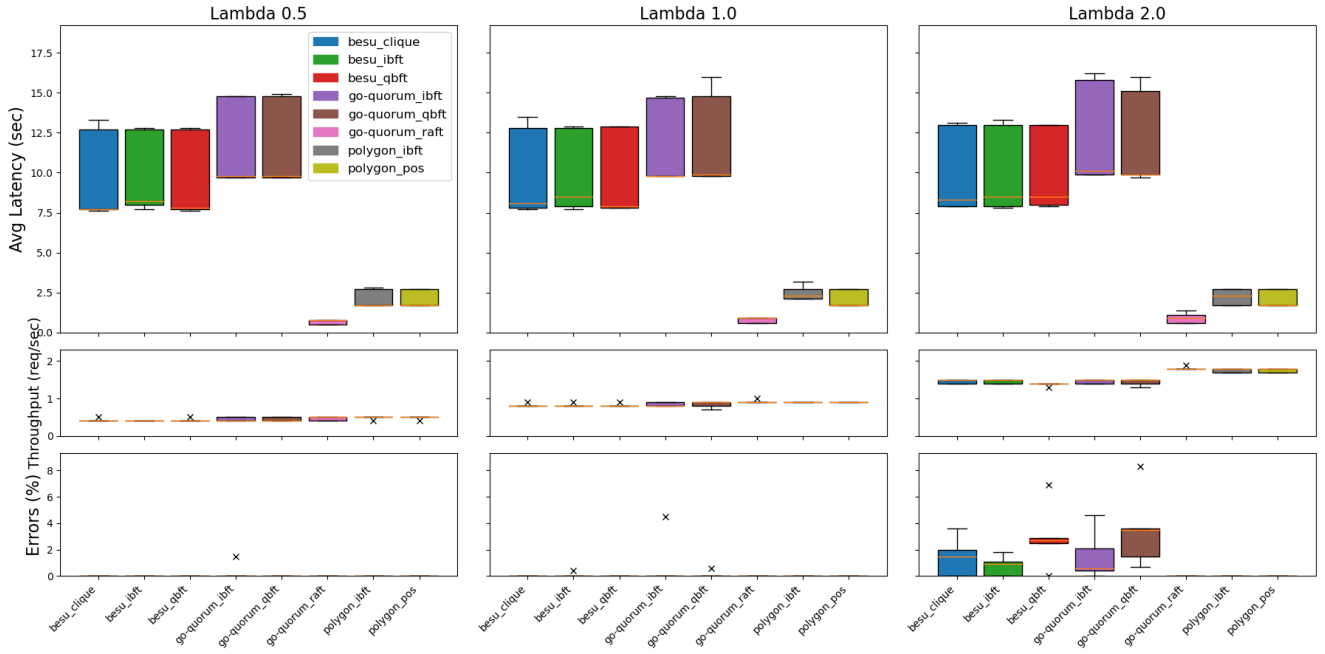

FIGURE 9. Steady-state performance measures for the *Read_not_found* operation.

