frontiers | Frontiers in Blockchain

TYPE Original Research
PUBLISHED 21 April 2023
DOI 10.3389/fbloc.2023.1141760

OPEN ACCESS

EDITED BY
Julius Köpke,
University of Klagenfurt, Austria

REVIEWED BY
Volodymyr Shekhovtsov,
University of Klagenfurt, Austria
Felix Härer,
University of Fribourg, Switzerland

*CORRESPONDENCE
Pierluigi Plebani,
pierluigi.plebani@polimi.it

†These authors have contributed equally to this work and share first authorship

RECEIVED 10 January 2023
ACCEPTED 03 April 2023
PUBLISHED 21 April 2023

CITATION
Plebani P, Rossetto D and Tiezzi F (2023), Empowering trusted data sharing for data analytics in a federated environment: A blockchain-based approach.
Front. Blockchain 6:1141760.
doi: 10.3389/fbloc.2023.1141760

Empowering trusted data sharing for data analytics in a federated environment: A blockchain-based approach

Pierluigi Plebani[1*†], Davide Rossetto[1†] and Francesco Tiezzi[2†]

[1]Dipartimento di Elettronica, Informazione e Bioingegneria Politecnico di Milano, Milan, Italy, [2]Dipartimento di Statistica, Informatica, Applicazioni "G. Parenti" (DiSIA), Università degli Studi di Firenze, Florence, Italy

As data analytics is used in business to increase profits, organizations use it to pursue their goals. Even if enterprise data could be already valuable on its own, in many cases, combining it with external data sources would boost the value of the output, making data sharing a need in data analytics. At the same time, organizations are reluctant to share data, as they are scared of disclosing critical information. This calls for solutions that are able to safeguard data holders by regulating how data can be shared to ensure the so-called data sovereignty. This paper focuses on the usage of data lakes as well-established technology across enterprises for data analytics where internal or publicly available data are considered. The goal is to extend data lakes with functionalities that, respecting the data sovereignty, enable a data lake also to be ingested with data shared by other organizations and to share data to external organizations. Notable, the purpose of this work is to face this issue by defining an architecture that, inserted in a federated environment: restricts data access and enables monitoring that the actual usage of data respects the data sovereignty expressed in the policies agreed upon by the involved parties; makes use of Blockchain technology as a means for guaranteeing the traceability of data sharing; and allows for balancing computation movement and data movement. The proposed approach has been applied to a healthcare scenario where several institutions (e.g., hospitals and clinics, research institutes, and medical universities) produce and collect clinical data in local data lakes.

KEYWORDS

data sharing, data sovereignty, blockchain, access control, data lake federation

1 Introduction

Organizations are more and more recognizing data as one of their fundamental assets, not only to increase the efficiency and effectiveness of the internal processes but also to provide a service to other organizations. Much effort has been spent providing methods and tools to collect and store data produced during the execution of the operational/transactional business processes. Thus, these data are now collected in dedicated systems (e.g., data warehouses), ready to be analyzed to offer the management a more detailed and updated snapshot of the organization by computing indicators concerning, among others, process performances and resource usage. More recently, with the Big Data paradigm, the scope is becoming broader. Organizations can now collect more and diverse data from, e.g., social networks, smart devices, and sensors embedded in manufacturing machineries. More

flexible architectures, such as *data lakes* (LaPlante and Sharma, 2016), help to efficiently support the data management while the amount of data increases.

In this context, once the possibility to collect data from internal structures can be taken for granted as the data sources are under the realm of the same organization, it becomes more complex to deal with scenarios in which an organization can take advantage of data owned by another organization. For instance, in a supply chain, a manufacturing company that wants to increase the efficiency of the just-in-time production can find beneficial the possibility to have an updated view on the availability of the raw material of its suppliers. Considering a different context, a hospital that wants to validate some clinical trial results could be interested in applying the developed clinical study to the data of patients stored in another hospital. In these kinds of scenarios, a *federation of the data lakes* can be put in place to enable analytics on data coming from different organizations.

If, on the one hand, many organizations are willing to access data managed by other organizations, on the other hand, most of the time, the organizations that are supposed to provide the data are skeptical. These organizations, indeed, want to keep the so-called *data sovereignty*, i.e., the power to keep in control of the data they generate (European Commission, 2020). Thus, ensuring data sovereignty requires balancing the demand for data sharing and the need to secure privacy. This implies considering that: 1) data can contain personal information, thus the data provisioning is subjected to specific norms that are not easy to implement (e.g., GDPR), 2) once data are shared, the control over these data becomes looser and this could be a problem in case of business-critical data, 3) data management becomes more complex as each data consumer could access to a different portion of data with different rights. Even if data lakes are a well-established technology, there is not yet a standard solution to share data in a trusted way, as it is difficult to both allow the execution of federated queries and guarantee the respect of the data holder's perimeter. This lack of ready solutions may prevent organizations from gaining the most from data analytics.

On this basis, the research question addressed in this paper is: how is it possible to preserve the data sovereignty in a data lake federation setting while enabling the data sharing among the members of such a federation? In this direction, we propose **THROTTLE** (Trusted sHaRing fOr federaTed daTa LakEs), a data sharing mechanism that combines: 1) an attribute-based access control system to guarantee access to the data only to the users with appropriate privileges, 2) a container-based architecture to increase the flexibility of the approach allowing the data to be moved, when possible, where the analysis is more efficient, and 3) a blockchain-based approach to create a trusted environment. In particular, **THROTTLE**:

1. Restricts the access to a data source and enables monitoring that the usage of its data is compliant with the access policies agreed upon between the data provider and data user.
2. Ensures that the operations against the data source are logged according to agreed logging policies, and that all the logs are stored in a tamper-resistant manner.
3. Enables an auditor to verify if the usage of the data respects the access policies.
4. Enables the balance of computation movement and data movement, as data sources are managed in portable containers.

The proposed mechanism is designed to be framed in a data lake federation. The members of such a federation are assumed to have a data lake platform where owned data sets are stored, classified, and managed (Gorelik, 2019). With **THROTTLE**, each member can decide to share—under some agreements—data sets with other members of the same federation for analytical purposes, monitoring whether the agreement is respected. In this way, **THROTTLE** can hold a relevant role in supporting organizations to achieve the data sovereignty.

The rest of this paper is organized as follows. Section 2 presents the **THROTTLE**'s architecture. Section 3 provides implementation details and reports on the application of the **THROTTLE** approach to a case study from the healthcare domain. Section 4 reviews the relevant related literature. Finally, Section 5 concludes the paper and discusses directions for future work.

2 Proposed architecture

A data lake (a.k.a. data lakehouse) is a platform composed of a set of software tools supporting the acquisition, governance, and provisioning of heterogeneous datasets to improve the effectiveness and the efficiency of data analytics, especially when considering the Big Data domain (Gorelik, 2019). Usually, data lakes are seen as platforms specifically deployed for supporting the secondary usage of data for a given organization. Data that feed the data lakes come from internal sources or publicly available sources (e.g., open data repositories). Computational resources can be on-premise or, more commonly, relies on solutions offered by cloud providers as a combination of different services.

