Cluster Computing (2024) 27:10235–10244
https://doi.org/10.1007/s10586-024-04534-z

An autonomous blockchain-based workflow execution broker for e-science

Alper Alimoğlu[1] · Can Özturan[1]

Received: 1 February 2024 / Revised: 12 March 2024 / Accepted: 22 April 2024 / Published online: 15 May 2024
© The Author(s) 2024

Abstract
Scientific workflows are essential for many applications, enabling the configuration and execution of complex tasks across distributed resources. In this paper, we contribute an Ethereum blockchain-based scientific workflow execution manager, which distributes workflows to run on cluster computing providers that utilize the Slurm workload manager to execute them. We extended our blockchain-based autonomous resource broker called eBlocBroker, which is a DAO-based decentralized coordinator, by providing distributed workflow execution via blockchain. Through various tests, we demonstrate how our eBlockBroker autonomous organization, which is programmed as a smart contract, can manage scientific workflow submission, scheduling, and execution on cluster computing providers. The utilization of blockchain for distributed workflow execution is a new concept. We are motivated because our system has been developed with e-Science in mind where scientific workflows are widely utilized.

Keywords Blockchain · Scheduling · Cloud · Distributed programming · Partitioning · Cluster computing

1 Introduction

Blockchain technology is becoming more widespread and being used for multiple purposes, such as citizen, cluster computing, e-Science, Internet of Things, economy, finance, governance, and science. In addition to being used as a peer-to-peer (P2P) digital currency, Ethereum [4, 29] blockchain platform can also execute smart contracts. Once a smart contract is implemented and deployed onto the blockchain, it runs autonomously and exactly as programmed.

A smart contract called eBlocBroker [1, 12] was developed in cloud computing as an autonomous blockchain-based computational Broker for e-Science. eBlocBroker is a blockchain based computational broker, which lets computational service providers provide computing power to users. It brings together users (requesters), clusters (providers), and cloud storage services in a blockchain-based system to deliver a decentralized, secure, transparent, and efficient computational and storage resource distribution service. The eBlocBroker infrastructure was designed to meet the demands of academic and research communities, relying on the Ethereum blockchain platform and volunteer computing with a noncommercial focus.

A scientific workflow [26] consists of a series of jobs that perform required functions and have dependencies between jobs in terms of control or data. Many areas of study, e.g., astronomy, biology, and physics, have adopted scientific workflows. Workflows in Pegasus [18] are conceptualized without specific resource assignments; hence, executing scientific workflows in a distributed fashion has the following research challenge: When workflow executions are conducted in a centralized way, with all data routed via a single server, this leads to an excessive amount of network traffic, thereby, causing a performance bottleneck.

In this paper, we have extended the eBlocBroker infrastructure for applying blockchain technology to provide scientific workflow submission, execution, and data resource services to research communities. The eBlocBroker model allows a provider to execute a single job without dependency on any other jobs. On the other

✉ Alper Alimoğlu
alper.alimoglu@boun.edu.tr

[1] Department of Computer Engineering, Boğaziçi University, Istanbul, Turkey

hand, a single cluster or multiple independent clusters can also handle a directed acyclic graph (DAG) based workflow. We propose and implement a workflow engine that coordinates workflow execution recorded on blockchains, and that is executed using the Slurm workload manager [22] on computing clusters; Slurm is an open-source job scheduler that has been designed specifically for Linux clusters of all sizes. We contribute a DAG-based workflow engine on the eBlocBroker infrastructure which the user can run in order to partition and schedule a workflow for execution on multiple providers' clusters, also allowing parallel processing. Implementing blockchain-based computational and workflow services requires a diverse range of system software, optimization, decentralized autonomous smart contract programming, storage management, and system integration. Our main motivation for adding workflow execution features to the eBlocBroker is to create an alternative workflow execution service that is blockchain-based for researchers and academicians.

In the rest of the paper, we first present an overview of the related works in Sect. 2. In Sect. 3, we provide an overview of our system's architecture, including the interactions between components and the internal modules of the components. Next, in Sect. 4, we describe the workflow engine system that can map workflows onto multiple providers. The workflow engine manages a partitioning method, which splits a workflow into smaller sub-workflows that are distributed through blockchain and that can be concurrently executed on the computing clusters. Afterward, we present a mathematical representation of the cost model and explain the workflow engine algorithms developed. In Sect. 5, we present tests and experimental results from demonstrations of eBlocBroker on the Bloxberg scientific research blockchain [19]. Finally, we conclude the paper with a discussion and plans for future research in Sect. 6.

2 Related work

First, we will review blockchain-based decentralized cloud computing services. Then, we will present workflow execution algorithms.