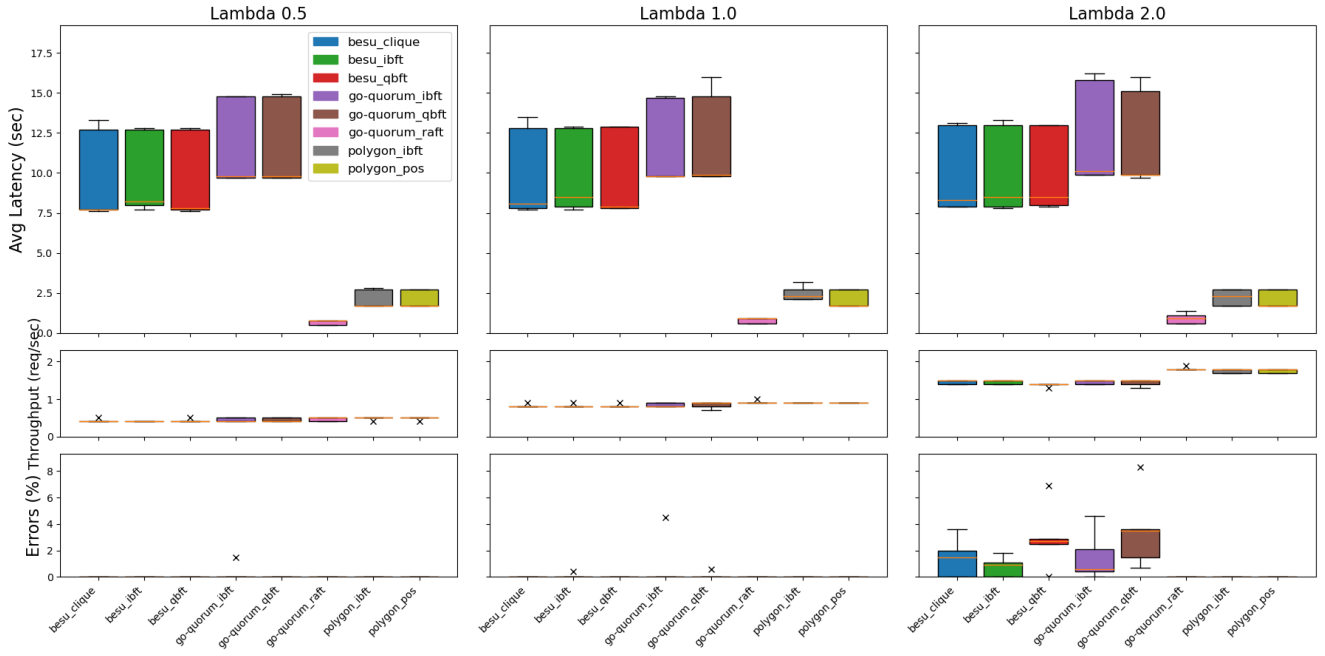

FIGURE 10. Steady-state performance measures for the *Upload* operation.

represent an economic disincentive to the above-mentioned approach, even if gas usage is low for frequent operations (see Section VI-C3). Thus, the choice of which blockchain technology to use remains a significant problem that needs to be addressed in future research. Indeed, it is possible that traditional Ethereum-like blockchain solutions are not the most appropriate in this context. Other approaches are based on a different, more scalable structure for representing the distributed ledger and removing fees, such as IOTA [32]. Indeed, a permissioned blockchain would have the advantage of being more efficient, scalable, and only accessible by a dedicated group of entities who have the eligibility to join it.

The second issue concerns the centralization of the Monitor. The oracle component that implements the Monitor is a standalone module that can be de-centralized. A decentralized oracle allows both the User and the Cloud to put