Although a data lake offers a good solution for improving data analytics, especially when managed data can have different formats, different structures (if any), and different ways to be ingested (i.e., stream or batch), frictions arise when 1) organizations want to share some of the collected data with other organizations and 2) organizations want to use data of other organizations to improve or extend their analytics. In fact, data sharing implies an agreement specifying, in addition to technical details, the data that are shared and the permitted usage. Moreover, compliance with the agreed terms must be preserved. To make these aspects easier, federation frameworks, like GAIA-X (gaia-x, 2022), that can also be exploited for data lakes, have been proposed to manage data sharing while preserving data sovereignty. Framed in this type of solution, i.e., assuming that a federation of data lakes has been already established and the organizations can specify agreements concerning which data can be shared and the permitted usage, **THROTTLE** is proposed as a solution to ensure the correct access to the data and to monitor the usage. Before introducing the main aspects of the **THROTTLE** architecture, it is important to clarify which are the main actors and which are the initial phases that are assumed to be already performed to define the agreement.

As shown in Figure 1, the three main actors involved in **THROTTLE** are: the data holder, the data user, and the auditor. The *data holder* is a federation member owning a dataset that he/she wants to share. To this purpose, metadata (Gilliland and Baca, 2008)

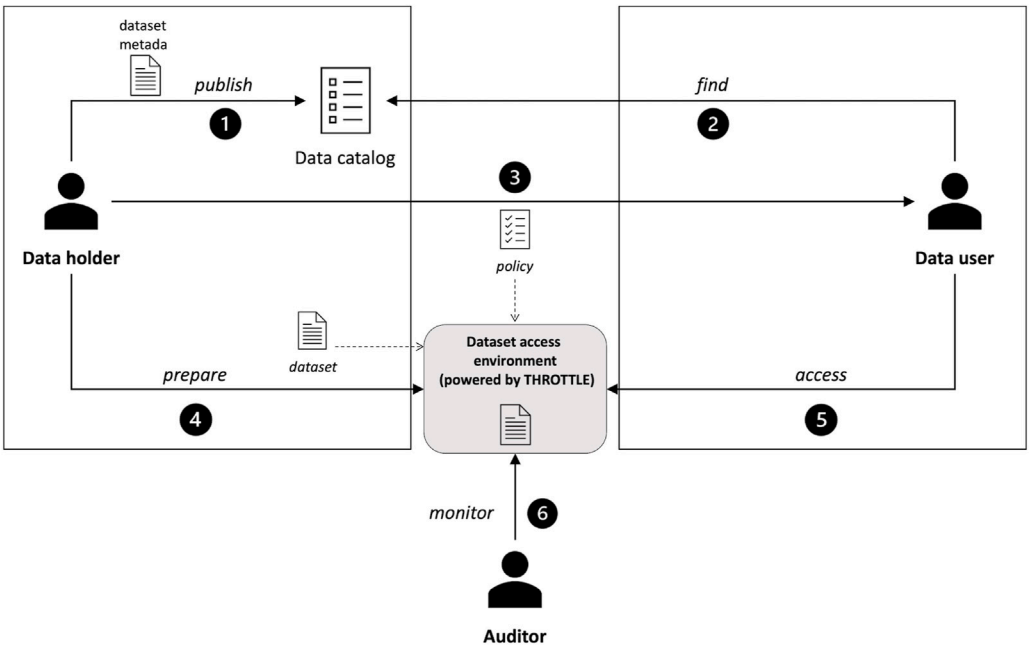

FIGURE 1
THROTTLE's main actors.

associated with the shareable dataset is used to index and publish (step 1) the dataset in a data catalog, which is one of the tools usually included in a data lake (Jahnke et al., 2022). On the other side, the data catalog is used by the *data user* to browse for datasets that can be relevant to the analysis that has to perform (step 2). Metadata can be used as keys to filter out data sources to find the right one. Once found, before the data is usable, the data holder and data user define an agreement to formalize in a *policy* document the terms of use (step 3). These terms could include whether the dataset is completely accessible or only as a portion. In the latter case, if the portion is the result of a selection, i.e., only specific rows can be seen, or a projection, i.e., only some attributes can be accessed. The agreement can also include information concerning where the data can be used, i.e., whether the analysis can be done at the data holder or user side. This decision could depend on privacy reasons, e.g., to avoid data leakage, or performance reasons, e.g., to reduce the quantity of data to be moved (Plebani et al., 2018). Based on the agreement, the data holder prepares the environment to make the dataset available (step 4), and the data user can now access the dataset (step 5). Finally, the *auditor* is a third party in charge of certifying that data have been accessed and used in compliance with the agreement (step 6).

Based on this setting, the goal of **THROTTLE** is to define an architecture to implement the dataset access environment able to 1) enforce this policy in terms of data access and 2) monitor the data policy in terms of data usage, and 3) provide a portable solution that permits to either leave the data at the data holder side where the analysis will be performed, or to move the data to the data user side. In this way, while it will not be possible for the data recipient to access data that are not allowed in the agreement as the involved mechanisms will block any unauthorized access, the control of the usage relies on the generation of a log to store information about how the data are used. Notably, a blockchain is used as a storage element (Tai et al., 2017) for the log. Being available to all the federation members, this information stored in the blockchain can also be accessed by the auditor to check the aforementioned compliance. Indeed, while the access control system guarantees access to the data only to authorized users and the blockchain guarantees the traceability of data sharing, no guarantee is enforced concerning the usage of accessed data; hence, the need for monitoring by auditors. Finally, a container-based approach is used to ensure the portability of the dataset. Containerization, in fact, allows to create portable elements that host data sets along with the functionalities required by **THROTTLE** for logging the accesses. It is worth noticing that the use of containers is not mandatory to create trusted data sharing, but it is convenient to simplify the dynamic deployment of the solution.

Figure 2 shows the overall architecture of **THROTTLE** which is composed of four main elements:

- The eXtended Access Control (XAC) is a component that:
  1. Parses the incoming access request from the data user and takes an access decision on the request in accordance with the policies in place.
  2. Stores the decision on the Blockchain to make it auditable by means of the Persistence Manager.
  3. Builds a container-based image of the Data Analyzer Module (DAM) and creates an instance of this image in the agreed execution environment. The created DAM will contain only the requested data.
  4. Returns to the data user the endpoint of the just instantiated DAM.