Two companies have emerged that offer cloud computing services based on the Ethereum blockchain technology. These are Golem [6, 10] and iExec [7, 8]. Golem is the first to present a decentralized supercomputer infrastructure service, targeting complex applications such as computer-generated imagery rendering, scientific calculations, and machine learning. Golem Network allows users to connect and pay each other for sharing their unused resources. iExec aims to give decentralized applications using blockchain technology easy, safe, and broad access to the necessary services, computing resources, and data sets. Ethereum smart contracts are used as the basis for this technology that makes possible the construction of a virtual cloud architecture, which offers high-performance computing services on demand. Our work in this paper contributes a workflow execution engine that partitions and submits workflows through our eBlocBroker smart contract to multiple providers. Golem and iExec do not offer support for blockchain-based workflow submission and execution features.

Heterogeneous EarliestFinish-Time (HEFT) is a stat-ic scheduling algorithm proposed by Topcuoğlu et al. [28], containing processor selection and job prioritizing phases. During the processor selection phase, a chosen job is scheduled on a computing node that will bring about the least amount of time to complete, and in the job prioritizing stage, jobs are ranked. Pegasus, a workflow management system, utilizes a centralized engine to schedule distributed jobs. Pegasus uses HTCondor [27], which is a workload management system that is specifically designed to handle compute-intensive jobs and submit workflows. Another well-known project on workflow systems is Kepler [23], which supports the reproducibility and discovery of results and offers a user-friendly graphical interface for constructing workflows.

A network cut problem [24] can be used to identify the partitioning of a workflow, with a sub-workflow represented as a sub-graph. A system decentralized orchestration, which disassembles a workflow into smaller sub-workflows, is discussed in [21]. Many scheduling heuristics, such as HEFT, MinMin [15], Max-Min, and MCT [16], attempt to solve the workflow mapping problem.

Scheduling of extensive workflows to various execution locations with storage constraints using HEFT and MinMin are discussed in [17]. The provision of a scalable decentralized orchestration platform utilizing a high-level, functional data coordination language to execute workflows is discussed in [21].

Our work contributes to blockchain and decentralized autonomous organization-based workflow support. Our approach is based on coordinating the whole workflow submission with a distributed design model that partitions the workflow to be executed using the (i) topological generations layering (TGL) [11] scheduling and (ii) HEFT algorithm, as detailed in Sect. 4.5.

3 Blockchain-based autonomous organization model

Our approach (shown in Fig. 1) consists of three phases: (i) inputting a workflow, (ii) partitioning it for scheduling, and (iii) submission to computational and storage providers

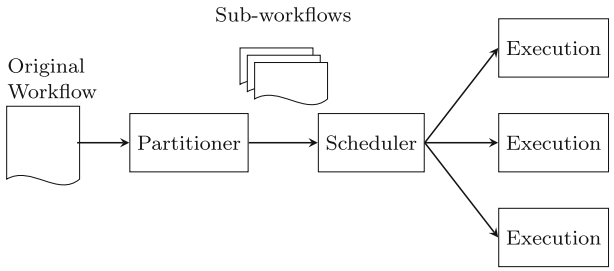


Fig. 1 The process of dividing and organizing a workflow. The sub-workflows are allocated to various providers by the scheduler

for execution. The partitioner takes the original workflow, which includes information about the estimated time of each job, the DAG of the workflow, and the calculated cost of each job as input. It then produces various sub-workflows utilizing a given scheduler (e.g., HEFT or TGL) and executes them based on their dependency on each other. The workflow engine algorithms enable users to deploy sub-workflows to different providers through the eBlocBroker smart contract as illustrated in Fig. 2.

The engine utilizes our approach to execute the workflow by submitting transactions to the eBlocBroker smart contract deployed on the Bloxberg blockchain [19] which is accessible to all providers and requesters. Every provider has a Flask web server that will provide information on the state of Slurm nodes and partitions, such as available cores. With this information, requesters can accurately determine the availability of each provider. Data storage and sharing are done through the InterPlanetary File System (IPFS) [14], which is a content-addressable, P2P network that is open-source and offers distributed data storage and delivery.

Figure 3 illustrates the communication between different eBlocBroker entities, such as the details of the jobs and workflows, their output files' path to the IPFS objects, which are publicly accessible on the blockchain.

The source codes of eBlocBroker and its script programs to interact with Bloxberg are available at [1] and source codes to carry out workflow engine tests are available at [2]. The components of the eBlocBroker infrastructure are illustrated in Fig. 4, which shows their interactions that are defined as follows:

- The *Bloxberg* [19] network is a P2P infrastructure that utilizes the Ethereum blockchain to execute smart contracts like eBlocBroker.
- The *providers* offer their computational and storage resources. They make use of the Slurm workload manager to run jobs.
- The *requesters* use providers' computational and data resources to upload prepared source code and data files into cloud storage. They then submit their jobs through eBlocBroker smart contract to the corresponding provider. Note that there is no possibility for abusing the providers' resources since the autonomous eBlocBroker smart contract performs necessary checks. Requesters pay for whatever they request to use. If they provide the wrong cost and it is less than the original cost, it will be rejected by the smart contract and the job will not be executed by the providers.
- *IPFS* is where the requesters' source code and data files are read from, and the completed job changes are written to by the provider. Note that since IPFS is a decentralized file system, each requester shares their source code, and each provider can share generated results with each other via IPFS. Also, providers, via the user, are able to interact indirectly with IPFS. As long as the IPFS node is operational and accessible to the public, any stored files will be accessible to the entire P2P network. In order to maintain the privacy of

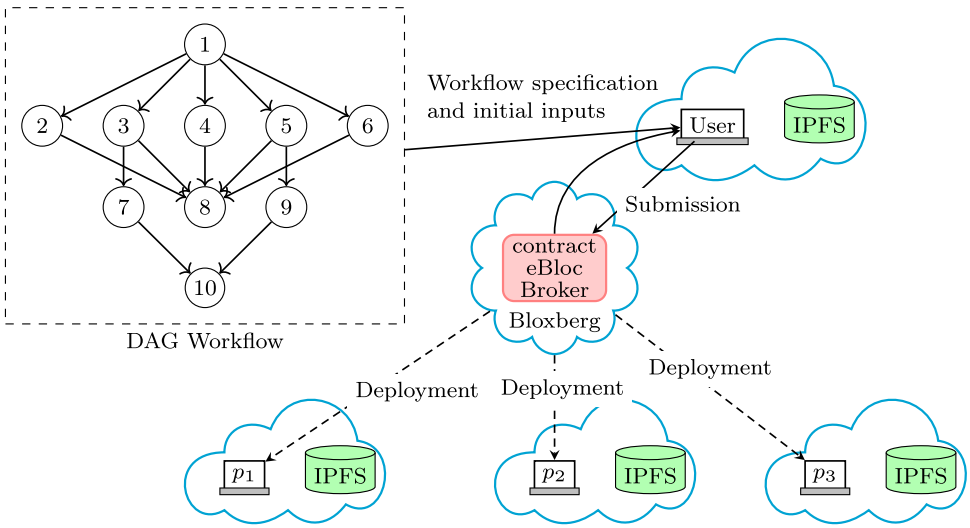


Fig. 2 Overview of our distributed architecture and the interactions between the user and providers. Source code and data files are transferred between providers and users through IPFS

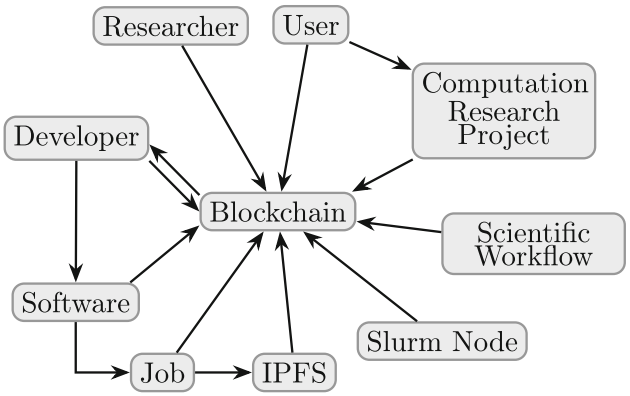


Fig. 3 Interactions between diverse eBlocBroker entities

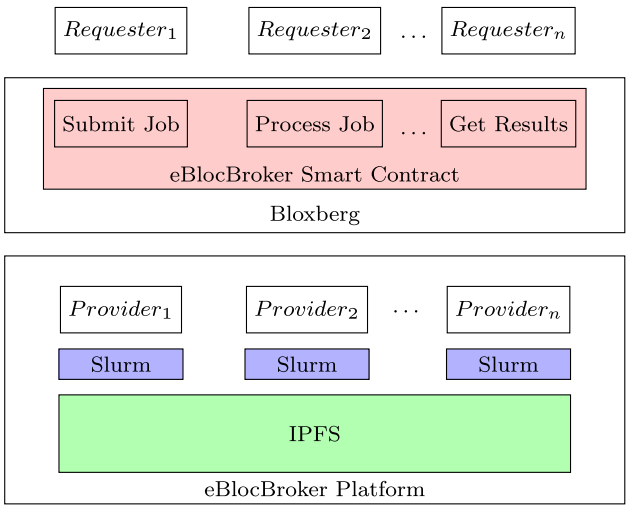


Fig. 4 Overview of the components of the eBlocBroker architecture and their connections with each other

files shared through IPFS, requesters can encrypt the files with the desired provider's fingerprint using the GNU Privacy Guard before uploading. The output files will also be encrypted with the job owner's fingerprint by providers before being uploaded onto IPFS, ensuring end-to-end encryption for file sharing. Note that the eBlocBroker infrastructure also provides file sharing using The EUDAT Personal Cloud Storage or Google Drive as a centralized cloud storage facility, which requesters can also use during job submissions.

4 Workflow support

We will investigate providing DAG workflow distribution support using eBlocBroker, which will help to automate computational management. The scheduler optimizes cost and run-time performance by selecting the best resources for the sub-workflows. Our scheduler coordinates the scheduling from the user side and not from the provider side, and implements both the HEFT and TGL algorithms. There is a difference compared to its original version since our experiments involve a shared file system, and the data transfer cost within a sub-workflow is disregarded.

HEFT and TGL algorithms require that characteristics of an application are known a priori, including run times, data size of communication between jobs, and job dependencies. Hence a static model is used to represent the workflow, which is used by HEFT and TGL. If variable output sizes are present, it is assumed that the user provides an estimate for it beforehand. In the scheduling phase of dependent jobs submitted to the same provider, output data transferring costs are considered to be zero.

We model workflows using DAGs with nodes representing computations and directed edges identifying the dependency of data flow between nodes.

4.1 Job-scheduling problem

The scheduling system model comprises a designated computing environment, an application, and a performance measure for scheduling. A workflow is represented by a directed acyclic graph, $G = (V, E)$, where E is the set of e edges between the jobs and V is the set of v jobs. Note that *node* and *job* terms are interchangeable. Each edge $(i, m) \in E$ defines the precedence constraint, indicating which job j_i must complete its execution before job j_m starts. Also, $data_{i,m}$ is the quantity of data that needs to be transferred from job j_i to job j_m.

Nodes at the tail of incoming edges of a node are called *in-dependency* nodes. Nodes appearing at the head of outcoming edges of a node are called *out-dependency* nodes. An *entry job* has no in-dependency within a particular workflow, while an *exit job* has no out-dependency. The workflow of the jobs is modeled as a graph. The graph's vertices represent jobs, while the edges represent intertask communication and job dependencies. DAGs are used to model parallel applications. Clustering effectively reduces communication costs and delays in distributed and parallel systems by grouping tasks that communicate with large amounts of data into the same labeled clusters and then allocating them in the cluster.

4.2 Random graph generator

In order to test the workflow engine algorithms and robustness of our system, a random DAG was generated and used, which is partitioned using HEFT and TGL algorithms. Figure 5 illustrates a randomly generated DAG.

NetworkX [9], a Python package, is employed to conduct graph analysis and create a workflow of jobs with

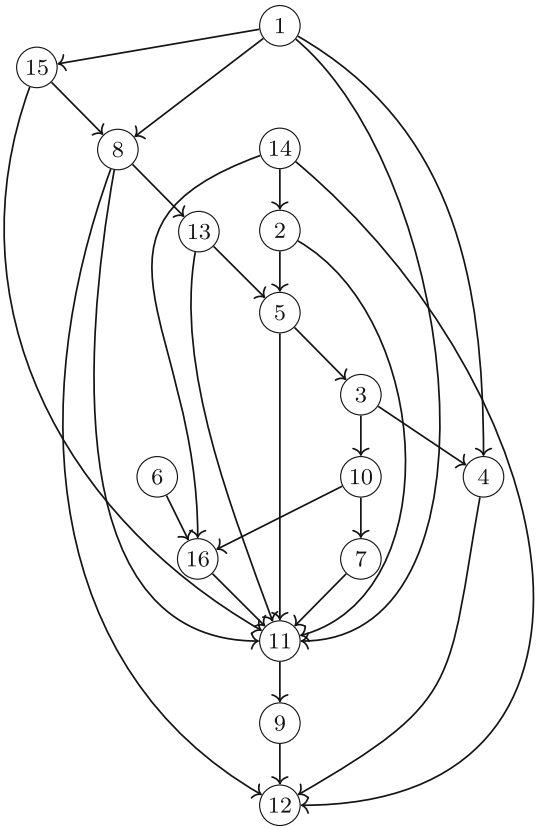


Fig. 5 A randomly generated Directed Acyclic Graph (DAG) workflow that has 16 nodes and 28 edges

dependencies between them. This package provides data structures and methods for storing graphs and can generate randomly directed DAGs without any cycles based on the given number of nodes and edges. Users can then submit their jobs as partitioned workflows to providers with the help of the constructed graph of the random DAG through the blockchain, as detailed in Sect. 4.5.

4.3 Cost model for executing a workflow

The list of symbols, their meanings, and definitions for calculating the computation and data cost for executing a workflow is represented in Table 1. The total computation cost of a workflow W_i on provider p is defined in Eq. (1). Data costs consist of data transfer, storage, and caching costs.

$$C_{W_i,p} = \sum_{j \in W_i} U_j \cdot X_j \cdot F_p^{\text{cpu}} \tag{1}$$

The total cost is composed of computation costs which involve running multiple workflow partitions on various providers, as well as data costs associated with each job. Our computation model is similar to that of Amazon AWS's, where vCPU-Hours of machine instance usage and a price per hour are used for calculating costs. We have not considered RAM cost as an explicit (separate) cost in our current cost model. Overall, machine configuration is already given based on the CPU core number, and a single provider does not provide more than one instance (the same organization having multiple instances can register them as separate providers with different Ethereum addresses). Like on Amazon AWS, a requester can look at the machine instance by each provider, which is pre-configured with CPU(s) and a certain memory amount, and choose the appropriate machine instance to run his job on.

Algorithm 1 Cost calculation on the Submit-Job

```
1: cost ← 0
2: for each job in workflow job set (j ∈ W_i) do
3:     cost ← cost + U_j · X_j · F_p^cpu
4: end for
5: cost ← cost + δ_{D_{W_i}} · F_p^trans
6: for each data in data set (d ∈ D_{W_i}) do
7:     if data d is paid for and stored on provider p then
8:         continue
9:     end if
10:    if data d is registered on provider p then
11:        cost ← cost + F_{p,d}^data
12:    else
13:        if data d is requested to be stored on provider p then
14:            cost ← cost + Z_d · F_p^store
15:        else
16:            cost ← cost + Z_d · F_p^cache
17:        end if
18:        cost ← cost + Z_d · F_p^trans
19:    end if
20: end for
```

We examine the *submitJob* function in detail, which is responsible for calculating payment for each job based on resource allocation in the eBlocBroker smart contract. Our cost model is implemented within the smart contract, shown in Algorithm 1. The cost calculation for computation is implemented in lines 2–4, while the cost calculation for data is implemented in lines 5–20.

4.4 Layer explanation

The NetworkX's *nx.topological_generations(G)* function divides a DAG into generations, which breaks down the workflow into smaller sub-workflows. Note that in a topologically generated set of nodes by this function, the ancestors of any node will always be included in the previous generation, and any descendants of a node will always be part of the following generation. Nodes are assigned to the earliest generation level they can be a part of. This stratification of the graph makes it so that each level contains all the nodes whose dependencies have been fulfilled by the nodes in the previous level. Each generated level has no dependency between nodes; therefore, all the jobs in each generation can be executed simultaneously.

Table 1 Main symbols used in the cost function given in Eq. (1) and Algorithms 1–3

Symbol	Meaning
P	Set of providers: $P = \{p_1, p_2, \ldots, p_{\lvert P\rvert}\}$
R	Set of requesters: $R = \{r_1, r_2, \ldots, r_{\lvert R\rvert}\}$
J	Set of jobs: $J = \{j_1, j_2, \ldots, j_{\lvert J\rvert}\}$
W	Set of workflows: $W = \{\omega_1, \omega_2, \ldots, \omega_{\lvert W\rvert}\}$
W_i	Set of jobs in a workflow: $W_i \subseteq J$
D_{W_i}	Set of data files on which workflow W_i uses during its execution: $D_j = \{d_1, d_2, \ldots, d_{\lvert D_{W_i}\rvert}\}$
F_p^{cpu}	Price per *core-minute* of the provider p
F_p^{cache}	Price per *cache-megabyte* of the provider p
F_p^{store}	Price per *storage-hour* of the provider p
F_p^{trans}	Price per *transfer-megabyte* of the provider p
$F_{p,d}^{\text{data}}$	Set of prices to use data files indexed by data d of the provider p
U_j	Number of CPU cores requested to run the job j
X_j	Expected *total CPU time* in minutes to run the job j
Z_d	Expected size in MB to download data d
$\delta_{D_{W_i}}$	Size of the output files as patches in MB generated by the workflow W_i on each D_{W_i} required to be uploaded

Algorithm 2 The workflow engine algorithm to distribute jobs using the HEFT [25, 28] algorithm

```
Generate a list of jobs to submit to each provider set (p_k ∈ P) using the HEFT algorithm.
Compute a sub-workflow ω_i, traversing graph starting from the sink node till there is no dependency from jobs in other providers.
Submit sub-workflow ω_i to its corresponding provider p_k.
while there are incomplete jobs in J do
    Update the status of all the ongoing jobs
    if there is a newly completed job in the list then
        for each generated list of remaining jobs to submit to each provider p_k in the provider-set (p_k ∈ P) do
            Traverse graph starting list of incomplete jobs till there is no dependency from jobs in other providers.
            Compute sub-workflow ω_i to submit
            Submit sub-workflow batch ω_i to its corresponding provider p_k.
        end for
    end if
end while
```

Algorithm 3 The workflow engine algorithm to distribute jobs using the topological generations [11] algorithm

```
Generate a list of layers of jobs to submit to each provider set (p_k ∈ P) using the TGL algorithm.
for each generated list in each layer that will be submitted to each provider p_k in the provider-set (p_k ∈ P) do
    while layer's all jobs' dependent jobs are incomplete do
        Update the status of all the ongoing jobs.
    end while
    Partition workload sub-workflow ω_i into number of available providers in P and submit them to its corresponding provider p_k.
end for
```

The assumption made in the engine is that the workflow does not contain any cycles. The engine takes a *lazy* approach to mapping workflow and produces partial executable workflows based on the layering of topological generations. The scheduler partitions the abstract workflow into smaller partial workflows. Each sub-workflow can only start running after all its dependent jobs have been

completed. Figure 6 illustrates examples of partitioned workflows using the topological generations layering.

4.5 Workflow engine algorithms

For the tests, the workflow engine Algorithms 2 and 3 were used to submit scheduled jobs. During the workflow submission, the state of all the submitted jobs is checked every 30 s and updated based on the state flow in Fig. 7. State changes trigger our algorithm to distribute new sub-workflows to providers.

Note that in our current job submission implementation, we assume that the number of jobs submitted in the parameter list of contract function call fits in the Ethereum transaction calldata, which is restricted in size.

4.6 Caching

In our model, the overall data transfer between the providers is carried out through IPFS. Note that the output data file consists of only changes as a patch generated by the source code, which will minimize bandwidth consumption and the cloud repository storage utilization.

At each step during the execution of a sub-workflow, each provider is aware of each input data file's IPFS hash through eBlocBroker, which will be used to download the input data from the corresponding provider's IPFS repo. For that reason, providers maintain the corresponding data files to run the workflow and each job's generated outputs in their IPFS repo until the workflow's execution has been completed. For data reusability, output data files may also be stored longer on the provider's IPFS repo if the user requests.

It is essential to consider data reusability when executing sub-workflow on distinct providers because dependent jobs may run on different providers instead of the same one. This will create additional data transfer costs where each provider should download the required input data from the generated output of a job from different providers into their local storage. Utilizing IPFS as a distributed storage solution will provide the benefit of sharing jobs' output results directly between providers.

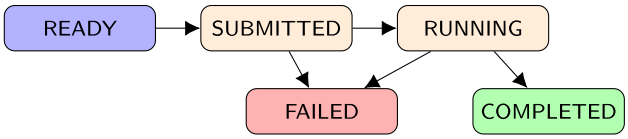


Fig. 7 State changes of a job

5 Experimental results and discussion

Before testing in real environments, the eBlocBroker contract was developed and tested using Brownie [20], and the contracts were implemented using Solidity version 0.8.20 and Ethereum Virtual Machine in Istanbul. Furthermore, it was deployed on Bloxberg, tested, and debugged further. The correctness of the cost model implemented in the smart contract and the robustness of our system are tested using a requester node to submit workflows and three provider nodes to obtain and run the workflows. The requester node and all the provider nodes are created on an AMD Ryzen 5 3600 6-Core Processor, each virtual machine of which has 2 vCPUs and 4 GB memory and allows file sharing using IPFS cloud storage. The HEFT algorithm is implemented in [25] and TGL is implemented in [11].

In the experiments, synthetic DAGs were constructed with a predetermined number of jobs and edges. Afterward, we randomly assigned values to these jobs and edges and conducted 10 separate tests shown in Table 2. In addition, in Table 3, T_{11}–T_{12} to test HEFT and T_{13}–T_{14} to test TGL run concurrently. During those tests if the sub-workflow's provider to be submitted has no idle core, it will be redirected to another provider with no load in order to get a fast

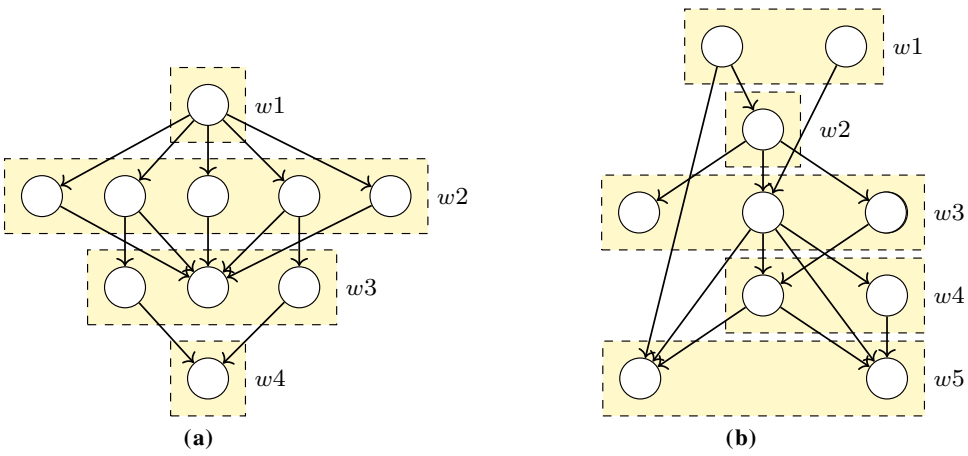


Fig. 6 Example view of topological generations layering on randomly generated DAGs

response. Note that for each individual case, tests are repeated three times, and the mean value is used for evaluation.

Our random graph generator uses the following input parameters to construct DAGs:

- Quantity of nodes in the graph ($|V|$) and edges in the graph ($|E|$) as shown in Tables 2 and 3, which illustrates various analyses of the test results. Note that detailed test transaction data for the tests carried out on the Bloxberg blockchain are available here [3].
- Each edge's weight is chosen uniformly randomly within the range of 15 to 1000.
- Every job's run-time is uniformly randomly determined between 2 and 5 min.

Furthermore, there are no disconnected nodes, meaning every node has a connection to one or more other nodes. For this purpose, the edge count is 1.75 times the node count.

In order to simulate the execution of workloads, the Slurm emulation mode is employed to create a cluster, and the sleep programs are utilized as jobs by their run time. Note that in emulation tests, the submitted jobs do not involve actual communication or computation; instead, they sleep for the given run-time duration. For that reason, input data files for each job are not downloaded onto the provider, but their size is only considered for the cost calculation. We assume that only the submitted jobs run on the providers during the tests. In our experiment, we used three providers, each having 8 cores in the Slurm emulation mode. Each provider has a separate computation and data price to ensure variety, as shown in Table 4. Note that once the workflow is submitted through eBlocBroker, the cost will remain unchanged until the workflow is finished, regardless of any price fluctuation.

We first schedule jobs according to the HEFT and our TGL algorithm. The pre-existing randomly generated DAG data helps us estimate the duration of the jobs. Then, we submit workflows in the simulated real environment using HEFT and TGL scheduling algorithms. Note that the HEFT algorithm requires pre-hand compile-time and run-time for executing jobs.

The provider's failure to deploy a *processPayment* transaction in order to receive payment can lead to unsuccessful job payments. This failure happens when multiple jobs are completed simultaneously and could attempt to obtain the same nonce during *processPayment* transaction deployment.

In Tables 2 and 3, the completion time is from when the workflow's request via the *submitJob* function is deployed until the workflow is completed in Slurm and its *processPayment* function request is deployed. Every workflow's completion time is logged in minutes. Since we ran Slurm in emulation mode, the run-time of jobs is simulation times. Completion time takes longer for TGL than HEFT for all given node values. Multiple layers may contain a single job that slows down workflow completion.

Furthermore, TGL is more aggressively distributing jobs to each provider. Since it just equally splits job sets to providers, there is an additional cost for data transfer. The HEFT algorithm distributes jobs to keep data transfer costs at a minimum. Consequently, TGL is not an efficient algorithm for workflow partitioning.

6 Conclusions and future work

The introduction of blockchain technology has revolutionized the banking and financial sectors, as well as e-Science. We proposed and implemented a blockchain-

Table 2 Results of the HEFT and TGL algorithms

Test	Workflow		Method	Completion time (min)	Gas used for *submitJob* (gas)	Gas used for *processPayment* (gas)	Actual/complete cost (*Cent*)	Failed
	$\|V\|$	$\|E\|$						
T_1	16	28		27	1,722,983	1,942,770	0.827/0.827	0
T_2	32	56		44	3,738,070	3,986,219	2.037/2.037	0
T_3	64	112	HEFT	52	6,745,934	8,037,017	3.234/3.234	0
T_4	128	224		100	13,858,812	16,524,333	6.867/7.025	2
T_5	256	448		191.5	27,859,947	33,574,732	13.909/14.060	3
T_6	16	28		48.5	3,533,072	1,932,928	1.768/1.768	0
T_7	32	56		64	5,709,622	3,965,208	3.169/3.169	0
T_8	64	112	TGL	96	10,192,826	8,017,429	5.410/5.410	0
T_9	128	224		222	21,100,398	15,922,550	12.243/12.243	0
T_{10}	256	448		266	29,775,297	32,933,948	19.511/19.528	1

Table 3 Results of the HEFT and TGL algorithms taking provider loads into account

Test	Workflow		Method	Completion time (min)	Gas used for *submitJob* (gas)	Gas used for *processPayment* (gas)	Actual/complete cost (*Cent*)	Failed
	$\|V\|$	$\|E\|$						
T_{11}	128	224	HEFT	102	13,591,822	16,278,401	6.972/7.009	1
T_{12}	128	224	HEFT	111	13,072,131	16,125,526	6.961/7.025	1
T_{13}	32	56	TGL	59.5	5,777,102	4,089,870	3.167/3.167	0
T_{14}	32	56	TGL	58.3	5,762,126	4,082,402	3.170/3.170	0

Table 4 Description of prices on provider p

Provider	F_p^{cpu} (*Cent*)	F_p^{storage} (*Cent*)	F_p^{cache} (*Cent*)	F_p^{trans} (*Cent*)
p_1	0.0110	0.000099	0.000099	0.000099
p_2	0.0100	0.000100	0.000100	0.000100
p_3	0.0099	0.000120	0.000120	0.000120

based scientific workflow execution manager. We conducted tests to ensure the correctness of our smart contract and evaluate the robustness of our engine algorithm in handling job submissions for providers to execute. The test results indicated that the providers had successfully completed most of the submitted jobs except for a few.

In the future, we plan to implement the reproducibility component, which is essential for keeping software sustainable [13]. The hash of the generated outputs from jobs can be recorded to facilitate reproducibility checks for organizations. Finally, the RISC Zero [5] tool may be used for verifiable computation using zero-knowledge proof. It can prove the computation of source codes implemented in Rust, and generated proofs could be verified.

In addition, a review feedback mechanism could be built for computation providers, similar to review feedback mechanism in Airbnb for home owners. To ensure optimal outcomes, a reputation system can also be established for requesters that leave reviews and ratings for providers, thus, allowing them to choose the most trustable providers for the job.

Acknowledgements This work was supported by the Turkish Directorate of Strategy and Budget under the TAM Project number 2007K12-873. We would like to thank Ben Hauser for his invaluable support on Brownie.

Author contributions A.A. contributed conceptualization, methodology, software development, validation, testing, formal analysis, investigation, writing, review and editing. C.O. contributed conceptualization, methodology, supervision, review and editing.

Funding Open access funding provided by the Scientific and Technological Research Council of Türkiye (TÜBİTAK). This study was funded by Turkish Directorate of Strategy and Budget (TAM Project, No. 2007K12-873).

Data availability https://github.com/ebloc/ebloc-broker/tree/v2.3.6/broker/test_setup_w; https://github.com/ebloc/ebloc-broker/blob/v2.3.6/broker/results/eblocbroker_wf_test-results_October_2023.xlsx.

Declarations

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. eblocbroker. https://github.com/ebloc/ebloc-broker
2. eblocbroker workflow scripts. https://github.com/ebloc/ebloc-broker/tree/v2.3.5/broker/test_setup_w
3. eblocbroker workflow test results. https://github.com/ebloc/ebloc-broker/blob/v2.3.6/broker/results/eblocbroker_wf_test-results_October_2023.xlsx
4. Ethereum Project. https://ethereum.org
5. The general purpose zero-knowledge vm. https://www.risczero.com
6. Golem. https://golem.network/
7. iExec. https://iex.ec/
8. iExec Whitepaper, Blockchain-Based Decentralized Cloud Computing. https://github.com/iExecBlockchainComputing/whitepaper/raw/master/V3/iExec-WPv3.0-English.pdf
9. Networkx. https://networkx.org
10. The Golem Project Crowdfunding—Whitepaper. https://golem.network/doc/Golemwhitepaper.pdf
11. Topological generations. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.dag.topological_generations.html
12. Alimoglu, A., Özturan, C.: An autonomous blockchain-based computational broker for e-science. Concurr. Comput. Prac. Exper. (2024). https://doi.org/10.1002/cpe.8087
13. Alimoglu, A., Özturan, C.: Design of a smart contract based autonomous organization for sustainable software. In: 2017 IEEE

13th International Conference on e-Science (e-Science), pp. 471–476 (2017)
14. Benet, J.: IPFS—content addressed, versioned, P2P file system. CoRR (2014). https://arxiv.org/abs/1407.3561
15. Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., Kennedy, K.: Task scheduling strategies for workflow-based applications in grids. In: CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005, vol. 2, pp. 759–767. IEEE, Cardiff, Wales (2005)
16. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. J. Parallel Distrib. Comput. **61**(6), 810–837 (2001)
17. Chen, W., Deelman, E.: Partitioning and Scheduling Workflows across Multiple Sites with Storage Constraints. In: Parallel Processing and Applied Mathematics, vol. 7204, pp. 11–20. Lecture Notes in Computer Science. Springer, Berlin (2012)
18. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M.H., Vahi, K., Livny, M.: Pegasus: Mapping Scientific Workflows onto the Grid. In: Grid Computing, vol. 3165, pp. 11–20. Lecture Notes in Computer Science. Springer, Berlin Heidelberg (2004)
19. Friederike Kleinfercher Sandra Vengadasalam, J.L.: Bloxberg Whitepaper, the blockchain for science, v2.0. https://bloxberg.org/wp-content/uploads/2022/07/bloxberg_whitepaper_2.0.pdf
20. Hauser, B.: Brownie. https://github.com/eth-brownie/brownie
21. Jaradat, W., Dearle, A., Barker, A.: Towards an Autonomous Decentralised Orchestration System (2015)
22. Jette, M.A., Yoo, A.B., Grondona, M.: SLURM: simple linux utility for resource management. In: Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003, pp. 44–60. Springer, Berlin (2002)
23. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurr. Comput. Pract. Exper. **18**(10), 1039–1065 (2006)
24. Papadimitriou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Books on Computer Science, Dover Publications, New York (1998)
25. Rocklin, M., Nikel-Shepherd, D., Rodrigues, C.: Heterogeneous earliest finish time. https://github.com/mrocklin/heft
26. Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M.: Workflows for E-Science: Scientific Workflows for Grids. Springer, Berlin (2006)
27. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurr. Comput. Pract. Exper. **17**(2–4), 323–356 (2005)
28. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst. **13**(3), 260–274 (2002)
29. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**(2014), 1–32 (2014)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Alper Alimoğlu received his B.S. in Information Systems Engineering from the Dual Degree Program between Binghamton University SUNY, NY, USA, and Istanbul Technical University in Istanbul, Turkey, in 2012. He received his M.S. in computer science from Binghamton University, in 2014. He is currently working towards the Ph.D. degree in computer engineering at Boğaziçi University in Istanbul, Turkey. His current interests are blockchain technologies, smart contract, and cloud computing.



Can Özturan received his Ph.D. Degree in Computer Science from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1995. After working as a Postdoctoral Staff Scientist at the Institute for Computer Applications in Science, NASA Langley Research Center, he joined the Department of Computer Engineering, Boğaziçi in Istanbul, Turkey, as a Faculty Member in 1996. His research interests are blockchain technologies, parallel processing, scientific computing, resource management, graph algorithms, and grid/cloud computing.