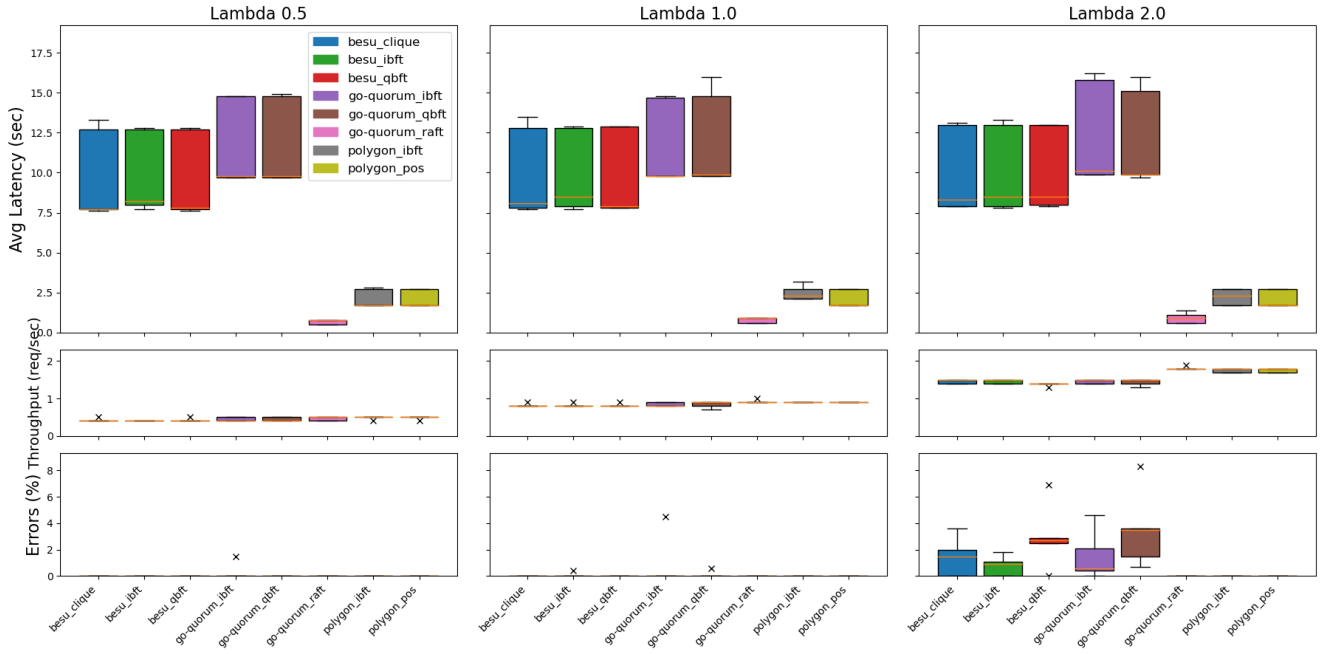

FIGURE 11. Steady-state performance measures for the *Delete* operation.

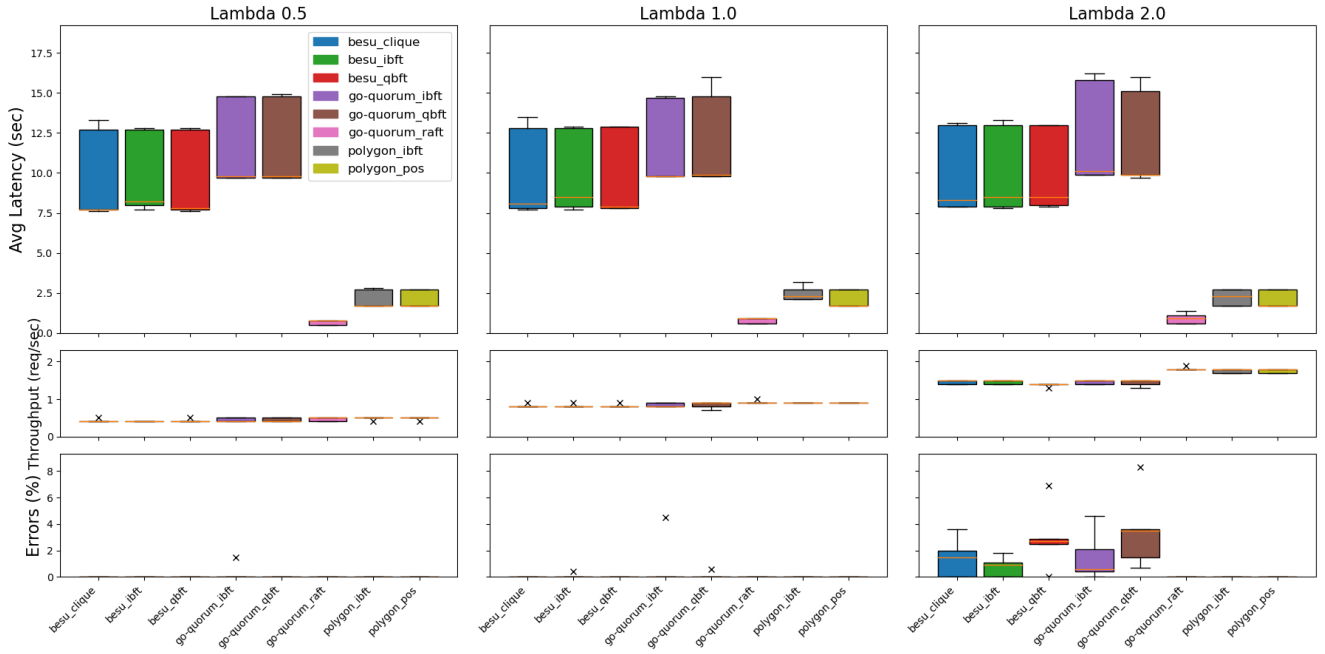

FIGURE 12. Steady-state performance measures for the *Monitor_check* operation.

their trust in a group of coordinated oracles instead of a single Monitor entity. In practice, a pool of monitors reduces the likelihood of incurring in a malicious third-party oracle while, at the same, making it harder for an attacker to compromise a (possibly large) set of servers [50]. In what follows, we describe a preliminary implementation of such a decentralized version of the Monitor based on Chainlink [51].