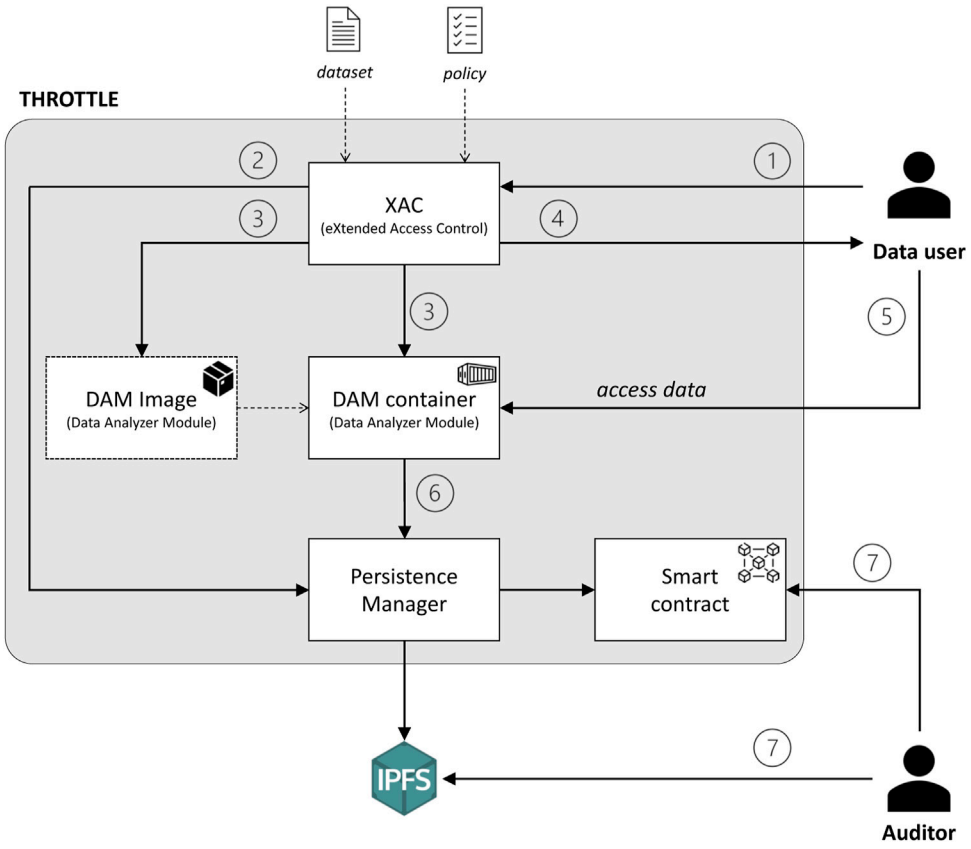

FIGURE 2
THROTTLE's overall architecture.

- The Data Analyzer Module (DAM) is the component running inside the container that:
  5. Exposes an endpoint allowing the data user to access the requested dataset.
  6. Logs access to data according to logging policies specifically defined for this dataset and stores this log by means of the Persistence Manager.
- A Persistence Manager (PM) acting as a gateway to the blockchain when it is needed to store auditable information. This component is also responsible for balancing between on-chain/off-chain storage based on the type and amount of data to be stored. For the off-chain storage, IPFS is assumed to be adopted.
- A Smart Contract (SC) is the on-chain component that enables the system to actually implement the tamper-resistant storage, by registering on-chain the hashes of the logs that are stored off-chain in the IPFS.

The data stored in the blockchain and the IPFS can be used by the auditor (step 7) to verify, through the logs, that the data sovereignty requirements prescribed by the agreed policies are met. Due to the limitations of blockchain in terms of velocity and the high costs when a significant amount of data are stored, the Persistence Manager can decide to not entirely store the log in the blockchain. Conversely, IPFS is used to store the entire log while the Content IDentifier (CID) is stored on the blockchain. Using the hash of the file stored on IPFS as CID, this solution has two advantages: the amount of information stored in the blockchain is proportional to the number of files (regardless of their size) constituting the log, and any modification done on the file log can be detected as it alters the corresponding hash. The combination of these two aspects offers tamper-resistance storage for the logs. Notably, the data stored in the data holder's datasets are not inserted in IPFS because they typically are private data (in our case study, e.g., we consider personal and medical data of patients) that the data holder would like to keep under its control. Summing up, to check if data users violated the agreed access policies, the requirements for auditors consist of just having access to the blockchain and IPFS.

Decoupling the data access into two separate components, i.e., the XAC and the DAM, increases the data access's interoperability and flexibility. In fact, the request submitted in step 1 could include not only information about the data that the user wants to access, but also how the data should be exposed (e.g., SQL-based or REST-based) and where the data should be stored. Thus, the XAC is in charge of analyzing the feasibility of the access and preparing the environment for making the access possible according to the policy through a container that is created on demand. Moreover, only the data that have been requested by the user—which can be a subset of the original dataset—are

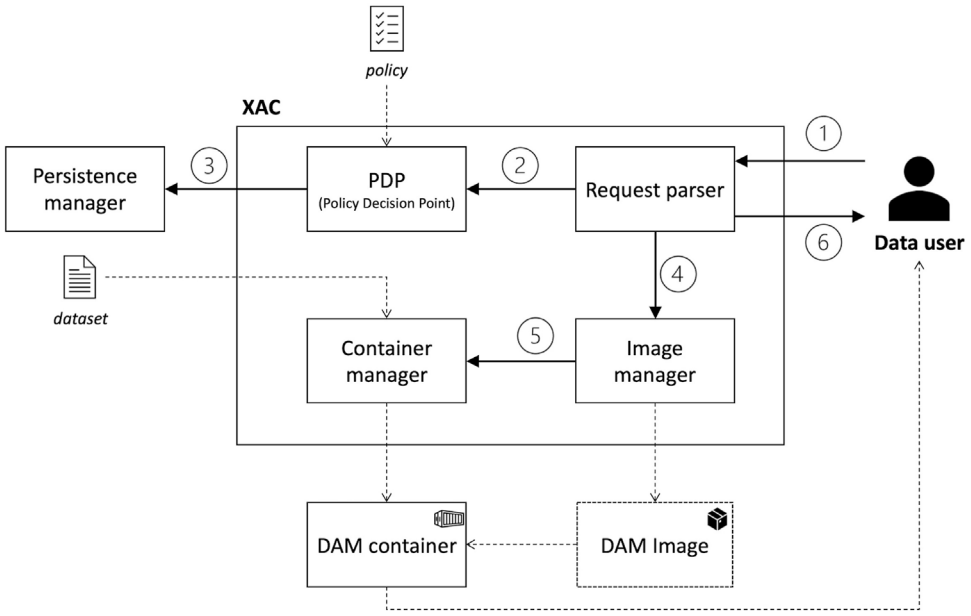

FIGURE 3
Internal structure of the XAC component.

embedded in the container. Thus, the user has no visibility on the complete dataset with a limited effort also at the data holder side. Finally, the adoption of a container-based solution allows the XAC to create a container to host a DAM based on the technology adopted by the infrastructure that will host the container (e.g., x68 vs. ARM), and the container can be deployed either on resources managed by the data holder or the data user, again, with a limited effort.

From a management standpoint, the proposed solution requires to publish in the data catalog only the endpoint of the XAC, regardless of the dataset to be accessed. The real endpoint is produced at runtime through the container. In this way, given a dataset, each user has a specific and personal endpoint to access. Moreover, the data holder can decide to re-deploy the dataset internally without informing the data user.

Based on this overall description, details of the three elements are described in the next paragraphs.

2.1 XAC—eXtended access control

Figure 3 shows the internal structure of the XAC, which is composed of the following modules:

- Request Parser: the front-end component offering an API to enable communication with the data user. This module receives the data request and returns the endpoint to the instantiated DAM container.
- Policy Decision Point (PDP): the module that has access to the policies specifying the terms of usage of a data user to a dataset. Based on this knowledge, the PDP decides whether to allow or deny a data user's request;
- Image Manager: the component responsible for building the container image of a DAM based on the technology as requested by the data user and defined in the policy;
- Container Manager: the component in charge of instantiating a container image of a DAM on the resources as requested by the data user and defined in the policy.

As different solutions are now available to implement container-based solutions (e.g., Docker, Podman), the Image and Container Manager modules offer a technology-independent interface to the other modules in **THROTTLE** to create and instantiate images and containers. Internally, these modules are responsible to translate the received requests to commands that are specific to the selected technology.