Chainlink is a general-purpose framework and infrastructure for providing computational resources to smart contracts. It can transfer tamper-proof data from off-chain sources to on-chain smart contracts. Our preliminary implementation uses the Chainlink decentralized network of oracles to check file hashes to ensure their integrity. This allows the implementation of a Monitor as an unbiased, decentralized entity. The Monitor implementation is an extension of a base Chainlink node. This extension adds the

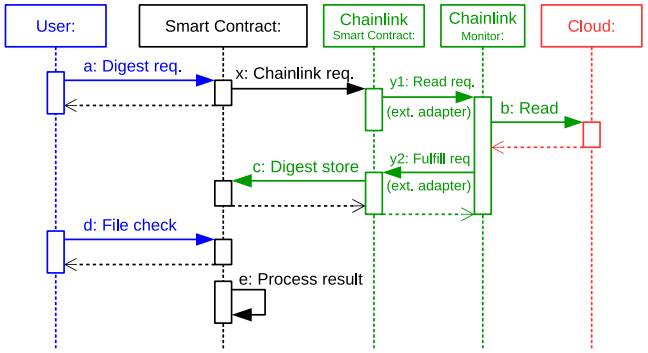

FIGURE 13. UML sequence diagram for the interactions involved with a decentralized monitor.

Monitor operations described above to a Chainlink node software. The prototype CloudSLA smart contract then "invokes" the decentralized oracle network and receives a reply from Chainlink nodes that implemented the extension.

In order to describe how the procedure works, we use a modified version of Figure 5. In Figure 13 we can see how the Monitor has been split into two entities, one on-chain, i.e., the Chainlink smart contract, and one off-chain, i.e., the Chainlink monitor node. Once the User invokes the Digest request operation (a), the CloudSLA smart contract invokes the Chainlink smart contract with a new Chainlink request (x). Upon execution of this function, the Chainlink smart contract emits a Read request event (y1) containing information about the request. This event is crucial, as several off-chain Chainlink Monitor nodes monitor it. This Chainlink smart contract, indeed, mainly provides an on-chain interface to the Chainlink decentralized infrastructure. The off-chain Chainlink monitor node is responsible for listening for events emitted, and once it detects a request, it uses the data emitted to perform a "job". This requested job is the same as before, i.e., the Read operation (b), but now executed in a decentralized way as many nodes can reply and get an automatic reward (through the Chainlink smart contract). Moreover, the Chainlink decentralized infrastructure verifies the process because all the procedure's metadata passes through its network. Finally, a Chainlink monitor node can Fulfill a request (y2) once the job results. The original request contained a callback to be executed upon completion of the job that, in our case, consists of the Digest store operation (c). The monitoring process then terminates, as in the centralized case, with an on-chain File-check (d) and result processing (e). The implementation of our prototype can be found on GitHub [52].

VIII. CONCLUSIONS

In this paper, we explored the use of blockchain technologies to build an unforgeable log for Cloud accountability. The blockchain allows tamper-proof logging of events into a distributed ledger that can then be used to verify if SLAs are violated. We have shown that smart contracts allow automatic identification of responsibilities if SLA violations occur, therefore simplifying the process of settling disputes. As a practical case study, we considered a standard cloud storage service, and, for each possible operation, we described an interaction protocol that logs all relevant events using a smart contract. We developed our smart contract in the Solidity language to allow interoperability since this language is supported by the Ethereum Virtual Machine on which several blockchain platforms are based.

Our implementation was deployed and tested over different blockchain platforms, i.e., GoQuorum, Hyperledger Besu, and Polygon, with different consensus protocols to study response times, error rates, and gas usage. Performance results show that in the configurations we have tested, Polygon and GoQuorum Raft can provide significantly lower response times and negligible (if not null) error rates. The high gas usage for some operations suggests that a permissioned blockchain should be preferred to avoid high fees and offer better scalability.

Along the lines of this work, our future research will be devoted to evaluate the performance of our prototype decentralized oracle and enhance it.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie International Training Network European Joint Doctorate grant agreement No 814177 Law, Science and Technology Joint Doctorate - Rights of Internet of Everything, and from the University of Urbino through the "Bit4Food" research project. We are indebted to Paola Persico, Giosuè Cotugno, Davide Pruscini, Emanuele Sinagra and Valerio Tonelli for their contribution on a preliminary implementation of the system.

REFERENCES

[1] G. D'Angelo, S. Ferretti, and M. Marzolla, "A blockchain-based flight data recorder for cloud accountability," in *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, ser. CryBlock'18. New York, NY, USA: ACM, 2018, pp. 93–98. [Online]. Available: http://doi.acm.org/10.1145/3211933.3211950

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, April 2010.

[3] C. Esposito, A. De Santis, G. Tortora, H. Chang, and K.-K. R. Choo, "Blockchain: A panacea for healthcare cloud-based data security and privacy?" *IEEE Cloud Computing*, vol. 5, no. 1, pp. 31–37, 2018.

[4] G. Ateniese, M. T. Goodrich, V. Lekakis, C. Papamanthou, E. Paraskevas, and R. Tamassia, "Accountable storage," in *Applied Cryptography and Network Security*, D. Gollmann, A. Miyaji, and H. Kikuchi, Eds. Cham: Springer International Publishing, 2017, pp. 623–644.

[5] A. Haeberlen, "A case for the accountable cloud," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 52–57, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773926

[6] R. Neisse, G. Steri, and I. Nai-Fovino, "A blockchain-based approach for data accountability and provenance tracking," in *Proc. 12th Int. Conf. on Availability, Reliability and Security*, ser. ARES '17. ACM, 2017, pp. 14:1–14:10. [Online]. Available: http://doi.acm.org/10.1145/3098954.3098958

[7] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards blockchain-based auditable storage and sharing of iot data," in *Proc. 2017 Cloud Computing Security Workshop*, ser. CCSW '17. ACM, 2017, pp. 45–50. [Online]. Available: http://doi.acm.org/10.1145/3140649.3140656

[8] D. E. Adjepon-Yamoah, "Cloud accountability method: Towards accountable cloud service-level agreements," in *Proceedings of Sixth International Congress on Information and Communication Technology*, X.-S. Yang, S. Sherratt, N. Dey, and A. Joshi, Eds. Singapore: Springer Singapore, 2022, pp. 439–458.

[9] Y. S. Tan, R. K. Ko, and G. Holmes, "Security and data accountability in distributed systems: A provenance survey," in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE, 2013, pp. 1571–1578.

[10] J. Becker and E. Bailey, "A comparison of it governance & control frameworks in cloud computing," in *Twentieth Americas Conference on Information Systems*, 2014.

[11] P. M. Mell and T. Grance, "The NIST definition of Cloud Computing," National Institute of Standards & Technology, Gaithersburg, MD, United States, Tech. Rep. SP 800-145, 2011.

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[13] M. Marzolla, S. Ferretti, and G. D'Angelo, "Dynamic resource provisioning for cloud-based gaming infrastructures," *Comput. Entertain.*, vol. 10, no. 1, pp. 4:1–4:20, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2381876.2381880

[14] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini, "Qos-aware clouds," in *Proc. 2010 IEEE 3rd Int. Conf. on Cloud Computing*, ser. CLOUD '10. IEEE Computer Society, 2010, pp. 321–328. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2010.17

[15] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: Practical accountability for distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 175–188, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294279

[16] A. R. Yumerefendi and J. S. Chase, "Trust but verify: Accountability for network services," in *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, ser. EW 11. ACM, 2004, pp. 175–188. [Online]. Available: http://doi.acm.org/10.1145/1133572.1133585

[17] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 175–188. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855533.1855536

[18] J. Yao, S. Chen, C. Wang, D. Levy, and J. Zic, "Accountability as a service for the cloud," in *2010 IEEE International Conference on Services Computing*, 2010, pp. 81–88.

[19] R. K. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee, "Trustcloud: A framework for accountability and trust in cloud computing," in *2011 IEEE World Congress on Services*, 2011, pp. 584–588.

[20] S. Pearson, "Toward accountability in the cloud," *IEEE Internet Computing*, vol. 15, no. 4, pp. 64–69, 2011.

[21] V. C. Emeakaroha, T. C. Ferreto, M. A. S. Netto, I. Brandic, and C. A. F. De Rose, "Casvid: Application level monitoring for sla violation detection in clouds," in *IEEE 36th Annual Computer Software and Applications Conference*, 2012, pp. 499–508.

[22] Q. Li, Z. Yang, X. Qin, D. Tao, H. Pan, and Y. Huang, "Cbff: A cloud–blockchain fusion framework ensuring data accountability for multi-cloud environments," *Journal of Systems Architecture*, vol. 124, p. 102436, 2022.

[23] M. R. Dorsala, V. Sastry, and S. Chapram, "Blockchain-based solutions for cloud computing: A survey," *Journal of Network and Computer Applications*, vol. 196, p. 103246, 2021.