Focusing on the interaction among the modules, the inbound request from the data user is handled by the Request Parser (step 1). Notably, the request contains the dataset, the technology through which the dataset is reachable (e.g., SQL, REST, S3, etc.), the action the data user is willing to perform, and the resource onto which the action should be performed. The request is forwarded to the PDP for evaluation (step 2), and the decision is stored on the blockchain through the Persistence Manager (step 3). In case of a negative answer from the PDP, the data user is informed, and the process ends. In case of a positive answer, the Request Parser contacts the Image Manager (step 4), which is in charge of building a container image that exposes an endpoint matching the technology and containing the dataset as decided by the PDP. The instantiation of the container based on the built image is left to the Container Manager (step 5), as the same image might be used in different settings. This could occur when the same dataset, based on the same technology, is

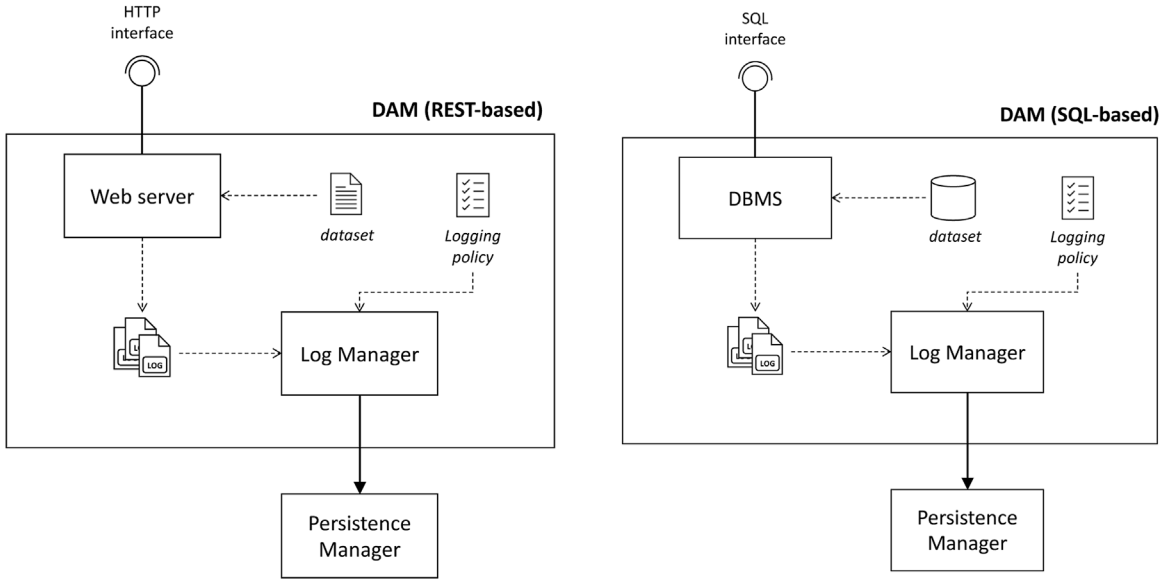

FIGURE 4
Two examples of DAM structures: REST and SQL based.

requested to be deployed on different resources. In this case, the Image Manager immediately forwards the control to the Container Manager. Finally, after the container is created, the XAC returns to the data user the URI of the endpoint exposing the requested data, anywhere it is deployed (step 6).

A malicious user could extract some data directly from the image without running it, therefore bypassing the endpoint. This would lead to data access that would not generate logs. This possibility can be mitigated by performing file system-level logging on data access or encrypting the data at rest, preventing the data user from handling the image itself but making the data holder in charge of the deployment. We leave as future work the investigation of this aspect.

It is worth noticing that the access control system provided by the XAC component has been mainly conceived for dealing with private data that the organization would share to a restricted and well-identified audience. In fact, in general, there is no need to prevent access to publicly-available data. Anyway, a dataset containing both private and public data can be easily managed by our approach: it is sufficient to define access policies for public data that apply to all requests and always grant access. Even if public data are always shared, this mechanism permits keeping track of these accesses via the logging strategies provided by **THROTTLE**. Notably, to make private data public or *vice versa*, it is necessary to change access policies. This change does not affect the access to the dataset previously granted to the data user, who must send a new access request to take advantage of the access right change. Similarly, changes to the data stored by the data holder are not automatically reflected in a dataset previously provided to the data user; again, the latter has to send a new access request. An automatic alignment mechanism could be put in place, but this is out of the scope of this work because it would raise complex issues concerning data consistency.

2.2 Data analytics module

A DAM is a technology-specific component that offers an environment to access a dataset according to the request done by the data user. This component is built on-the-fly by the XAC based on the requirements expressed by the data user and in compliance with the agreed policies. For this reason, DAM implementations are slightly different for each specific technology (e.g., a relational DBMS for SQL, a web server for REST), and we assume that cookbooks specifying typical settings related to the most common technologies are defined. Figure 4 shows two possible configurations of two DAMs that offer, namely, a REST-based and SQL-based interface. Regardless of the specific technology, a DAM always includes two common modules:

- The Log Manager, which is in charge of filtering the logs that are considered to be relevant, according to the logging policy specifically defined for the dataset and the data user's request, ensuring that the relevant logs will be stored in a tamper-resistant storage.
- A connection with the Persistence Manager to store the logs in the tamper-resist ant storage.

Based on this setting, all the created DAMs will rely on the same Persistence Manager to store the information on the blockchain, thus causing a possible bottleneck. For this reason, if the resource that will host the container has enough capacity, it is also possible to have a copy of the Persistence Manager inside the container. This alternative does not affect the system's functionalities due to the distributed and peer-to-peer nature of the blockchain.

To increase the portability of a DAM, which contains the dataset along with the modules to log the accesses and to store the log to the blockchain, a container-related technologies is

adopted. This requires that a template of container configuration file (e.g., Dockerfile) is defined in advance by the same actor offering the dataset. When the dataset is requested and the XAC defines the logging policy, if an image with a logging policy does not exist then a new image is created, otherwise a new container of an image already created with that policy is instantiated The logging policy document is derived from the policy reporting the agreement between the data user and the data holder and specifies the information that must be logged and stored in the blockchain for auditability. To increase the transparency, the logging policy itself could be published on the blockchain, too.

Finally, exploiting the portability of containers, where the DAM container has to be deployed can be decided at run-time. As long as the hosting environment is properly configured to host a container (e.g., in case of using Docker the related deamon is installed and running) the XAC can evaluate the available resources and select which is the best place where to deploy. It is worth noticing that the discussion on how to select the deployment location is out of the scope of this paper.

2.3 Persistence manager

The Persistence Manager is the component in charge of properly optimizing the storage of relevant data. When invoked it buffers the incoming data. Once the number of documents in the buffer reaches a certain threshold or the buffer is flushed, the persistence manager will store all the buffered data on the IPFS. Then, the CID of the data stored on IPFS is written on-chain, by invoking a method of the Smart Contract.

Handling the logs in batches can minimize the number of operations performed on the IPFS and, consequently, on the Smart Contract, reducing the overall cost of execution. A weakness of this approach is that some logs may be lost if the container is forced to shut down while some logs still need to be stored on-chain. This can be addressed by setting the value of 1 as batch size; this particular setting is equivalent to persistently storing the logs one by one (i.e., not buffering the logs). Independently of the batch size, the logs are stored on IPFS as a Merkle DAG. This data structure ensures the immutability of all the nodes; in this way, it is enough to store on-chain the CID of the root node to guarantee that all the nodes are tamper resistant.

It is worth noticing that IPFS is one possible solution to storing a large amount of data off-chain. To use this solution in practice, we assume to exploit one or more dedicated servers. On the other hand, other solutions may be considered, such as Layer 2 technologies (e.g., Polygon[1]); thanks to the modular architecture of **THROTTLE**, this change would imply simply replacing the current Persistent Manager component by another dealing with the alternative technology.

2.4 Smart contract

The Smart Contract is the element of **THROTTLE** responsible for storing on and retrieving from the blockchain the information about the decision taken by the data holder and the usage of the data user. Although a smart contract can be invoked by anyone connected to the blockchain, we assume that the Persistence Manager is the actor in charge of calling the contract to store information, while the Auditor is the one in charge of calling the contract to retrieve information. Notably, the smart contract exposes three methods:

- `storeDecision`: this method is invoked by the Persistence Manager after the policy evaluation step to store the CID of the decision previously stored on the IPFS.
- `storeLog`: this method is invoked by the Persistence Manager in DAM when new logs are stored on IPFS and, consequently, the CID of the root node of the Merkle DAG changes. The Smart Contract only needs to store the last value of CID to allow the retrieval of all the logs.
- `getRequestInfo`: this method is invoked by an auditor to retrieve information about the operations consequent to a request; the method returns the CID of the decision and the CID of the root node of the Merkle DAG containing all the logs. The auditor can then retrieve the documents on IPFS and analyze them.

3 Validation

To validate the feasibility and effectiveness of our proposal, we provided a prototypical implementation of the **THROTTLE** architecture and we applied it to a healthcare case study. Sources of the implementation and instructions for replicating the validation experiments are available on GitHub[2]. We provide technical details about the implementation in this section, illustrating them by means of a scenario of the case study used as a running example.

The language used for implementing the components of the **THROTTLE** architecture is Typescript. We selected this language since it inherits the advantages of the rich ecosystem of Javascript and, in addition, brings the possibility to use types, improving the robustness and readability of the code.

In the following paragraphs, we first introduce the healthcare case study and then describe the implementation of the **THROTTLE** modules.

3.1 Case study

We consider the hereafter described case study related to a healthcare scenario in Europe where the need for data sharing is urgent. Multi-centric clinical trials are studies that involve different hospitals. The collaboration among hospitals allows them to increase the number of patients that can be enrolled for the study, thus increasing the reliability of the result. Nevertheless, since the considered data deal with personal information, they cannot freely flow among the hospitals, but some constraints are required for compliance to the GDPR[3]. For instance, when moving

1 https://polygon.technology/

2 https://github.com/davide94/Data-Sharing-Framework

3 General Data Protection Regulation, a.k.a. GDPR, is a fundamental regulation active in all the countries belonging to the European Union which regulates how personal data must be treated to avoid abuse or data leakage.

10.3389/fbloc.2023.1141760

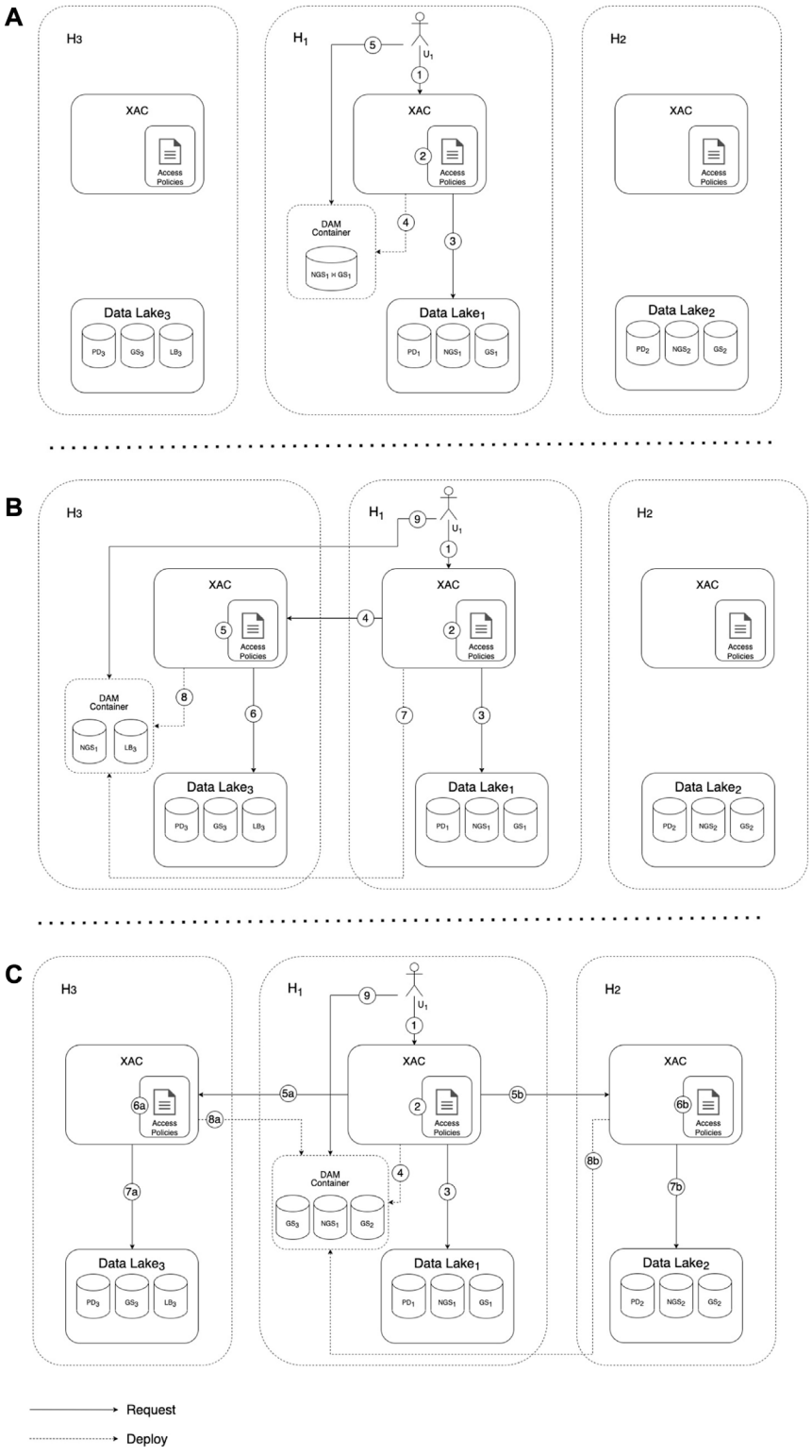

FIGURE 5
Scenarios of the healthcare case study: **(A)** local comparison **(B)** remote comparison **(C)** remote query and local comparison.

data from one hospital to another, all the information that can disclose the patients' identity must be removed. For this reason, anonymization techniques are required. At the same time, only data of patients that have explicitly given consent to use the data for the given trial can be managed. Finally, assuming that the ethical committee that supervises the study has decided to limit the

analysis to patients aged between 18 and 35 years, none of the data about patients aged outside this range can be considered, also in case they gave consent.

More specifically, we consider a clinical trial studying the validity of a new genome sequencer, called Next-Generation Sequencing (NGS), and a new technique, called Liquid Biopsy (LB), to find tumoral markers. With respect to the current approach, which is based on a different sequencing technique, the NGS can find in a single step many tumoral markers, while the LB allows finding tumoral markers with a blood test, thus without requiring invasive surgery.

The goal of the clinical trial is to verify if the NGS and the LB are able at least to find the same tumoral markers that the former sequencing technique was able to find. To this aim, each of the three hospitals involved in this trial, named H_1, H_2, and H_3, have two data sets:

- Personal data about a set of patients. This dataset is named PD_n, where n indicates the hospital.
- Tumoral markers found for each patient by applying the traditional sequencing technique. This dataset represents the golden set for the clinical trial and it is named GS_n, where n again indicates the hospital.

Moreover, due to the limits in the instruments available in the hospitals:

- H_1 and H_2 have the tumoral markers obtained by applying the NGS techniques for their patients. This produces the datasets NGS_1 and NGS_2, respectively.
- H_3 has the tumoral markers obtained with the LB technique, thus producing the dataset LB_3.

In the considered case study, a researcher U_1 of the hospital H_1 wants to validate the results of the NGS technique stored in the dataset NGS_1. To this aim, we consider three different scenarios (depicted in Figures 5A–C, respectively):

[(*a*)] NGS_1 is compared with GS_1. In this case, the two involved datasets belong to the same organization, hence the query is locally executed in H_1. More specifically, once the request sent by U_1 (step 1) is evaluated against the access policies (step 2), the requested data are retrieved from the datasets (step 3), locally deployed (step 4), and finally accessed by U_1 (step 5).
[(*b*)] NGS_1 is compared with LB_3 on H_3 premises. Being too heavy, the LB_3 dataset cannot be moved to H_1, thus it is preferable to move NGS_1 to H_3 and to perform the comparison remotely. This time, the access request is forwarded from H_1 to H_3 (step 4), and both NGS_1 and LB_3 are deployed within H_3 (steps 7 and 8), hence U_1 accesses the data remotely (step 9).
[(*c*)] NGS_1 is compared with GS_2 and GS_3. In this case, a federated query is performed and the data satisfying the request are moved to H_1 for the comparison. Here, the access request is parallelly forwarded from H_1 to H_3 and H_2 (steps 5a-b), resulting on the deployment of all requested datasets in H_1 (steps 4 and 8a-b).

Based on the agreement among the parties, all the actions performed on the datasets must be traced appropriately. The resulting logs must be made available, on request, to an auditor that can check whether the clinical protocol defined by the ethical committee has been respected. We assume that the logs of the decisions and the data accesses can be made publicly available without disclosing sensitive information. If this is not the case, the problem can be addressed by encrypting the data before storing it through a cryptography technique; in this way, only those who are entitled can decrypt and interpret the data.

3.2 eXtended access control

A paramount concern for this component was to expose an API that a variety of clients can consume without forcing them to use a specific technology. Our choice, therefore, was HTTP since, for this purpose, it is the most widely used and well-known standard. Specifically, we expose the HTTP API of the XAC component via a web server implemented with the Express framework.[4]

The data user can send a request by performing an HTTP POST request. The request's body should contain all the information in JSON format. In our running example, considering the scenario depicted in Figure 5A, where the data user wants to compare the datasets NGS_1 and GS_1, the body of the request is as follows:

```
{
  "sender": "H1",
  "technology": "SQL",
  "query": "SELECT * FROM NGS1, GS1 WHERE NGS1.Patient_ID = GS1.Patient_ID"
}
```

In this case, the data user is a member of the hospital H_1, that would like to access via SQL the data resulting from the join of NGS_1 and GS_1.

The XAC component parses the request's payload and stores it in a queue as a task. The id of the created task is returned to the user, who can periodically check the advancement of the request processing by calling a specific endpoint with the obtained task's id. More in detail, the endpoint in place for this purpose can be invoked by a POST request to `/status/:id`, being `id` the identifier of the task.

Asynchronously, a worker checks if there are some pending tasks in the queue. If it finds one, it processes the task as follows. The request, stored in the task, is disposed to the Parser, which produces a request that is formatted according to the input language of the PDP. In our implementation of the **THROTTLE** architecture, the language exploited by the PDP is XACML (OASIS, 2013). This is the OASIS standard language for writing access policies and requests according to the Attribute-Based Access Control (ABAC) model. The ABAC model evaluates the access rules against the attributes of the subjects and objects involved by the access request, without the need to specify individual relationships between each subject and each object. Attribute-based rules are typically hierarchically structured in policies and paired with strategies (i.e., combining algorithms) for resolving possible conflicting authorisation results. More specifically, we have implemented our PDP as an instance of the WSO2 Balana[5] engine, which is an implementation of the

4 https://expressjs.com/

5 https://github.com/wso2/balana

XACML standard. Since Balana is implemented in Java, we could not natively interact with Balana from the Typescript code. Therefore, we have wrapped Balana with a Java web server, which provides a REST interface that simply invokes the Balana method for evaluating the request received as input. This enabled us to interact with the instance of the PDP in a technology-agnostic manner.

A policy evaluated by the PDP in our running example is as follows (for the sake of readability we omitted some details, such as the namespaces):

```
<Policy PolicyId="GS1_policy" RuleCombiningAlgId="permit-overrides">
  <Target>
    <AnyOf>
      <AllOf>
        <Match MatchId="string-is-in">
          <AttributeValue DataType="string">GS1</AttributeValue>
          <AttributeDesignator AttributeId="resource-id" DataType="string" MustBePresent="true"/>
        </Match>
      </AllOf>
    </AnyOf>
  </Target>
  <Rule Effect="Permit" RuleId="H1-permit-rule">
    <Condition>
      <Apply FunctionId="string-equal">
        <AttributeValue DataType="string">H1</AttributeValue>
        <AttributeDesignator AttributeId="subject-id" Category="subject-category:access-subject"
          DataType="string" MustBePresent="true"/>
      </Apply>
    </Condition>
  </Rule>
  <Rule Effect="Deny" RuleId="default-deny"/>
  <ObligationExpressions>
    <ObligationExpression FulfillOn="Permit" ObligationId="logging-policy">
      <AttributeAssignmentExpression AttributeId="logging-policy-id">
        <AttributeValue DataType="string">logging-policy-GS1.xml</AttributeValue>
      </AttributeAssignmentExpression>
    </ObligationExpression>
    <ObligationExpression FulfillOn="Permit" ObligationId="data-locality">
      <AttributeAssignmentExpression AttributeId="data-locality-id">
        <AttributeValue DataType="string">local</AttributeValue>
      </AttributeAssignmentExpression>
    </ObligationExpression>
  </ObligationExpressions>
</Policy>
```

This is the policy regulating access to the GS_1 dataset. Indeed, as specified by its `<Target>` element (lines 2–11), it applies to all requests concerning the resource identifier (line 7) with value `GS1` (line 6). Notably, the value of the `resource-id` attribute is set in the XACML request by the Parser, which extracted this information from the query specified in the JSON request; in our example, this is a multivalued attribute whose evaluation produces a bag containing the strings `GS1` and `NGS1`. The policy contains two rules: the first (lines 12–20) grants access to any subject (line 16) from the hospital `H1` (line 15), while the second (line 21) always forbids access. The decisions returned by these rules are combined by the `permit-overrides` algorithm (line 1), which returns a `permit` decision if the evaluation of at least one rule is `permit`. The second rule is used, in fact, to define `deny` as the default decision, to be returned whenever no rule returns `permit`.[6] Finally, the evaluation of this policy returns two obligations (lines 22–33) whenever the final decision is `permit`. Obligations are additional actions produced by the access control system that must be discharged at the end of the policy evaluation. In this case, a `logging-policy` is generated to keep track of the accesses to the dataset GS_1 (lines 24–26), and the `data-locality` value `local` is returned to indicate that the data within GS_1 must remain local to the data holder's locality (lines 29–31). It is worth noticing that, in the overall implementation of the case study, this policy would be enclosed in a larger policy (a `<PolicySet>` element in XACML) that collects the policies related to all datasets under the control of the hospital H_1.

Given the access request and the policy discussed above, the decision (in JSON format) computed by the PDP is as follows:

```
{
  "request": {
    "id": "5b210542-22ba-4cbf-9f57-519c7b9b3f98",
    "sender": "H1",
    "technology": "SQL",
    "query": "SELECT * FROM NGS1, GS1 WHERE NGS1.Patient_ID = GS1.Patient_ID"
  },
  "allow": true,
  "deployLocal": true,
  "loggingPolicy": "logging-policy-GS1.xml"
}
```

In the actual implementation, the logging policy is an XML file that simply contains a `<level>` element specifying a logging level. In our running example, the policy `logging-policy-GS1.xml` specifies the level `ALL`, meaning that all accesses to the dataset GS_1 must be logged.

The PDP decision is passed to the Persistence Manager to store it permanently. Then, if the decision is not to allow the request, the task is marked as completed, and the worker terminates. If, conversely, the decision is to allow the request, the request and the resulting obligations are disposed to the Image Manager, which selects the Dockerfile template matching the technology asked, injects in it the data source and logging policy, builds it, and deploys the image to a Registry. The URI of the image is stored in the task. If the container has to be deployed by the Data Holder, the Image Manager is asked to run it, and the endpoint exposed by the container instance is stored in the task. The task is finally marked as completed.

On the other side, the data user will find in the response either the URI of the image to be instantiated, or the URL of the endpoint ready to be queried, depending on the prescribed deployment location of the container.

3.3 Data analytics module

The instance of DAM will vary, depending on the technology specified in the request. Currently, the implementations for two technologies are provided:

- **REST**. To implement a REST endpoint that allows the data user to query the data, an approach similar to the XAC endpoint has been followed. We used, also in this case, the Express framework to implement an HTTP GET endpoint that returns the requested data to the data user. To make the web server reachable from outside the container (i.e., by the data user), the TCP port onto which the web server is listening (80) is made available to the services outside of Docker by specifying the `-p 80:80` flag while running the container. The web server is configured to store a log of each request in the `/logs` folder.
- **SQL**. To implement an SQL endpoint and allow the data user to run full SQL queries, we selected the Postgres DBMS[7]. One

6 We recall that in XACML, besides `permit` and `deny`, there are other two decision values: `indeterminate` and `notApplicable`.

7 https://www.postgresql.org/

of the main advantages of Postgres is that it is open source and easily customizable, which are two fundamental requirements for a DBMS to be integrated into our DAM. The DBMS is configured to log any query performed to the file system in the `/logs` folder.

Regardless the technology used for the endpoint, the DAM always includes a Log Manager component. It detects when a new file is created in the `/logs` folder, reads the logs contained in the file, filters them according to the logging policy, and sends the filtered information to the Persistence Manager. Specifically, the Log manager invokes an OS's API that callbacks a function when a new document is created inside a selected folder. The callback function is the one that actually reads the file, and filters and sends the logging data.

In our running example, when the data user accesses the obtained dataset, reading all contained data, the log produced by the DBMS and filtered by the Log Manager is as follows:

```
[2022-12-28 17:19:49.244Z] 192.168.2.10:ossecdb LOG:  duration: 4.550 ms
statement: SELECT * FROM NGS1, GS1 WHERE NGS1.Patient_ID = GS1.Patient_ID
```

3.4 Persistence manager

The Persistence Manager implements a queue populated with the logs from the Log Manager. When the number of logs exceeds the desired batch size or when the flush function is invoked, the Persistence Manager writes to IPFS all the documents by calling the `store` method exposed by the IPFS Adapter. The CIDs of the documents are used to update the Merkle DAG, and the CID of the root is stored on-chain through the Smart Contract by calling the `storeLogs` method of the Smart Contract Adapter. The IPFS Adapter and Smart Contract Adapter are common classes used in the DAM and the XAC.

3.5 Smart contract

The Smart Contract, which is the software component that allows the system to store and retrieve data on-chain, is implemented in Solidity[8].

The smart contract has two state variables:

- `decisions`: a mapping that associates to each request id the CID of the decision that has been stored on IPFS;
- `logs`: a mapping that associates to each request id the CID of the log stored on IPFS.

Both the variables are declared as `mapping(bytes32 => bytes32)`, since bytes32 is appropriate to store a request id in UUID4 format and a CID.

The function `storeDecision` takes as input the id of a request and the CID of the decision for that request that has been stored on IPFS, and updates the `decisions` mapping to map the former to the latter.

Similarly, the function `storeLog` takes as input the id of a request and the CID of the logs related to that request that has been stored on IPFS, and updates the `logs` mapping accordingly.

Finally, the function `getRequestInfo` takes in input the id of a request and returns a pair containing the CID of the decision and the CID of the logs related to the request by accessing the two state variables of the contract.

3.6 Validation results

To evaluate our implementation, we simulated a series of operations using a designated script. These operations included the deployment of the smart contract, the storage of a decision, and the storage of a generated log. The performed experiments allowed us to assess the costs associated with the use in the practice of the **THROTTLE**'s approach.

Our experiments have been carried out on Sepolia, an Ethereum testnet employing a proof-of-stake consensus mechanism to simulate the mainnet environment closely. The smart contract deployment incurred an average cost of 254,508 gas, equating to approximately 0.0004 ETH at the time of writing. Storing a decision consumed an average of 46,810 gas units (approximately 0.00007 ETH), while storing a log required an average of 46,855 gas units for the first log associated with a specific request and 26,956 gas units for appending subsequent logs to the DAG that contains logs for that request. Notably, retrieving a decision or a log did not entail any costs, as these operations do not necessitate updating any state on the blockchain and thus do not need to be committed. All the invocations can be examined on Etherscan.[9]

The costs of these operations are considered optimized, given that the on-chain data storage is minimal, consisting of only 32-byte CIDs for IPFS data. Layer 2 solutions (e.g., the Polygon scaling technology) may be explored if further optimization is deemed necessary.

4 Related work

Access Control (AC) systems (Hu et al., 2006) are software components required to determine the allowed activities of legitimate users, allowing or denying each attempt to access data in a system. Among the different approaches available, Attribute-Based Access Control (ABAC) gives great flexibility and finer granularity (Hu et al., 2015). In **THROTTLE** we adopt the ABAC approach, due to its suitability in scenarios where granular access controls for each individual is required (like, e.g., in the healthcare case study considered in this paper). Specifically, we use the OASIS standard language XACML (OASIS, 2013) for writing the policies regulating the access to the data stored in the federated data lakes.

8 https://docs.soliditylang.org/

9 https://sepolia.etherscan.io/address/0xe507c8252af80a684c20d95ec901d4eb4326c483

In the literature, some work has focused on combining AC systems with Blockchain/DLT technologies. In Ghaffari et al. (2020), an assessment of Blockchain/DLT-based AC systems has been performed, and a taxonomy is proposed to categorize the existing methods based on their type, application environment, and purpose. Concerning the authentication side, the goal is to adopt Blockchain/DLT to save credentials and user identities in immutable, secure, while easily accessible storage. Looking at the access control side, the immutability of the storage is exploited 1) to store the policies that regulate the authorization, or 2) to rely on smart contracts to execute the policies.

Focusing on the latter, which is closer to the aim of this paper (Di Francesco Maesa et al., 2018), proposes to codify AC policies as smart contracts named Smart Policies. Here, XACML policies are translated into Solidity-written smart contracts deployed on Ethereum. In this approach, the evaluation of an access request results in an execution of the smart policy that computes the decision on-chain and then informs the Policy Enforcement Point that actually enforces the decision. In Azaria et al. (2016) and Dagher et al. (2018), Ethereum's smart contracts are used to create intelligent representations of existing medical records stored within individual network nodes. Smart contracts contain metadata about the record ownership, permissions, and data integrity, while the contract's state-transition functions carry out policy evaluations, enforcing data alteration only by legitimate transactions. Finally, in Ugobame Uchibeke et al. (2018), Hyperledger Fabric is used to implement access control of big data by borrowing from two existing access control paradigms: IBAC and RBAC. IDs are assigned to data assets, and the blockchain serves as an auditable access control layer between users and their secure data store. The data IDs can be defined to represent a specific asset, a query that pulls some data, or an encoded function that runs to pull data from an existing data store.

Differently from the works discussed above, in the **THROTTLE** approach the access control policies are evaluated off-chain, and then the resulting decisions are stored on-chain. This allows to reduce the policy evaluation costs (in terms of money and time), still guaranteeing the traceability of data sharing. In fact, a large amount of resources with specific access rules (e.g., clinical data with a specific informed consent policy for each patient) leads to AC policies with large size. When such policies are rendered as smart contracts, their evaluation consists of executing much code in the blockchain, which may become expensive. In addition, our work differs from the other ones since it does not focus only on access control, but it defines an architecture that allows enacting in a federated environment the computed access decisions, providing automatic mechanisms for moving data and computation, and for logging the data accesses on IPFS and blockchain. Indeed, the aim of **THROTTLE** is not limited to data privacy, but it has been conceived to enable trusted data sharing for guaranteeing data sovereignty.

Notably, blockchain platforms register on the blockchain, in the form of transactions, information about the operations performed by smart contracts on the data structure stored on the blockchain. However, in general, this cannot be considered as a logging mechanism, because some operations in blockchain-based applications based on smart contracts do not generate, for performance and cost reasons, transactions. For example, in Solidity, getter functions are free, because they do not require any work from miners, as they just read information from a node. More in general, in the case of public, permissionless blockchain (as Ethereum, the blockchain considered in this work), the stored data can be freely accessed without performing any access request. Instead, in the works combining AC systems with Blockchain/DLT technologies mentioned above, requests must be sent to access data (typically not stored in the blockchain) and their evaluations are logged in the blockchain. From this perspective, our approach in addition allows us to define different logging strategies, to regulate which accesses have to be logged.

5 Concluding remarks

In this work, we have presented **THROTTLE**, a blockchain-based architecture we propose to support trusted data sharing for data analytics in federated data lakes. **THROTTLE** relies on access and logging policies, and on the immutability of blockchain, for guaranteeing data sovereignty. In addition, containerization enables data and computation mobility, thus increasing flexibility.

We applied the **THROTTLE** approach to a case study from the healthcare domain. Establishing a federation of hospitals' data lakes opens the possibility of setting up multi-centric clinical trials that can take advantage of data belonging to different institutions. With respect to the current settings, the approach we propose makes the definition of access policies easier, improving data sharing while creating a trusted environment.

As a future work, we plan to investigate the integration of a more expressive language for writing logging policies. At the time being, we can define simple policies that prescribe the logging level for a given data resource. We intend to develop a language specifically designed for logging policies that permit to specify with a fine grain which accesses have to be logged and how to log them. We also plan to extend the **THROTTLE** implementation to support other technologies to expose the dataset to the data user. This can be done by implementing new functions for parsing the requests with the new technology and adding a Dockerfile template that builds a container architected adequately for the new technology. Finally, we intend to provide a full-fledged implementation of the case study to perform a more extensive validation of the proposed approach.

Data availability statement

The raw data supporting the conclusion of this article will be made available by the authors, without undue reservation.

Author contributions

All authors listed have made a substantial, direct, and intellectual contribution to the work and approved it for publication.

Funding

This work has been partially supported by the Health Big Data Project (CCR-2018-23669122), funded by the Italian Ministry of Economy and Finance and coordinated by the Italian Ministry of Health and the network Alleanza Contro il Cancro, by the European Union's Horizon Europe research and innovation programme under grant agreement No 101070186 (TEADAL), by the PRIN project

SEDUCE (2017TWRCNB), and by the project SERICS (PE00000014) under the NRRP MUR program funded by the EU–NextGenerationEU.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Azaria, A., Ekblaw, A., Vieira, T., and Lippman, A. (2016). "Medrec: Using blockchain for medical data access and permission management," in 2016 2nd International Conference on Open and Big Data (OBD), 25–30. doi:10.1109/OBD.2016.11

Dagher, G. G., Mohler, J., Milojkovic, M., and Marella, P. B. (2018). Ancile: Privacy-preserving framework for access control and interoperability of electronic health records using blockchain technology. *Sustain. Cities Soc.* 39, 283–297. doi:10.1016/j.scs.2018.02.014

Di Francesco Maesa, D., Mori, P., and Ricci, L. (2018). "Blockchain based access control services," in 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 1379–1386. doi:10.1109/Cybermatics/_2018.2018.00237

European Commission (2020). *A european strategy for data*. [Dataset].

gaia-x (2022). *Gaia-x - architecture document*. [Dataset].

Ghaffari, F., Bertin, E., Hatin, J., and Crespi, N. (2020). "Authentication and access control based on distributed ledger technology: A survey," in 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), 79–86. doi:10.1109/BRAINS49436.2020.9223297

Gilliland, A. J. (2008). "Setting the stage," in *Introduction to metadata*. Editor M. Baca Second edn. (Los Angeles, CA: Getty Research Institute), 9.

Gorelik, A. (2019). *The enteprise big data lake*. O' Reilly.

Hu, V. C., Kuhn, D. R., and Ferraiolo, D. F. (2015). Attribute-based access control. *Computer* 48, 85–88. doi:10.1109/MC.2015.33

Jahnke, N., Spiekermann, M., and Ramouzeh, B. (2022). *Data catalogs - implementing capabilities for data curation, data enablement and regulatory compliance*. Tech. rep. Fraunhofer Institute for Software and Systems Engineering ISST.

LaPlante, A., and Sharma, B. (2016). *Architecting data lakes*. O'Reilly Media, Inc.

OASIS (2013). *Extensible access control markup language (xacml) version 3.0 oasis standard*. [Dataset].

Plebani, P., Salnitri, M., and Vitali, M. (2018). "Fog computing and data as a service: A goal-based modeling approach to enable effective data movements," in *Advanced information systems engineering*. Editors J. Krogstie and H. A. Reijers (Cham: Springer International Publishing), 203–219.

Tai, S., Eberhardt, J., and Klems, M. (2017). "Not acid, not base, but salt,". Lda in Proceedings of the 7th International Conference on Cloud Computing and Services Science (Setubal, PRT: SCITEPRESS - Science and Technology Publications), 755–764. CLOSER 2017. doi:10.5220/0006408207550764

Ugobame Uchibeke, U., Schneider, K. A., Hosseinzadeh Kassani, S., and Deters, R. (2018). "Blockchain access control ecosystem for big data security," in 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 1373–1378. doi:10.1109/Cybermatics_2018.2018.00236

Hu, V., Ferraiolo, D., and Kuhn, R. (2006). *Assessment of access control systems*. [Dataset].