[24] S. Xie, Z. Zheng, W. Chen, J. Wu, H.-N. Dai, and M. Imran, "Blockchain for cloud exchange: A survey," *Computers and Electrical Engineering*, vol. 81, p. 106526, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790618332750

[25] M. Xie, Y. Yu, R. Chen, H. Li, J. Wei, and Q. Sun, "Accountable outsourcing data storage atop blockchain," *Computer Standards & Interfaces*, vol. 82, p. 103628, 2022.

[26] Q. Li, Z. Yang, X. Qin, D. Tao, H. Pan, and Y. Huang, "Cbff: A cloud–blockchain fusion framework ensuring data accountability for multi-cloud environments," *Journal of Systems Architecture*, vol. 124, p. 102436, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122000339

[27] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 468–477.

[28] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[29] M. Becker and B. Bodó, "Trust in blockchain-based systems," *Internet Policy Review*, vol. 10, no. 2, 2021.

[30] Y. Kurt Peker, X. Rodriguez, J. Ericsson, S. J. Lee, and A. J. Perez, "A cost analysis of internet of things sensor data storage on blockchain via smart contracts," *Electronics*, vol. 9, no. 2, p. 244, 2020.

[31] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[32] M. Zichichi, S. Ferretti, and G. D'Angelo, "A framework based on distributed ledger technologies for data management and services in intelligent transportation systems," *IEEE Access*, vol. 8, pp. 100 384–100 402, 2020.

[33] S. Popov, "The tangle," *White paper*, vol. 1, no. 3, 2018.

[34] P. De Filippi, C. Wray, and G. Sileno, "Smart contracts," *Internet Policy Review*, vol. 10, no. 2, 2021.

[35] B. Liu, P. Szalachowski, and J. Zhou, "A first look into defi oracles," in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 2021, pp. 39–48.

[36] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proc. of the 26th International Conference on World Wide Web*, ser. WWW '17, 2017, pp. 469–478. [Online]. Available: https://doi.org/10.1145/3038912.3052649

[37] G. Kesidis, B. Urgaonkar, N. Nasiriani, and C. Wang, "Neutrality in future public clouds: Implications and challenges," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 90–95. [Online]. Available: http://dl.acm.org/citation.cfm?id=3027041.3027056

[38] R. Merkle and M. Hellman, "Hiding information and signatures in trapdoor knapsacks," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 525–530, 1978.

[39] P. Sun, "Security and privacy protection in cloud computing: Discussions and challenges," *Journal of Network and Computer Applications*, vol. 160, p. 102642, 2020.

[40] P. Persico, D. Pruscini, G. Cotugno, and M. Zichichi, "cloud-chain," 2022. [Online]. Available: https://github.com/miker83z/cloud-chain

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[42] L. Zhang, B. Lee, Y. Ye, and Y. Qiao, "Evaluation of ethereum end-to-end transaction latency," in *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2021, pp. 1–6.

[43] M. Mazzoni, A. Corradi, and V. Di Nicola, "Performance evaluation of permissioned blockchains for financial applications: The consensys quorum case study," *Blockchain: Research and applications*, vol. 3, no. 1, p. 100026, 2022.

[44] A. Donovan and B. Kernighan, *The Go Programming Language*. Addison-Wesley, 2015.

[45] H. Moniz, "The istanbul bft consensus algorithm," 2020. [Online]. Available: https://arxiv.org/abs/2002.03613

[46] R. Saltini, "QBFT blockchain consensus protocol specification v1," EEA Editor's Draft, 2022, (Accessed 2022-06-29). [Online]. Available: https://entethalliance.github.io/client-spec/qbft_spec.html

[47] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[48] P. Szilágyi, "Eip-225: Clique proof-of-authority consensus protocol," Ethereum Improvement Proposals, no. 225, Mar. 2017, (Accessed on 2022-06-29). [Online]. Available: https://eips.ethereum.org/EIPS/eip-225

[49] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, no. 1, 2012.

[50] L. Ma, K. Kaneko, S. Sharma, and K. Sakurai, "Reliable decentralized oracle with mechanisms for verification and disputation," in *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE, 2019, pp. 346–352.

[51] L. Breidenbach, C. Cachin, B. Chan, A. Coventry, S. Ellis, A. Juels, F. Koushanfar, A. Miller, B. Magauran, D. Moroz *et al.*, "Chainlink 2.0: