EBLOCBROKER: AN AUTONOMOUS BLOCKCHAIN-BASED COMPUTATIONAL BROKER FOR E-SCIENCE

by

Alper Alimoğlu

B.S., Information Systems Engineering, Dual Degree Program between State University of New York at Binghamton University and Istanbul Technical University, 2012

M.S., Computer Science, State University of New York at Binghamton University, 2014

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering
Boğaziçi University

2024

ACKNOWLEDGEMENTS

I wish to convey my heartfelt gratitude to my thesis supervisor, Prof. Can Özturan, who supported me at every stage of my Ph.D. journey. His guidance has equipped me to handle challenges with professional expertise. Without his wisdom and friendship, I would not have been able to bear this burden.

I would like to thank my fellow jury members Prof. Arda Yurdakul, Prof. Öznur Özkasap, and Prof. Cem Ersoy for their valuable inputs over the years. I would also like to thank fellow jury member Prof. D. Turgay Altılar for reviewing and providing feedback on the thesis.

I wish to express my gratitude to all members of the CMPE family. Research and support facilities are crucial for achieving a successful graduation. TETAM has provided me with this support, and I extend my heartfelt thanks to the founders of this crucial research center, especially Prof. Ufuk Çağlayan.

I would like to acknowledge the financial support from Turkish Directorate of Strategy and Budget under the TAM Project number 2007K12-873.

Last but not least, I want to express my gratitude to my parents, Ali Kemal and Şükran Alimoğlu, for their endless support.

ABSTRACT

EBLOCBROKER: AN AUTONOMOUS BLOCKCHAIN-BASED COMPUTATIONAL BROKER FOR E-SCIENCE

Blockchain infrastructures have emerged as disruptive technology, enabling cryptocurrencies and smart contracts. They can have a wide range of application areas in e-Science due to their open, public nature and global accessibility in a trustless manner. We propose and implement eBlocBroker, an autonomous blockchain-based middleware for selling computing power and providing data resources for eScience. eBlocBroker connects requesters with providers via an Ethereum-based private blockchain (Bloxberg) and uses cloud storage like B2DROP, IPFS, or Google Drive. In this thesis, we demonstrate how a smart contract can deploy a marketplace for data and computation-intensive research. We propose a cost model in the smart contract to calculate and record computation, transfer, storage, and cache usage costs. A Python-based system communicates with eBlocBroker and orchestrates job execution on the provider's end. We present eBlocBroker's features, infrastructure, implementation, algorithms, and experimental results. We also contribute an Ethereum blockchain-based scientific workflow execution manager that distributes workflows to cluster computing providers. We extended eBlocBroker for distributed workflow execution via blockchain, demonstrating its effectiveness through various tests. Finally, we introduce a decentralized autonomous software organization model and the AutonomousSoftwareOrg smart contract, providing project funding, voting mechanisms, and recordkeeping for software usage. Furthermore, software executions and their data files can be transactionally recorded, enabling the construction of execution graphs for analysis. A model of input/output data and software executions is presented using an AND/OR graph.

ÖZET

EBLOCBROKER: E-BİLİM İÇİN OTONOM BLOKZİNCİR TABANLI HESAPLAMA SERVİS SAĞLAYICISI

Blokzincir altyapıları yeni bir teknoloji olarak ortaya çıkmış ve kripto paraların ve akıllı kontratların gerçekleştirilmesine yol açmıştır. Blokzincir altyapıları açık, kamusal yapıları ve güvenilir bir şekilde genel erişilebilirlikleri nedeniyle e-Bilim'de çok çeşitli uygulama alanlarına sahip olabilirler. Biz bu tezde eBlocBroker adlı, e-Bilim için hesaplama gücü satışı ve veri kaynakları sağlama amacıyla otonom bir blokzincir tabanlı ara katman yazılımını öneriyor ve uyguluyoruz. eBlocBroker, Ethereum tabanlı özel bir blokzincir (Bloxberg) aracılığıyla talepleri sağlayıcılara bağlar ve B2DROP, IPFS veya Google Drive gibi bulut depolama hizmetlerini kullanır. Bu tezde, bir akıllı kontratın veri ve hesaplama yoğun araştırmalar için bir pazar yeri nasıl oluşturabileceğini gösteriyoruz. Akıllı kontratta, hesaplama, transfer, depolama ve önbellek kullanım maliyetlerini hesaplayan ve kaydeden bir maliyet modeli öneriyoruz. Bir Python tabanlı sistem, eBlocBroker ile iletişim kurar ve sağlayıcının tarafında iş yürütmesini düzenler. Biz bu tezde eBlocBroker'ın özelliklerini, altyapısını, uygulamasını, algoritmalarını ve deneysel sonuçlarını sunduk. Ayrıca, iş akışlarını küme bilgi işlem sağlayıcılarına dağıtan Ethereum blokzincir tabanlı bir bilimsel iş akışı yürütme yöneticisi sunuyoruz. Bu tezde, eBlocBroker'ı blokzincir aracılığıyla dağıtılmış iş akışı yürütmesi için genişlettik ve çeşitli testlerle etkinliğini gösterdik. Son olarak, merkezi olmayan otonom bir yazılım organizasyon modeli ve yazılım kullanımını kaydetmek, proje finansmanı sağlamak ve oylama mekanizmaları sunmak için AutonomousSoftwareOrg adlı bir akıllı kontrat tanıtıyoruz. Ayrıca, yazılım yürütmeleri ve bunların veri dosyaları işlemsel olarak kaydedilebilir, bu da analiz için yürütme grafiklerinin oluşturulmasını sağlar. Bu tezde giriş/çıkış verilerinin ve yazılım yürütmelerinin bir VE/VEYA grafik modeli sunduk.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES vii
LIST OF TABLES viii
LIST OF SYMBOLS x
LIST OF ACRONYMS/ABBREVIATIONS xii
1. INTRODUCTION 1
1.1. Contributions 5
2. RELATED WORK 8
2.1. Cloud Computing Services Based On The Ethereum Blockchain Technology 8
2.2. Workflow Execution Algorithms 12
2.3. Sustainable Software 13
3. EBLOCBROKER: AN AUTONOMOUS BLOCKCHAIN-BASED COMPUTATIONAL BROKER FOR E-SCIENCE 16
3.1. Introduction 16
3.2. The High-Level Structure Of eBlocBroker 17
3.2.1. Example: Deployment of a Job with Multiple Datasets 19
3.3. Cost Model 21
3.3.1. Cost Calculation in Job Submission Example 26
3.4. Detailed Explanation Of Processing Jobs On A Provider 29
3.5. Smart Contract Implementation 34
3.5.1. Job Submission Conditions 40
3.5.2. Provider's Price Commitment 41
3.5.3. Overlap Check 42
3.6. Data Transfer and Reusability 44
3.7. Results 46

3.7.1. Gas Consumption Limitation 49
3.8. Discussion 54
4. AN AUTONOMOUS BLOCKCHAIN-BASED WORKFLOW EXECUTION BROKER FOR E-SCIENCE 55
4.1. Introduction 55
4.2. Blockchain-Based Autonomous Organization Model 56
4.3. Workflow Support 58
4.3.1. Job-Scheduling Problem 59
4.3.2. Random Graph Generator 60
4.3.3. Cost Model for Executing a Workflow 61
4.3.4. Layer Explanation 63
4.3.5. Workflow Engine Algorithms 65
4.3.6. Caching 65
4.4. Experimental Results and Discussion 66
4.5. Discussion 70
5. DESIGN OF A SMART CONTRACT BASED AUTONOMOUS ORGANIZATION FOR SUSTAINABLE SOFTWARE AND E-SCIENCE 71
5.1. Introduction 71
5.2. Blockchain-Based Autonomous Organization Model 74
5.3. Smart Contract Details 79
5.3.1. Security Model 79
5.3.1.1. Unbounded Loops Vulnerability 79
5.3.1.2. Oracle Manipulation 79
5.3.1.3. Withdraw Funds for each Proposal 80
5.3.2. The Research-Certificate smart contract 81
5.4. Tests 81
5.4.0.1. Gas Consumption 82
5.5. Analyzing Software Executions and Data Usages 83
5.5.1. Data and Software Execution Graph 84
5.5.2. Analysis Algorithms 86
5.5.2.1. PageRank 87

5.5.2.2. DAGify 87
5.5.2.3. Merging same software 89
5.5.2.4. MaxKnockedDown 89
5.5.2.5. SWExecMinInput 90
5.6. Discussion 90
6. CONCLUSION 92
REFERENCES 95
APPENDIX A: COPYRIGHT NOTICE 104

LIST OF FIGURES

Figure 2.1. Interactions between various AutonomousSoftwareOrg entities. . . 15

Figure 3.1. Connections between diverse eBlocBroker entities in a decentralized ecosystem. 17

Figure 3.2. Components of the eBlocBroker architecture and their high-level relations with one another. 18

Figure 3.3. Creating a new job and sending it to the provider via eBlocBroker smart contract. 20

Figure 3.4. The data structure for storing provider records p on each provider's storage array, which includes job records. 24

Figure 3.5. A breakdown of the steps the provider takes to receive a job. . . . 30

Figure 3.6. Signatures of the eBlocBroker smart contract's fundamental functions. 35

Figure 3.7. Transferring source code and datasets from users to providers through remote cloud storage and IPFS. 41

Figure 3.8. Example of price commitment of a provider. 42

Figure 3.9. Linked list using a mapping after all the given transactions on the given example are deployed. 45

Figure 3.10. Comparison of job waiting times (in seconds) for our workloads. . 49

Figure 3.11. Comparison of gas cost for submitJob and processPayment. 49

Figure 3.12. Comparison of job run times (in minutes) for our workloads in Slurm. 49

Figure 3.13. Comparison of job completion times (in minutes) for our workloads in Slurm. 50

Figure 4.1. Overview of our distributed architecture and the interactions between the user and providers. 57

Figure 4.2. The process of dividing and organizing a workflow. 57

Figure 4.3. Interactions between diverse eBlocBroker entities. 58

Figure 4.4. A randomly generated Directed Acyclic Graph workflow that has 16 nodes and 28 edges. 60

Figure 4.5. Cost calculation in the *submitJob* function. 62

Figure 4.6. Example view of topological generations layering on randomly generated DAGs. 63

Figure 4.7. The workflow engine algorithm to distribute jobs using the HEFT algorithm. 64

Figure 4.8. The workflow engine algorithm to distribute jobs using the topological generations algorithm. 64

Figure 4.9. State changes of a submitted job. 65

Figure 5.1. Blockchain-based autonomous organization model. 74

Figure 5.2. Interactions between the users and providers through eBlocBroker smart contract. 77

Figure 5.3. Example of a software execution AND/OR graph. 77

Figure 5.4. The AutonomousSoftwareOrg smart contract function signatures. . 78

Figure 5.5. Withdraw fund of a proposal using the *Checks Effects Interactions* pattern. 81

Figure 5.6. ResearchCertificate smart contract function signatures. 81

Figure 5.7. Example of a general AND/OR graph of software executions. . . . 85

Figure 5.8. Example of a software hierarchy. 86

Figure 5.9. Graph of merged software version executions. 88

Figure 5.10. Remaining nodes after the maximum number of nodes have been knocked down, with only data node 11 is removed. 89

Figure 5.11. Software executions to generate data node 42. 90

LIST OF TABLES

Table 2.1. Similarities and differences between Ethereum blockchain-based decentralized cloud computing services and our work. 11

Table 3.1. Main symbols used to describe and calculate cost model in Equations 3.1 and 3.2, and also used in Figure 3.4 and Tables 3.2-3.4. 21

Table 3.2. Description of prices on provider p. 27

Table 3.3. Description of fees of the data file C and D on provider p. 27

Table 3.4. Description of gained cost calculations for each dataset of job j. . . 27

Table 3.5. List of eBlocBroker smart contract's fundamental and event functions with their descriptions. 38

Table 3.6. Description of the number of jobs submitted and completed for each workload during the test. 48

Table 3.7. Average gas cost of the eBlocBroker's core functions obtained from the Bloxberg as a real test environment. 53

Table 3.8. Average gas cost of the eBlocBroker's core functions obtained from the Bloxberg as a real test environment. 53

Table 4.1. Main symbols used in the cost function given in Equation 4.1 and Figures 4.5-4.8. 62

Table 4.2. Results of the HEFT and TGL algorithms. 68

Table 4.3. Results of the HEFT and TGL algorithms taking provider loads into account. 68

Table 4.4. Description of prices on provider p. 69

Table 5.1. Conditional modifiers in the AutonomousSoftwareOrg smart contract. 80

Table 5.2. Average gas cost of the AutonomousSoftwareOrg's core functions. . 83

Table 5.3. Main symbols used defining data and software execution graph. . . 86

Table 5.4. Graph of Software Execution Analysis Algorithms. 86

Table 5.5. Analyzing PageRank on example graphs. 87

Table 5.6. List of software in execution order and their initial generated data files. 87

LIST OF SYMBOLS

A_p	Number of available CPU cores of provider p
a_p	Address of the provider p
a_{jr}	Address of the requester r who owns the job j
B^{time}	Time interval between blocks in Bloxberg, which equals six seconds
$c_{D_j}^{\text{cache}}$	Caching cost for data files in D_j
$c_{D_j}^{\text{trans}}$	Transferring cost for data files in D_j
$c_{D_j}^{\text{data}}$	Sum of deposit fees for the set of data in D_j
$c_{D_j}^{\text{storage}}$	Storage deposit for data files in D_j
c_j^{cpu}	Computation cost for job j
D_j	Set of data files on which job j uses during its execution: $D_j = \{d_1, d_2, \ldots, d_{\lvert D_j \rvert}\}$
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^{storage}	Price per *storage-hour* of the provider p
F_p^{trans}	Price per *transfer-megabyte* of the provider p
F_{p,h_d}^{data}	Set of prices to use data files indexed by h_d of the provider p
H_{D_j}	Set of data file hashes for each data file in D_j: $H_{D_j} = \{h_{d_1}, h_{d_2}, \ldots, h_{d_{\lvert D_j \rvert}}\}$ where $d_j \in D_j$
h_d	Hash of the data d
h_j	Hash of source code that has Slurm script of job j
I_j	Number of CPU cores requested to run job j
J	Set of jobs: $J = \{j_1, j_2, \ldots, j_{\lvert J \rvert}\}$
$M_p(a_{jr}, h_d)$	Hours remaining to store with h_d on provider p
P	Set of providers: $P = \{p_1, p_2, \ldots, p_{\lvert P \rvert}\}$
$\bar{P}$	Set of provider addresses: $\bar{P} = \{a_{p_1}, a_{p_2}, \ldots, a_{p_{\lvert N \rvert}}\}$

q_j	Status of the job j with set $Q = \{0, 1, 2, 3, 4\}$, initial status $q_0 = 0$ (job is *submitted*), $q_1 = 1$ (job is *running*), $q_2 = 2$ (job is *refunded*), $q_3 = 3$ (job is *cancelled*), and $q_4 = 4$ (job is *completed*)
R	Set of requesters: $R = \{r_1, r_2, \ldots, r_{\|R\|}\}$
t_j^b	Unix timestamp of when the job j begins to run in Slurm
t_j^e	Unix timestamp of when the job j has ended in Slurm
U_{D_j}	Set of boolean variables indicating whether the provider's registered data file in D_j requested to be used: $U_{D_j} = \{u_{d_1}, u_{d_2}, \ldots, u_{d_{\|D_j\|}}\}$ where $d_j \in D_j$
W	Set of workflows: $W = \{\omega_1, \omega_2, \ldots, \omega_{\|W\|}\}$
W_i	Set of jobs in a workflow: $W_i \subseteq J$
X_j	Expected *total CPU time* in minutes to run the job j
Y_{D_j}	Set of the requested time in hours to store data files in D_j: $Y_{D_j} = \{y_{d_1}, y_{d_2}, \ldots, y_{d_{\|D_j\|}}\}$ where $d_j \in D_j$
Z_{D_j}	Set of file sizes to be downloaded in MB for each data file in D_j: $Z_{D_j} = \{z_{d_1}, z_{d_2}, \ldots, z_{d_{\|D_j\|}}\}$ where $d_j \in D_j$
α_{D_j}	Size of data files in D_j required to be downloaded in MB
δ_{D_j}	Size of the output files as patches in MB generated by the job j on each D_j required to be uploaded
$\delta_{D_{W_i}}$	Size of the output files as patches in MB generated by the workflow W_i on each D_{W_i} required to be uploaded
$\mathcal{J}_{p,h_j,i}$	Map of job records indexed by h_j and *index* i on provider p

LIST OF ACRONYMS/ABBREVIATIONS

AI	Artificial Intelligence
AuRa	Authority Round algorithm
B2DROP	EUDAT Personal Cloud Storage
BTFS	BitTorrent File System
COP	Condition Oriented Programming
cppr	colored parallel rush-relabel algorithm
DAG	Directed Acyclic Graph
DAO	Decentralized autonomous organization
EOSC	European Open Science Cloud
ETH	Ether
EUDAT	European Data Infrastructure
GPG	GNU Privacy Guard
HEFT	Heterogeneous EarliestFinish-Time
ICOs	Initial Coin Offerings
IPFS	InterPlanetary File System
IPOs	Initial Public Offerings
MATIC	Polygon
MB	Megabytes
NAS	NASA Advanced Supercomputing
NFT	Non-Fungible Token
ORCID iD	Open Researcher and Contributor ID
P2P	Peer-to-peer
PoA	Proof-of-Authority
Slurm	Slurm workload manager
TGL	Topological Generations Layering
VC	Volunteer computing
VOs	Virtual organizations

1. INTRODUCTION

Blockchain is an emerging technology promising to disrupt a variety of sectors. Banking, crowdfunding, cybersecurity, e-Science, finance, governance, healthcare, insurance, and the Internet of Things are some areas that blockchain technology will impact. It was originally developed to implement the digital cryptocurrency called Bitcoin [1] but soon proved to be a versatile technology with uses in diverse areas. Merkle [2] informally enumerates the features of the Bitcoin blockchain as follows: (i) runs on a peer-to-peer (P2P) network, (ii) pays incentives to keep it running, (iii) provides a service for which people pay, (iv) has an open-source code that anyone can run, (v) has a consensus mechanism, (vi) acts as a notary, (vii) keeps immutable public records, and (viii) cannot be stopped or interrupted.

Besides operating as a P2P digital currency, Ethereum [3,4] was later introduced, providing the ability to run smart contracts through a virtual machine. Smart contracts are applications that run autonomously and exactly as programmed, which can be used to implement publicly operating distributed autonomous organizations that could be used for crowdfunding, democratic governance, or a shared economy. Ethereum blockchain can be an ideal platform to help us realize e-Science goals sustainably. It is stated that e-Science promotes innovation in collaborative, computational, or data-intensive research across all disciplines throughout the research life cycle. Since the Ethereum blockchain runs smart contracts autonomously and continuously, is replicated, and its data are open to the public, it can be used for e-Science research. The Ethereum blockchain will enable the implementation of fundamental structures used in e-Science, such as computational resource managers and virtual organizations (VOs).

Our main focus is on providing support for Slurm-managed clusters due to their popularity among academics and researchers. From this point, we have begun developing autonomous organizations to help providers share their computation and earn a currency like dollars (in our tests, represented as USDmy token; in real-life deploy-

ments, it could be a stable USD coin such as USDC). Organizations, researchers, scientists, and individuals often invest in high-performance machines, high-performance gaming machines, and clusters for their work. These machines are idle when they are not being used. Our system is designed to create an ecosystem where providers can monetize their unused computing power by selling it to others. At present, there are a small number of highly centralized top cloud providers such as Amazon and Azure. Our system proposes a decentralized ecosystem where smaller players can sell their idle computing resources.

In this thesis, we first propose a blockchain-based autonomous resource broker called eBlocBroker, which is a smart contract that applies blockchain technology to provide a marketplace for computational and data resources to research communities. eBlocBroker is an open-source project, and a distributed autonomous market implemented in Solidity [5], which is an object-oriented, high-level programming language used to implement smart contracts on Ethereum. It was developed on the scientific research infrastructure (Bloxberg) [6], which is an Ethereum-based private blockchain network that has been operational since January 2019. The Bloxberg blockchain uses a Proof-of-Authority (PoA) algorithm as its consensus system, and its lowest interval between blocks in seconds is configured to 5 seconds. The Bloxberg blockchain is configured using the PoA consensus engine using Parity [7], which is an Ethereum client based on the Authority Round (AuRa) algorithm [8]. The eBlocBroker infrastructure connects the following:

- *Requester* is a user that wants to run applications (jobs) with datasets.
- *Provider* is a user that wants to share computational servers on clouds or home servers that utilize decoupled computational and data resources.
- *Bloxberg* is an Ethereum-based private blockchain network run by more than 50 research organization members worldwide.
- *Cloud storage* provider to store and share data through Google Drive, the InterPlanetary File System (IPFS) [9], or the EUDAT Personal Cloud Storage (B2DROP) [10,11] created by the European Data Infrastructure (EUDAT) [12,13]

for research communities. Note that IPFS is a content-addressable, P2P network that is open-source and offers distributed data storage and delivery.

- *The Slurm workload manager* (Slurm) [14] is an open-source job scheduler designed for Linux clusters of all sizes. It is utilized by each provider to execute jobs submitted through eBlocBroker smart contract.

In the traditional model, users (researchers) request computational servers from cloud providers or supercomputer centers. They submit a proposal for core hours in the case of national supercomputer centers or provide a credit card number for commercial cloud providers. In both cases, an account is created and maintained by the providers. These services are invaluable to the research communities. However, additional services can be available to users, especially at smaller scales. We enumerate potential scenarios for utilization as follows:

- Volunteer computing (VC) is a type of distributed computing in which users donate their computers' unused resources to a research-oriented project. One of the most popular VC framework is BOINC [15], which has more than a million volunteer computers. Blockchain can be used in VC as an instrument. A research team may run a cluster for a project; hence if the cluster has unused spare computing resources, it can be made available for other researchers and receive payments through smart contracts on the blockchain. The payments that are received can be utilized to cover the electricity expenses of the cluster, with any surplus being set aside as savings for future research.
- Communities formed due to citizen science and crowds of researchers that may need computational resources to run their applications. These communities may not have official representations to get computation server accounts on commercial or national centers. Smart contracts can be used to collect donations from communities on blockchains, which will be used to execute research applications and become a marketplace for the providers on the blockchain. Small donations can also be collected from communities using smart contracts, which can be used as a broker to distribute and run applications through blockchains.

- It is essential to maintain accounting records for applications running on providers. Since their costs are publicly recorded on the blockchain as they run, the transaction data can be analyzed while extracting information related to them. Auditors of projects and reviewers responsible for determining supported software projects may find this information valuable as well.

As stated in the European Commission's digital single market policy [16], European Open Science Cloud (EOSC) aims to provide (i) a world-class data infrastructure to store and manage data, (ii) high-speed connectivity to transfer data, and (iii) powerful high-performance computers to process data for Europe's 1.7 million researchers and 70 million science and technology professionals. The scale of EOSC involving millions of users requires an alternative infrastructure to connect them, which can operate uninterruptedly for an ecosystem consisting of crowds of requesters and providers, cloud storage services, computational and data resources, regulations, governance, and economy. Blockchain technology can provide this linkage necessary for the realization of this ecosystem. We surmise that blockchain can help the masses adopt Open Science Cloud.

The use of scientific workflows is essential for a wide range of applications, enabling the configuration and execution of complex tasks across distributed resources. Workflows are being integrated into the work of the scientific computing community. Consequently, this demonstrates the necessity of a workflow execution service for researchers and academicians. The primary motivation for integrating a workflow execution service into the eBloc computational broker was due to this specific necessity. There have been distributed workflow execution work, but our work is novel in that it is blockchain-based, and we are not aware of any work that provides such a blockchain-based workflow execution service. Our research focuses on the advancements in computational science and data utilization achieved through blockchain technology. We introduce an autonomous blockchain-based workflow execution manager for e-Science, extending the capabilities of the eBlocBroker infrastructure. This decentralized coordinator manages scientific workflow submission, scheduling, and execution on cluster

computing providers using the Slurm workload manager. Our system leverages the Ethereum blockchain to ensure secure, transparent, and efficient computational and storage resource distribution, addressing the bottlenecks associated with centralized workflow executions. Furthermore, the implementation of a DAG-based workflow engine on the eBlocBroker infrastructure facilitates parallel processing and increases the efficiency of scientific computations. Our system enhances the reproducibility and trustworthiness of scientific results by storing workflow execution details on the blockchain.

The emerging blockchain technologies have enabled the development of crypto-currencies and autonomous smart contracts that operate in decentralized and trustless settings. Distributed autonomous organizations can be implemented using smart contracts available on the Ethereum blockchain. In this thesis, we finally propose a distributed autonomous software organization model and its Ethereum smart contract implementation called AutonomousSoftwareOrg to provide a continuously operating virtual organization for open-source software development communities and users. AutonomousSoftwareOrg facilitates a funding mechanism based on crypto-currencies, a decision-making mechanism based on voting and record keeping for software usage citations and executions. Furthermore, we keep track of software executions along with their input and output data files to produce data and software execution graphs for analysis. AutonomousSoftwareOrg is deployed and tested on the Ethereum-based Bloxberg blockchain network.

1.1. Contributions

The contributions of this thesis are as follows:

(i) We propose and implement eBlocBroker infrastructure, which is an autonomous blockchain-based broker for sharing computing power and data in e-Science. The eBlocBroker smart contract, acting as a broker, aims to connect users (requesters), providers, and cloud storage services within a decentralized ecosystem,

utilizing blockchain's capabilities to enhance transparency, security, and efficiency in resource allocation. The source codes of eBlocBroker and its script programs are available at `https://github.com/ebloc/ebloc-broker`.

(ii) Data staging solutions for data stored on EUDAT, Google Drive, and a distributed IPFS file system.

(iii) Implementation of a cost model in the eBlocBroker smart contract, which calculates and records computation, transfer, storage, and cache usage costs.

(iv) An ERC20 token standard-based payment system for purchasing computing and data resources.

(v) Implementation of a new feature on top of the eBlocBroker infrastructure to offer scientific workflow submission, execution, and data resource services to research communities. Development of an engine on top of the eBlocBroker infrastructure that will be responsible for executing workflows on distributed providers through blockchain. The engine will partition and schedule a workflow for providers, enabling parallel processing. The smart contract's cost model has been adjusted to accommodate the computation and data costs of a workflow, supporting the engine's functioning.

(vi) Design and implement a smart contract called AutonomousSoftwareOrg, which is a DAO-based smart contract aimed at facilitating software development, ensuring sustainability and reproducibility of software, and promoting advancements in e-Science.

(vii) Implementation of the reproducibility component in software executions is essential for maintaining software sustainability [17]. Each software execution trace is recorded on the blockchain, showcasing the connection between input and output data files for each execution. The hash of the generated outputs from jobs can be recorded to facilitate reproducibility checks for organizations.

(viii) A blockchain-based software execution tracing system is implemented as part of an autonomous software organization to meet the need for software reproducibility. An AND/OR graph-based model for software executions and input/output data files has been developed. Various software tools, such as PageRank, DAGify, KnockedDown, and SWExecMinInput, were developed to analyze software execu-

tions. The source code of the AutonomousSoftwareOrg smart contract and graph tools are available at `https://github.com/ebloc/AutonomousSoftwareOrg`.

Contents of this thesis has been published in two journal papers and a conference paper. "An Autonomous Blockchain-Based Computational Broker for e-Science" as presented in Chapter 3 is published in the Concurrency and Computation: Practice and Experience journal [18]. "An Autonomous Blockchain-Based Workflow Execution Broker for e-Science" as presented in Chapter 4 is published in the Cluster Computing journal [19]. "Design of a Smart Contract Based Autonomous Organization for Sustainable Software" as presented in Chapter 5 is published in the 13th IEEE International Conference on e-Science [17]. Related work from these journal papers are given together in Chapter 2. Conclusions to the thesis are given in Chapter 6.

2. RELATED WORK

In this chapter, we present an overview of the existing approaches for Ethereum-based decentralized resource sharing through blockchain technology and workflow execution algorithms. Additionally, we provide an overview of current methods for a decentralized, blockchain-based system for sustainable software development, funding, citation tracking, and ensuring scientific reproducibility.

2.1. Cloud Computing Services Based On The Ethereum Blockchain Technology

Even though the technology for accessing remote computational servers using remote logins existed and has been used by scientists for decades, the grid computing technologies [20, 21] of the early 2000s first aggregated all the remote computational and data resources offered to scientists through grid middleware. Grids also introduced technologies to develop VOs that enabled large-scale resource sharing and problem-solving among distributed and loosely coordinated communities of users. Hence, grids facilitated the formation of virtual research environments.

Although grid technologies offered distributed resources, their governance model was more or less centralized. For example, governance of grids such as EGEE [22] or SEEGRID [23] was based on cultivating national e-Science centers by utilizing contributing resources, maintaining the software layer, and supporting its users. Such a model posed sustainability problems because relying on a few national centers to provide resources and support to many users can be a serious bottleneck. It is partly due to this bottleneck, grids such as EGEE or SEEGRID, and VOs [24] could not be sustained, even though much effort and support went into their development. Blockchain technologies offer smart contracts that can allow truly decentralized governance and funding schemes to be developed that are independent of national funding and support. A decentralized autonomous organizations (DAO) is a blockchain-based organization

governed by smart contracts and community consensus, operating without centralized control. Therefore, they have more potential to facilitate the formation of sustainable VOs.

VFuse [25] is an example of a volunteer distributed browser-based library to create and execute scientific workflows that use Ethereum only as a reward mechanism. Ethernity Cloud [26] is an example aiming to provide a decentralized cloud computing system that protects data privacy while the user's data is transferred, stored, or processed within the ecosystem.

Recently, five companies that provide Ethereum blockchain-based decentralized cloud computing services have emerged, which are Ethernity Cloud, Golem [27, 28], iExec [29, 30], SONM [31, 32], and Fluence [33].

Golem is the first service to showcase a decentralized supercomputer infrastructure, targeting complex applications such as computer-generated imagery rendering, scientific calculations, and machine learning. It aims (i) to facilitate renting resources (compute servers) of users to application owners and (ii) to evolve into an ecosystem where software developers will contribute software solutions to the Golem Application Registry, where machine providers will supply hardware that will run these software solutions for users who request it. Payment between the requesters and providers is made through Polygon [34]. Each application's generated output files should be in a specific folder (as `/golem/output`) under docker volumes. Golem uses Golem Service Bus [35], allowing its network services to communicate with one another and gftp [36] custom protocol for transferring files via Golem Network.

The iExec platform provides on-demand access to computing resources such as CPUs, GPUs, and datasets. It depends on the open-source Desktop Grid software XtremWeb-HEP [37], which has been used for parallel and distributed computing for a decade, and a diverse set of applications such as aeronautics, high energy physics, signal processing, and finance. The main iExec team has released version 3, which

presents a data wallet that provides secure data renting. For data transfer, iExec uses IPFS or pushes results to private storage providers.

SONM supplies cloud services based on distributed customer-level hardware. It uses the Cocaine platform [38], which is based on a distributed cloud structure. It can also work as a fog computing system by utilizing Cocaine's integrated locator service that allows the system to identify service accessibility. The SONM team launched an Ethereum sidechain for executing smart contracts with cheaper internal transaction fees. SONM can run essential scientific calculations requiring massive computing power, such as drug development, aerodynamic calculations, and meteor trajectory modeling. For data transfer output files, SONM uses BitTorrent File System (BTFS) or Samba share [39] using CIFS, which mounts shared directories content at a specified mount point. It uses the BTFS and CIFS Docker volume plugins to operate as any read-only directory in the requester's output folder. By 2023, SONM was no longer being maintained.

Ethernity Cloud recently emerged to generate software reproducibility; it records software traces in the blockchain. It is a decentralized confidential computing network powered by smart contracts with a core focus on decentralized confidential computing. In their solution, computational tasks with result data hash are certified on the blockchain while improving software reproducibility.

Fluence was established in 2017 and launched Mainnet in 2024. Fluence is a decentralized serverless platform and computing marketplace that utilizes blockchain economics. It is a decentralized "cloudless" computing platform that offers a worldwide, decentralized, scalable, and secure solution. With Fluence, developers create and launch applications on a network of computing providers, varying from high-end data centers to personal computers. To receive payment and incentives, providers must consistently demonstrate their ability to meet application requirements, competing on both price and performance.

Table 2.1. Similarities and differences between Ethereum blockchain-based decentralized cloud computing services and our work.

Study	**Billing model**	**Run environment**	**Cloud storage used for data transfer**	`git-diff` **support for output files**	**Cache support**	**GPU support**	**Data market**	**Multi-dataset access**	**Workflow support**
Golem	Pay-per-use	Docker container	gftp, Docker Hub	✗	✗	✓	✗	✓	✗
iExec	Pay-per-task	Docker container	IPFS, Docker Hub	✗	✗	✓	✓	✓	✗
SONM	Pay-per-use	Docker container	BFTS, CIFS, Docker Hub	✗	✗	✓	✗	✓	✗
Ethernity	Pay-per-use	Docker container	IPFS, Docker Hub	✗	✗	✗	✗	✓	✗
Fluence	Pay-per-use	Native OS	IPFS	✗	✗	✗	✗	✓	✗
Our work	Pay-per-use	Native OS	B2DROP, IPFS, Google Drive	✓	✓	✗	✗	✓	✓

Ethernity Cloud, Golem, SONM, iExec, and Fluence have minted their own currency tokens, concentrating primarily on business-oriented, general-purpose, and essentially closed-box systems. Users need to purchase their tokens, whose prices may fluctuate widely, to access the services of such systems. In order to use them except Fluence, providers package all the submitted tasks and run them in a Docker container; thus, users should dockerize their applications. The gaps in these studies are highlighted in Table 2.1. We based our requirements on the commonly used Slurm-based clusters by academicians and researchers. From this point, we began building autonomous organizations to facilitate providers in sharing their computational and data resources.

2.2. Workflow Execution Algorithms

Heterogeneous EarliestFinish-Time (HEFT) is a static scheduling algorithm proposed by Topcuoğlu et al. [40], 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 [41], 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 [42], which supports the reproducibility and discovery of results and offers a user-friendly graphical interface for constructing workflows.

A network cut problem [43] 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 [44]. Many scheduling heuristics, such as HEFT, MinMin [45], Max-Min, and MCT [46], 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 [47]. The provision of a scalable decentralized orchestration platform utilizing a high-level, functional data coordination language to execute workflows is discussed in [44].

2.3. Sustainable Software

Crowdsourced development has become increasingly significant in building successful platforms like OpenStreetMap, Instagram, Weather Underground, and Kickstarter. Communities are also being utilized for outsourcing software development and documentation [48]. A survey of the use of crowdsourcing in software engineering is presented in [49]. Blockchain technologies have become a popular method for obtaining crowdfunding for new projects. In contrast to previous methods of crowdsourcing and funding through centralized platforms, the latest blockchain technologies have paved the way for decentralized and trustless processes, making it possible for crypto-currencies to be utilized for crowdfunding. Companies can now gather funding from the crowd using smart contracts on the Ethereum blockchain, known as Initial Coin Offerings (ICOs), instead of the traditional Initial Public Offerings (IPOs).

In the Spring of 2016, the Ethereum platform launched its first DAO [50]. Millions of dollars' worth of Ether was gathered in the ICO. Unfortunately, due to a recursive Ethereum send exploit, a hacker was able to drain millions of dollars worth of Ether from the smart contract [51]. The DAO hack taught a valuable lesson about storing value and conducting transactions on blockchains: we must be careful about the correctness of smart contracts deployed on the blockchain. Before deploying smart contracts, verifying their correctness and testing them is essential.

Drips [52], an application developed on the Ethereum platform, enables flexible support of open-source projects in GitHub by sending funds with built-in dependency splitting. The investors' voting was used to coordinate decision-making within the virtual organization established by the DAO smart contract. The investors could make

project proposals, and their funding could be decided by voting. OpenZeppelin Governance [53] allows the deployment of on-chain voting protocols for general DAO.

The Autonomous Software Organization (*AutonomousSoftwareOrg*) implemented in this thesis shares similarities with the DAO but is simpler and designed for small software development communities. In addition to its other services, AutonomousSoftwareOrg facilitates gathering funds and selecting and financing software development proposals. The significance of distributed collaboration and conducting science on the Internet has been acknowledged by [54]. AutonomousSoftwareOrg has the ability to promote sustainable collaborations for software development. It presents a living, unstoppable, and uninterruptible smart contract that facilitates fund collection and delivery, along with a democratic mechanism for decision-making by members. Since the software can be characterized as being an invisible backbone for some of the world's largest scientific facilities [55], a smart contract running continuously on a blockchain has the potential to act as an autonomous entity representing the software infrastructure.

Recently, there has been a growing focus on the importance of citing software for credit, identification, discovery, and reuse [56]. The functions of our AutonomousSoftwareOrg smart contract include recording citations, usages, and software executions on the blockchain. By representing various entities (such as funding agencies, users, researchers, evaluators, and companies) as addresses on the Ethereum blockchain, our AutonomousSoftwareOrg can receive citations and usage links from them and log them as transactions on the public blockchain. As a result, the blockchain can provide the public with constantly updated and comprehensive linkage data between different entities, as depicted in Figure 2.1.

There has been an approach by [57] in order to ensure scientific reproducibility and data integrity of workflow executions by storing the hash of input and output data files in a MongoDB database. The Autonomous Software Organization shares a similar goal, where we have stored input and output hashes of a software executions on the

blockchain.

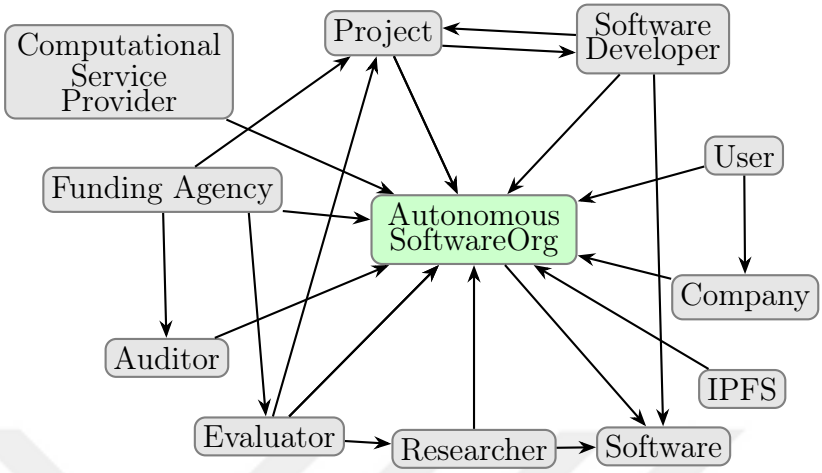


Figure 2.1. Interactions between various AutonomousSoftwareOrg entities.

When a smart contract function is called, all nodes in the P2P network execute the function and participate in the consensus process. As such, with a significant amount of users utilizing the Ethereum blockchain globally, the scalability issue must be resolved. This issue has been brought up in [58, 59], and solution proposals discussing the use of sharded blockchains are available [60]. Note that a high volume of transactions may occur every second in finance. Recording these on the blockchain in a timely manner is crucial. This does not apply to our AutonomousSoftwareOrg. While AutonomousSoftwareOrg supports activities such as membership, donations, voting, and citing, these actions occur infrequently, with significant intervals of time between each occurrence. As a result, any delay in deploying a transaction related to these activities into a confirmed block will not have any major consequences.

3. EBLOCBROKER: AN AUTONOMOUS BLOCKCHAIN-BASED COMPUTATIONAL BROKER FOR E-SCIENCE

3.1. Introduction

The eBlocBroker infrastructure, which also depends on the Ethereum blockchain platform, was developed to serve the needs of academic and research communities as non-commercial and provides the following capabilities:

- Records software executions on the blockchain, including their run-time, dependent data files' hashes, and the size of the generated output files. Figure 3.1 shows the communication between different eBlocBroker entities, such as the job and the data records publicly accessible on the blockchain.
- Receives jobs and their datasets onto the provider for each requester's home directory while maintaining the privacy of their folders, as detailed in Section 3.4.
- Allows end-users to run Slurm scripts on computational resource providers without the restrictions imposed by other state-of-the-art platforms, given the availability of the appropriate programs. This procedure will allow providers to run tasks on the native operating system instead of within a Docker container, as elaborated in Section 3.4.
- Determines the provider with the lowest price to execute the job using a cost model, as explained in Section 3.3.
- Gives end-users the option to choose the provider with the appropriate data in the cache to avoid redundant network traffic, thereby reducing costs and saving time. Data reusability for the large and complex datasets on the provider nodes, where jobs are scheduled on the provider node that contains the datasets, as described in Section 3.6.

- Provides file-sharing capabilities that support B2DROP, IPFS, and Google Drive storage facilities. While using IPFS, end-to-end encryption is also supported.

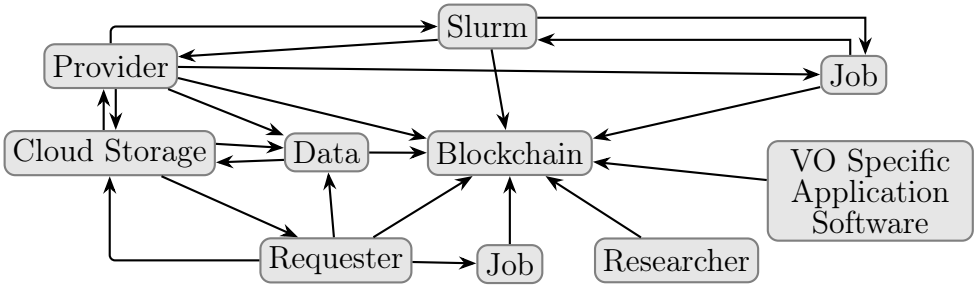


Figure 3.1. Connections between diverse eBlocBroker entities in a decentralized ecosystem.

The remainder of this chapter is organized as follows. In Section 3.2, we explain the infrastructure of eBlocBroker. In Section 3.3, we describe our cost model for sequential computations in cloud environments. In Section 3.4 we outline an in-depth description of a system design for job processing, which involves using a daemon on the provider side. In Section 3.5, we explain the implementation details of eBlocBroker smart contract. In Section 3.6, we explain the data transfer approaches used and the reusability of the transferred data. In Section 3.7, we present tests and experimental results from demonstrations of eBlocBroker on Bloxberg. In Section 3.8, we conclude the chapter with a discussion and plans for future research.

3.2. The High-Level Structure Of eBlocBroker

The eBlocBroker smart contract provides a marketplace for providers to share resources and publicly record all the executed jobs. It supports the relationship between multiple providers and requesters simultaneously. Our script programs act as a bridge, linking requesters to provider nodes through the eBlocBroker smart contract, enabling users to exchange data through cloud storage. The full source code of the eBlocBroker smart contract is available at `https://github.com/ebloc/ebloc-broker/tree/v2.3.1/contract/contracts`. The components of the eBlocBroker infrastructure are illustrated in Figure 3.2, which shows their interactions that are defined as follows:

- The *Ethereum blockchain infrastructure*, as Bloxberg, runs on a P2P network and executes smart contracts. The Bloxberg blockchain can be used by anyone by using their RPC API.
- The *providers* offer their computational and storage resources, and their datasets. The providers register their information to eBlocBroker, such as the number of CPU cores and resource prices in terms of *USDmy* token (10^8 USDmy tokens represent 1 dollar, i.e. 1 token represents 10^{-6} cents). The computational nodes of a cluster are managed with Slurm. The providers run a daemon process to interact with eBlocBroker and Slurm. They retrieve the list of the submitted jobs from eBlocBroker, and they run them.
- The *requesters* utilize providers' computational and data resources, which allows them to upload prepared source code and data files into the cloud storage and submit their jobs through eBlocBroker to the corresponding provider.
- The *centralized cloud storage facility* using B2DROP or Google Drive, and *decentralized storage* using IPFS are where the requesters' source code and data files are read, and the changes made by the completed jobs are written to by the provider. Note that since IPFS is a decentralized file system, file transfer operations are performed directly between the requester and provider in case the file object is not located at any other IPFS node.

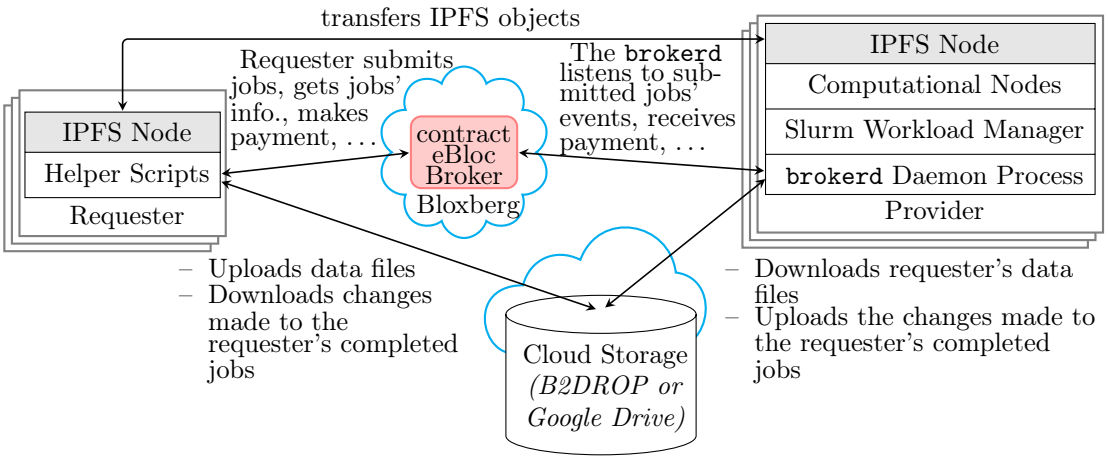


Figure 3.2. Components of the eBlocBroker architecture and their high-level relations with one another.

The requesters and the providers are represented as an Ethereum address, which is a unique identifier, has a balance, and is not linked to a real identity. Therefore, each requester and provider should be associated with their research identities for submitting a job through eBlocBroker. This procedure will create a more trustworthy environment where anyone can contact the contributors and assess them based on their development and job results. For the purpose of performing this, first, each requester and provider is registered in eBlocBroker with their ORCID iD (Open Researcher and Contributor ID), and authenticate their ORCID iD through a website we provide.

3.2.1. Example: Deployment of a Job with Multiple Datasets

Before interacting with eBlocBroker to send a job request to the provider, requesters should compress the source code of the job and data files using the MD5 hashing algorithm. During the compression, in order to create reproducible tarballs, the sort order is used by tar, and the timestamp is not preserved in the compressed file by gzip. Afterward, requesters send each compressed file to the provider through cloud storage.

Figure 3.3 illustrates an example of the steps involved in submitting a job, including the preparation of its dependent data files. Based on the given example, a detailed explanation of how the provider's system processes the job is given in Section 3.4. Note that during the job submission, the hash of the source code must be in the first order, and the order of the data file hashes is unimportant since they are considered individually. Our script is called ①, which coordinates the following steps:

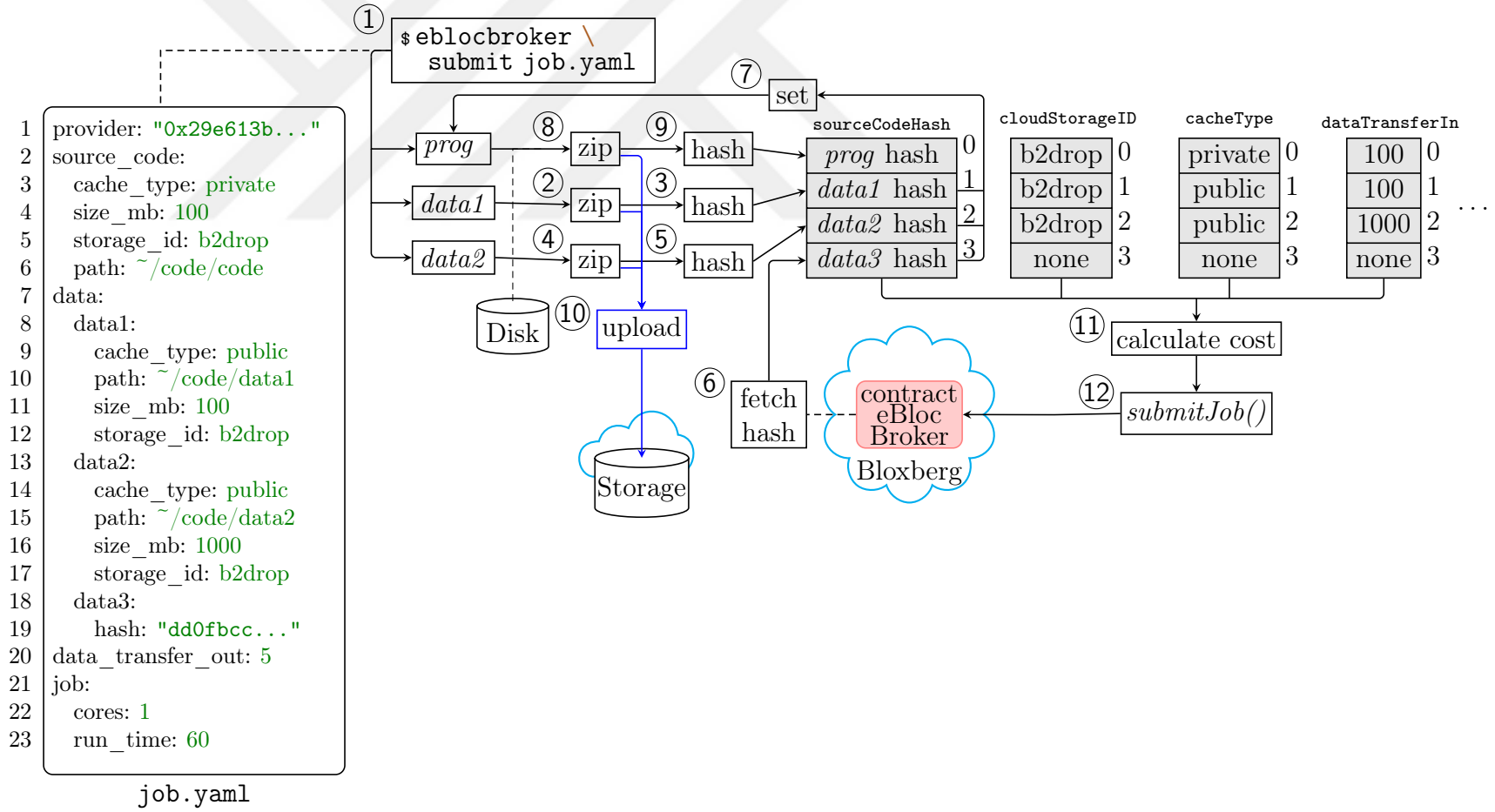


Figure 3.3. Creating a new job and sending it to the provider via eBlocBroker smart contract. The dashed arrows represent from which the data was fetched.

Steps ② and ④ compress *data1* and *data2*, and steps ③ and ⑤ generate and record their hash. Step ⑥ obtains the hash of registered *data3* from eBlocBroker, and records its hash. Step ⑦ sets and links all the data hashes on the *prog*. Step ⑧ compresses *prog*, and step ⑨ generates and records its hash. Step ⑩ uploads compressed *prog*, *data1*, and *data2* folders into the cloud storage. Step ⑪ calculates the cost of the job in *USDmy* tokens on off-chain (as detailed in Section 3.3) for each provider and selects the one with the lowest price. Step ⑫ converts the provided arguments inside the YAML file called *job.yaml* as the metadata of the job (which contains the requested number of cores, cache types, provider's Ethereum address, etc.) into a tuple and submits it as the job request to a provider through the eBlocBroker's *submitJob()* function.

3.3. Cost Model

A cost model for sequential cloud computing should follow a pay-per-use approach. In our work, we presented a cost model that calculates the amount of *USDmy* tokens a requester must pay to run a job through eBlocBroker, utilizing a provider's CPU cores and storage.

Table 3.1. Main symbols used to describe and calculate cost model in Equations 3.1 and 3.2, and also used in Figure 3.4 and Tables 3.2-3.4.

Symbol	**Meaning**
P	Set of providers: $P = \{p_1, p_2, \ldots, p_{\|P\|}\}$
R	Set of requesters: $R = \{r_1, r_2, \ldots, r_{\|R\|}\}$
J	Set of jobs: $J = \{j_1, j_2, \ldots, j_{\|J\|}\}$
a_p	Address of the provider p
$\bar{P}$	Set of provider addresses: $\bar{P} = \{a_{p_1}, a_{p_2}, \ldots, a_{p_{\|N\|}}\}$
a_{jr}	Address of the requester r who owns the job j
h_d	Hash of the data d
h_j	Hash of source code that has Slurm script of job j

Table 3.1. Main symbols used to describe and calculate cost model in Equations 3.1 and 3.2, and also used in Figure 3.4 and Tables 3.2-3.4. (cont.)

Symbol	**Meaning**
F_p^{cpu}	Price per *core-minute* of the provider p
F_p^{cache}	Price per *cache-megabyte* of the provider p
F_p^{storage}	Price per *storage-hour* of the provider p
F_p^{trans}	Price per *transfer-megabyte* of the provider p
F_{p,h_d}^{data}	Set of prices to use data files indexed by h_d of the provider p
A_p	Number of available CPU cores of provider p
$\mathcal{J}_{p,h_j,i}$	Map of job records indexed by h_j and *index* i on provider p
$M_n(a_{jr}, h_d)$	Hours remaining to store with h_d on provider p
I_j	Number of CPU cores requested to run job j
X_j	Expected *total CPU time* in minutes to run the job j
t_j^b	Unix timestamp of when the job j begins to run in Slurm
t_j^e	Unix timestamp of when the job j has ended in Slurm
q_j	Status of the job j with set $Q = \{0, 1, 2, 3, 4\}$, initial status $q_0 = 0$ (job is *submitted*), $q_1 = 1$ (job is *running*), $q_2 = 2$ (job is *refunded*), $q_3 = 3$ (job is *cancelled*), and $q_4 = 4$ (job is *completed*)
D_j	Set of data files on which job j uses during its execution: $D_j = \{d_1, d_2, \ldots, d_{\lvert D_j \rvert}\}$
H_{D_j}	Set of data file hashes for each data file in D_j: $H_{D_j} = \{h_{d_1}, h_{d_2}, \ldots, h_{d_{\lvert D_j \rvert}}\}$ where $d_j \in D_j$
Y_{D_j}	Set of the requested time in hours to store data files in D_j: $Y_{D_j} = \{y_{d_1}, y_{d_2}, \ldots, y_{d_{\lvert D_j \rvert}}\}$ where $d_j \in D_j$
α_{D_j}	Size of data files in D_j required to be downloaded in MB
U_{D_j}	Set of boolean variables indicating whether the provider's registered data file in D_j requested to be used: $U_{D_j} = \{u_{d_1}, u_{d_2}, \ldots, u_{d_{\lvert D_j \rvert}}\}$ where $d_j \in D_j$
Z_{D_j}	Set of file sizes to be downloaded in MB for each data file in D_j: $Z_{D_j} = \{z_{d_1}, z_{d_2}, \ldots, z_{d_{\lvert D_j \rvert}}\}$ where $d_j \in D_j$

Table 3.1. Main symbols used to describe and calculate cost model in Equations 3.1 and 3.2, and also used in Figure 3.4 and Tables 3.2-3.4. (cont.)

Symbol	**Meaning**
δ_{D_j}	Size of the output files as patches in MB generated by the job j on each D_j required to be uploaded
$c_{D_j}^{\text{cache}}$	Caching cost for data files in D_j
$c_{D_j}^{\text{trans}}$	Transferring cost for data files in D_j
$c_{D_j}^{\text{data}}$	Sum of deposit fees for the set of data in D_j
$c_{D_j}^{\text{storage}}$	Storage deposit for data files in D_j
c_j^{cpu}	Computation cost for job j

The list of symbols, their meanings, and definitions for calculating the cost is represented in Table 3.1. In our model, we have a set of requesters ($r \in R$) who owns a set of jobs ($j \in J$), which are executed by a set of providers ($n \in N$). The addresses of these entities are represented as a_{jr} and a_p, respectively. Each job j has a set of data files ($d \in D_j$). Here F_p^{cpu}, F_p^{storage}, F_p^{cache}, and F_p^{trans} provide us *core-minute*, *storage-hour*, *cache-megabyte*, and *transfer-megabyte* unit prices of the provider p respectively. We need this information to calculate the total cost of the computational and data resource on each provider.

Figure 3.4 illustrates the data structure to store provider and job records using the direct address approach. Each provider and job record consists of fields. Here the leftmost column $\bar{P} = \{a_{p_1}, a_{p_2}, \ldots, a_{p_{|N|}}\}$, corresponds to a *sha3* hash (as a unique key) to lookup the mapped item that is a provider record (p). Each provider address is in the $\bar{P}$ mapping. Furthermore, within the provided data, each job key (h_j) in the jobs mapping $J = \{h_1, h_2, \ldots, h_{|J|}\}$ maps to an array (storage of the job structure) and an *index* points to a structure that contains the job record (j).

We propose the cost function defined in Equation 3.2 to formalize our cost model. The total costs are comprised of the costs resulting from the computation of the job and

its data usage, determined by the storage duration of the data $M_n(a_{jr}, h_d)$ during execution as well as the latest committed prices of provider p for F_p^{cpu}, F_p^{cache}, F_p^{storage}, F_p^{trans}, and F_p^{data}.

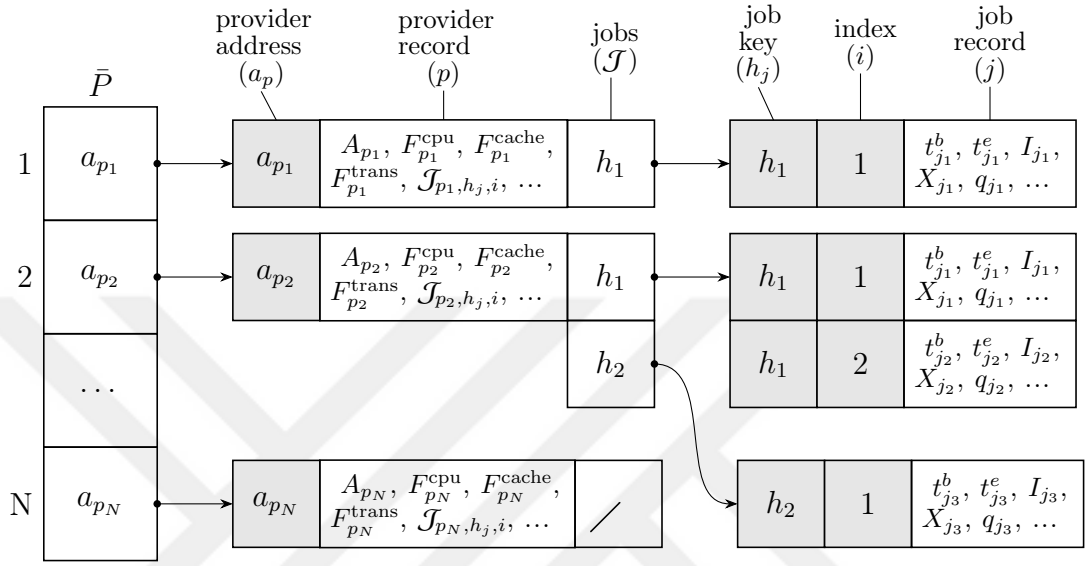


Figure 3.4. The data structure for storing provider records p on each provider's storage array, which includes job records. Nodes that are lookup values are colored gray. Here record of job j that has index i on provider p is indicated as $\mathcal{J}_{p,h_j,i}$.

We divided Equation 3.2 into two parts, where [A] and [B] define the computation and data costs of the provider resources, respectively. The computation cost in [A] is determined by multiplying the CPU run time of the job, number of cores to be used by the job, and F_p^{cpu}. The result will be summed with the data cost, which will be calculated in [B].

The sub-calculation of [B] is given in Equation 3.1, which is divided into four parts. [C], [D], [E], and [F] are the costs of the used data resources, defining transfer cost, storage cost, cache cost, and dataset cost, respectively. [C] in Equation 3.2 multiplies the size of the generated output files in MB to be uploaded and F_p^{trans}. Afterward, a set of data files on which job j uses during its execution is iterated one by one in order to calculate the remaining data cost, where corresponding information related to each data file is passed to Equation 3.2. In order to calculate the remaining transfer cost in [C], the size of the downloaded corresponding data d in MB is multiplied

by F_p^{trans}. Furthermore, to calculate the storage cost in [D], the size of the downloaded corresponding data d in MB, the required hours to store the data file, and F_p^{storage} are multiplied. Next, to calculate the cache cost in [D], the size of the downloaded corresponding data d in MB is multiplied by F_p^{cache}.

$$R\left(\begin{array}{l} n,\, a, \\ h,\, y, \\ z,\ u \end{array}\right) = \begin{cases} \left(\underbrace{z \cdot F_p^{\text{trans}}}_{\substack{\text{transfer cost} \\ [\mathbf{C}]}} + \underbrace{z \cdot y \cdot F_p^{\text{storage}}}_{\substack{\text{storage cost} \\ [\mathbf{D}]}} + \underbrace{z \cdot F_p^{\text{cache}} \cdot [\![y = 0]\!]}_{\substack{\text{cache cost} \\ [\mathbf{E}]}} \right) \cdot [\![M_p(a,h) = 0]\!] & \text{if } u \text{ is false} \\ \underbrace{F_{p,h}^{\text{data}}}_{\substack{\text{dataset cost} \\ [\mathbf{F}]}} & \text{otherwise.} \end{cases} \tag{3.1}$$

$$C\left(\begin{array}{cccc} n, & a_{jr}, & H_{D_j}, Y_{D_j}, Z_{D_j}, \\ U_{D_j}, \delta_{D_j}, & I_j, & X_j, & D_j \end{array}\right) = \underbrace{\underbrace{I_j \cdot X_j \cdot F_p^{\text{cpu}}}_{\substack{\text{computation} \\ \text{cost} \\ [\mathbf{A}]}} + \underbrace{\delta_{D_j} \cdot F_p^{\text{trans}}}_{[\mathbf{C}]} + \sum_{d \in D_j} R\left(n, a_{jr}, h_d, y_d, z_d, u_d\right)}_{\substack{\text{data cost} \\ [\mathbf{B}]}} \tag{3.2}$$

Note that, in Equation 3.1, for the corresponding data d, if the requested hours to store it is zero, then [D] is ignored; if the requested hours to store it is positive, then [E] is disregarded. Furthermore, if the data has already been registered and requested by the requester, only [F], which is viewed as a sub-storage cost, will be taken into account (where u_d equals 1). On the other hand, the corresponding data d may have already been stored or downloaded from the cloud storage, eliminating any data cost. This can happen if all of the following conditions hold true:

- When the provider has not yet registered the data d (where u_d equals 0).
- When there is still time remaining to store data d is greater than zero (where $M_p(a_{jr}, h_d)$ returns a positive value). This indicates that in previous job submissions, the requester was requested to store the data d on the provider and has already paid the fee.

3.3.1. Cost Calculation in Job Submission Example

Below we provide further details on some of the symbols listed in Table 3.1 related to provider price units.

- F_p^{cpu} is the *core-minute* price for using a core per minute on provider p. The execution of the job as computation is charged at the rate of `N_USDT/CoreNumber/JobRunDuration` by the minute they are executed. The calculated computational cost is defined as c_j^{cpu} for job j. For example, with 60 *core-minutes*, a requester can run a job on the provider for 60 minutes on one core or 15 minutes on 4 CPU cores. In addition, running 128 CPU cores for 100 minutes will cost 12,800 *core-minutes*.
- F_p^{storage} is the *storage-hour* price for long-term storage, an MB per hour on provider p. The storage of the dataset is charged at the rate of `N_USDT/MB/StorageDuration` by the hour they are stored. The calculated storage cost is defined as c_j^{storage} for job j. Time available on the provider's storage resource is denominated in *storage-hours*. For example, with 1000 *storage-hours*, a requester can store a data file on the provider for 1 GB per hour or 500 MB per 2 hours, and so on.
- F_p^{cache} is the *cache-megabyte* price for the storage used in MB to temporarily store a data file on a storage of provider p storage until its corresponding job is completed its execution. The calculated cache cost is defined as c_j^{cache} for job j. The amount of storage in MB on the provider's temporary storage resource is denominated in *cache-megabytes*. When the data file's related job is still running, the storage cost will be applied again if the data owner requests to use the same temporarily stored data file. For example, with 500 *cache-megabytes*, a requester can store a 500 MB data file on the provider.
- F_p^{trans} is the *transfer-megabyte* price for the bandwidth used in MB to transfer data between the provider p and the cloud storage facility. The calculated data transfer cost is defined as c_j^{trans} for job j. The amount of bandwidth used in MB is denominated in *transfer-megabytes*. For example, with 500 *transfer-megabytes*,

the requester can send a 500 MB data file through a cloud storage facility.

- F^{data}_{p,h_d} is the fee to use data file d that has hash h_d on the provider p.

In the following examples, we will assume that the provider registers itself with the following prices listed in Tables 3.2 and 3.3:

Table 3.2. Description of prices on provider p.

provider	F_p^{cpu} **(usd)**	$F_p^{storage}$ **(cents)**	F_p^{cache} **(cents)**	F_p^{trans} **(cents)**
n	0.001	0.00001	0.00001	0.0001

Table 3.3. Description of fees of the data file C and D on provider p.

data	F_{p,h_d}^{data} **(cents)**
C	0.2
D	0.2

Table 3.4. Description of gained cost calculations for each dataset of job j.

data	$c_{D_j}^{storage}$ **(cents)**	$c_{D_j}^{cache}$ **(cents)**	$c_{D_j}^{trans}$ **(cents)**	F_{p,h_d}^{data} **(cents)**
A	0	0.00001	0.0001	0
B	0.1	0	0.1	0
C	0	0	0	0.2
D	0	0	0	0.2
sum	0.1	0.00001	0.1001	0.4

Our first example is a requester who is willing to submit a job j that requests the following resources:

- 1 core for 100 minutes that will cost 0.1 USD.
- Single dataset (A), which is the source code. The size of the source code is 10 MB
- Content size of the generated output to be uploaded into the cloud is 10 MB.

Payment calculation for the job is as follows:

- Executed run time of the job is 60 minutes. The gained amount is 0.06 usd, and 0.04 usd is refunded (unused 40 minutes).
- Size of the downloaded data files is 9 MB, and the calculated data size to be uploaded is 10 MB. Here the caching cost gained amount is 0.0009 cents, and the refunded amount is 0.0001 cents. The transferring cost gained amount is 0.0019 cents, and the refunded amount is 0.0001 cents. Hence storage cost gained and refunded amount is zero.

Our second example is a requester who submits a job j with the following resource requirements:

- 1 core for 60 minutes, which will cost 0.06 USD.
- Outbound data transfer for 100 MB and 4 datasets: (i) size is 9 MB, (ii) size is 1000 MB, storage duration of 10 hours, (iii) data C, and (iv) data D.

Note that these costs are first calculated in the *submitJob()* function through smart contracts. For example, afterward, if the same job is submitted within 10 hours, which is the storage duration for dataset B, no storage and cache fee would be paid for data B. Also the size of the downloaded data files might be much smaller than the requested amount since they submitted compressed. After the job is completed, once the provider submits calls to the eBlocBroker's *processPayment()* function, the calculated costs for data resources are shown in Table 3.4. Payment calculation for the job is as follows:

- Executed run time of the job is 7 minutes. Hence for the computation cost, the gained amount is 0.007 usd (used 7 minutes), and 0.053 usd is refunded (unused 53 minutes).
- Downloaded and file space usage for A is 1 MB. Here the cache cost gained amount is 0.00001 cents, and 0.00008 cents is refunded.

- Size of the sum of the downloaded data files is 1001 MB, and the calculated data size to be uploaded is 100 MB. Here the data transferring cost gained amount is 0.1101 cents, and 0.0008 cents is refunded, which will cover the data transfer costs for A and B and the generated output data file to be uploaded.
- Storage cost for data file B is 0.10 cents. Since storage cost is already paid for data file B no cache cost is applied.
- Dataset fee for using the datasets C and D is 0.2 cents each.

3.4. Detailed Explanation Of Processing Jobs On A Provider

Each provider runs a Python daemon process called `brokerd`, which is responsible for facilitating the communication between Bloxberg, eBlocBroker, and the Slurm resource manager. The `brokerd` has permission to create new users in the Linux system and switch to their account to submit the jobs' batch script to Slurm as the user. The `brokerd` retrieves information related to submitted jobs from eBlocBroker, downloads them from the cloud storage, prepares the corresponding Slurm jobs, and submits them to Slurm through the `sbatch` command. When jobs are completed, `brokerd` uploads the changes made by jobs into the cloud storage and obtains the completed job's deposit from eBlocBroker.

The `brokerd` can interact with the functions of eBlocBroker via Bloxberg by using our script programs that use Brownie [61], which is a Python framework for deploying, testing, and interacting with Ethereum smart contracts. Note that under the hood, Brownie uses Web3.py [62], which is a Python interface for interacting with Ethereum and also for testing ganache [63], which is an Ethereum simulator.

Figure 3.5 provides an example of a job processing by `brokerd` is shown in the following steps:

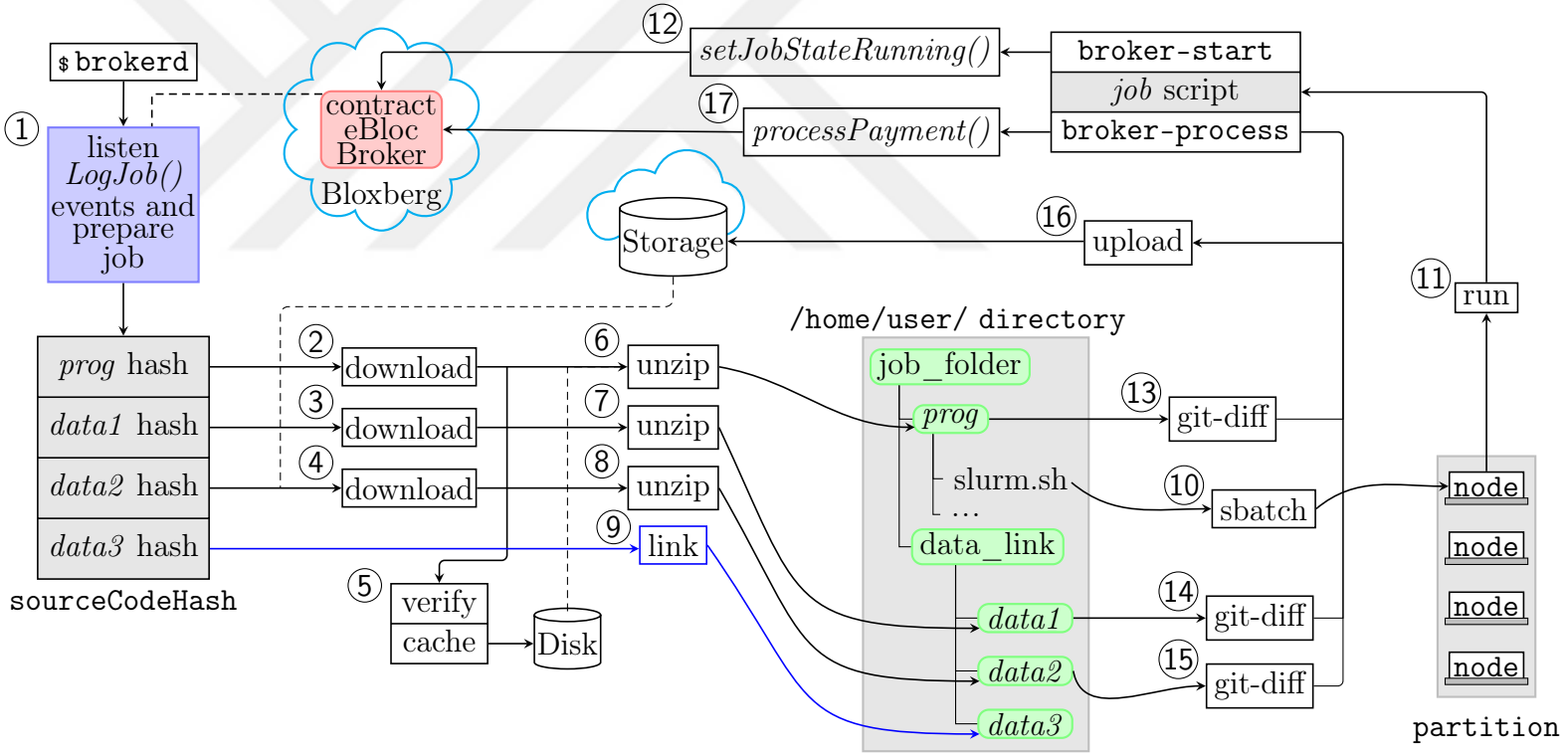


Figure 3.5. A breakdown of the steps the provider takes to receive a job, download the job's corresponding data, submit the job to Slurm, upload generated results to the cloud storage, and obtain the payments is outlined.

Step ① listens to eBlocBroker's *LogJob()* events, organizes jobs, and calls a new thread. Steps ②, ③, and ④ download *prog*, *data1*, and *data2* from the cloud storage. Step ⑤ verifies downloaded data files. If the verification is successful, `brokerd` will cache the corresponding data files. Steps ⑥, ⑦, and ⑧ extract downloaded data into the job folder that is created in the user's home directory. Step ⑨ links the directory of the requested *data3* read-only; hence `git-diff` will not be applied to it. Step ⑩ submits the Slurm script to the compute node using `sbatch`. The submitted job starts its execution in step ⑪. In step ⑫, the `broker-start` script runs right before the job, which calls the eBlocBroker's *setJobStateRunning()* function to update the job's state. When the job is completed, the `broker-process` script runs, which generates a patch for each corresponding data file using `git-diff` in steps ⑬, ⑭, and ⑮. Step ⑯ compresses and uploads them into cloud storage. Step ⑰ calls the eBlocBroker's *processPayment()* function to obtain the deposit of the job.

The `brokerd` performs the following steps:

Step (1). The `brokerd` checks whether the required programs, such as IPFS and Slurm, are running in the background and whether the correct credentials are used to login into B2DROP and Gdrive. It also checks whether Brownie is connected to Bloxberg and the provider's Ethereum account is unlocked that is used by Brownie.

Step (2). The `brokerd` continuously loops. On each iteration, it listens for a *LogJob()* based upon the Ethereum address of the provider, which is triggered by each job submitted by a requester to the provider's Ethereum address using the *submitJob()* function to eBlocBroker.

Step (2.1). As soon as `brokerd` catches *LogJob()* events, it processes them from the latest received job's block number to the latest block number on the blockchain. This approach maintains fairness in job assignment while retaining the first-in, first-out policy.

Step (2.2). In each iteration, `brokerd` generates an MD5 hash for the Ethereum address of each job owner to create a unique string of 32 characters, which will be the username of the job owner in Slurm and the Linux system. Suppose the generated username does not match with previously created usernames. In that case, `brokerd` creates a new user with a corresponding username in Slurm and the Linux system with a home directory that has read, write, and execute permissions only for the created user, preventing other users from accessing. For the purpose of performing this, the set file access control lists (setfacl) command is used as `setfacl -R -m user:$USERNAME:rwx $USER_HOME_DIR`.

Step (3). When there are idle CPU cores to run the job, `brokerd` passes metadata of the job such as its *key*, *index*, *cacheType*, and requested core to a function and starts it in a new thread.

Step (3.1). Afterward, the created thread iterates over the metadata of the received job's data files. As stated previously, the metadata of a data file consists of whether it is cached, its size and storage duration, etc. Within the job owner's home directory, each job is stored under a unique directory name, which is generated by concatenating the key and index values of the job. In each iteration, the thread first downloads the received job's data files, which have not been stored, from their shared cloud storage into the provider's storage.

Step (3.2). After downloading all the data files corresponding to the job to the provider's storage, their hashes are verified to compare the downloaded data files corresponding hash with the requester's provided hash. Next, whether the downloaded data file's size exceeds the user's requested amount is also checked. For the purpose of the integrity of download data files, if all the hashes are successfully verified, then the *setDataVerified()* function is called with the *sourceCodeHash* of the received job. If not, then the provider rejects the job and calls the *refund* function to pay a full refund except the used transfer cost to the job owner, and `brokerd` resumes from step 2.

Step (3.3). Moreover, the called thread function extracts and copies the received job's data files, which are already compressed by the requester, from their stored directory into the job owner's directory that is created for this job.

Step (4). The `brokerd` switches to the account of the job owner and sends the corresponding job's batch script to Slurm. A submitted job may exhaust various resource limits, such as the limit on the number of processes that may be created, the CPU time resource limit, or the network resource for downloading data. To prevent this possibility, `brokerd` uses the `unshare -r -n` command as a wrapper around the job script to run without network access and submits the job to Slurm using the `sbatch` command with the `-n` parameter indicating the number of CPUs required. Afterward, it updates the submitted job's time limit using the `scontrol update jobid=<job_id> TimeLimit=<time>` command. The `brokerd` performs this approach for all the received jobs, which prevents the job owner's source code from being modified or read by other users, and exhausts requested resource limits.

Step (5). Slurm calls a script program (`broker-start` or `broker-process`) when the state of a job (which is submitted to Slurm in step 4) is changed. This approach is performed by configuring the `MailProg` option in `slurm.conf`. The `MailProg` option is originally intended to be a fully qualified path of a program used to send emails upon job state changes to the users, but that program can do anything else. Its path is set to our script program that manages to call `broker-start` or `broker-process`.

Step (5.1). The `broker-start` script is triggered immediately after each job is started. It calls the *setJobStateRunning()* function to set the job's *stateCode* field to *running* and *startTimestamp* field to the Unix timestamp of when the job started.

Step (5.2). Conversely, the `broker-process` script runs right after each job is executed or is cancelled at the user's request. First, it generates a patch for each corresponding data file, which is the changes staged for the next commit relative to the latest commit, using the `git-diff` command. Then it compresses and uploads the patch into IPFS, or the user's B2DROP or Google Drive shared repository based on

the job's given *cloudStorageID*. Hence, only uploading changes as a patch will reduce bandwidth consumption and the cloud repository storage usage, which saves time in uploading files and provide efficient storage utilization.

Step (6). Afterward, it calls the *processPayment()* function to obtain the cancelled or completed job's cache, computation, and data transfer deposits. It shares the hash of the changes made by IPFS with the user through the *processPayment()* function if *cloudStorageID* is assigned as either IPFS or IPFS with GPG encryption.

Step (7). Lastly, `brokerd` continues from step 2, where it resumes to listen to the *LogJob()* events of the new jobs.

Likewise, when a requester cancels a job using the *refund()* function, it triggers a *LogRefund()* event on eBlocBroker. The `brokerd` runs another process, with the same approach, called `broker-canceld`, in the background, which cancels requested ongoing jobs and refunds them. The program performs the following steps:

Step (a). The `broker-canceld` listens to the *LogRefund()* events.

Step (b). As soon as `broker-canceld` catches a *LogRefund()* event, it sends a cancellation signal for the required job, which runs under the control of Slurm, by using the `scancel <job_id>` command. Cancelled job triggers the `broker-process` scripts and applies steps 5.2 and 6.

Step (c). Lastly, `broker-canceld` continues from step a, where it resumes listening to the *LogRefund()* events for the recently cancelled jobs.

3.5. Smart Contract Implementation

The signatures of the core functions of eBlocBroker are illustrated in Figure 3.6, and the event and various core functions of eBlocBroker are described in Table 3.5. Note that for test purposes, we have implemented eBlocBroker smart contract inheriting ERC20 smart contract. The more detailed explanations of the fundamental functions

of eBlocBroker and their arguments are as follows:

```
pragma solidity ^0.7.6;

contract eBlocBroker is ERC20 {
  function registerRequester(bytes32 gpgFingerprint, string memory gmail, string
    memory fcID, string memory ipfsAddress) public returns (bool);
  function registerProvider(bytes32 gpgFingerprint, string memory gmail, string
    memory fcID, string memory ipfsAddress, uint32 availableCore,uint32[] memory
    prices,uint32 commitmentBlockDur) public returns (bool);
  function updateProviderInfo(bytes32 gpgFingerprint, string memory gmail, string
    memory fcID, string memory ipfsAddress) public returns (bool);
  function suspendProvider() public returns (bool);
  function resumeProvider() public returns (bool);
  function submitJob(string memory key, uint32[] memory dataTransferIn,
    Lib.JobArgument memory args, uint32[] memory storageDuration,bytes32[] memory
    sourceCodeHash) public;
  function setJobStateRunning(string memory key, uint32 index, uint32
    startTimestamp) public returns (bool);
  function setDataVerified(bytes32[] memory sourceCodeHash) public returns (bool);
  function registerData(bytes32 sourceCodeHash, uint32 price, uint32
    commitmentBlockDur) public;
  function removeRegisteredData(bytes32 sourceCodeHash) public;
  function processPayment(string memory key, uint32 index, uint32 elapsedTime,
    bytes32 resultIpfsHash, uint32 endTimestamp, uint32 dataTransferIn, uint32
    dataTransferOut) public;
  function depositStorage(address dataOwner, bytes32 sourceCodeHash) public
    returns (bool);
  function refund(address provider, string memory key, uint32 index) public
    returns (bool);
  function authenticateOrcID(address user, bytes32 orcid) public returns (bool);
  // Event definitions and getter functions
}
```

Figure 3.6. Signatures of the eBlocBroker smart contract's fundamental functions.

- The *authenticateOrcID()* function takes the following arguments: The *user* argument is an *address* value denoting the Ethereum address of the enrolled requester or provider. The *orcid* argument is a *bytes32* value indicating an authenticated ORCID iD. Python Flask server runs on our backend to perform the main authentication process. The owner of eBlocBroker only performs this. The user's ORCID iD is retrieved via ORCID API, and the Ethereum address is retrieved via MetaMask. These are passed as arguments into the function on our backend, which registers them into the smart contract.
- The *submitJob()* function sends a job to a provider. The *key* argument contains the hash of the source code as a unique identifier for the requester's job. The *dataTransferIn* argument is an array of *uint32[]* values, denoting each data file's

size in megabytes (MB) to download the requester's data from cloud storage into the provider's storage. The *dataTransferOut* argument is an *uint32* value denoting the size of the data files in MB required to upload output files generated by the requester's job into the cloud storage. The *sourceCodeHash* argument is an array of *bytes32[]* values that are MD5 hashes for each of the corresponding source code and data files. The *storageDuration*[1] argument is an array of *uint32[]*, which denotes the hours the provider will store each downloaded data file of the received job. The *args* argument is a structure containing additional values as follows:

The *provider* argument is the Ethereum address of a provider that is requested to run the job.

The *cloudStorageID*[1] argument is an array of 8-bit unsigned integers (*uint8[]*), which denotes whether each data file is stored and shared using either Google Drive, B2DROP, IPFS, or IPFS with GPG encryption.

The *cacheType*[1] argument is an array of *uint8[]*, which denotes whether the provider will store each data file privately within the job owner's home directory or publicly for other requesters' access within a shared directory for all the requesters.

The *dataPricesSetBlockNum*[1] argument is an array of *uint32[]*, which denotes whether each data file except the source code is assumed to be the provider's registered data. If it is zero, then the requester's data will be considered, which is already stored or downloaded from the cloud storage. Otherwise, it should be the most recently set block number for the requested provider's price for the registered data file corresponding to each *sourceCodeHash*. It also records *storageDuration* for each of the corresponding *sourceCodeHash*.

The *core* argument is an unsigned 32-bit integer (*uint32*) value containing the number of CPU cores requested to run the job. The *runTime* argument is an *uint32* value containing the expected *total CPU time* in minutes to run the job. The *priceBlockIndex* argument is an *uint32* value containing the block number

[1] *dataTransferIn*, *cloudStorageID*, *cacheType*, *dataPricesSetBlockNum*, and *storageDuration* are considered for each of the corresponding *sourceCodeHash* of the data file.

when the requested provider sets its most recent prices.

- The *refund()* function returns the deposit of the job to its owner, if it is still pending or is incomplete an hour after its expected run time is ended. It triggers a *LogRefund()* event on the blockchain, which the provider will catch to cancel the job if it is running. Note that if the requested job was refunded, the provider cannot reclaim the deposit using the *processPayment()* function.
- The *registerProvider()* function registers the information of a provider. Providers should denote their prices in units of *USDmy* token. It takes the following arguments from the provider: The *gpgFingerprint* argument is a *bytes16* value containing a public key fingerprint of the key that is used by the GNU Privacy Guard (GPG) to import the keys from a public key server. GPG is used to encrypt or decrypt files, which is an open-source version of OpenPGP encryption and signing tool that uses the public key (asymmetric) cryptography. The *gmail* argument contains a Google account for sharing the requester's repository with the provider through Google Drive. The *fcID* argument contains a Federated Cloud ID, which is a unique identifier for sharing the requester's repository with the provider through B2DROP. The *ipfsAddress* argument contains the IPFS multiaddress of the IPFS node, which the requesters use to open a direct connection to share files. The *availableCore* argument is a 32-bit unsigned integer (*uint32*) value containing the available CPU cores. The *prices* argument is a structure containing four *uint32* values: price per *core-minute*, price per *storage-hour*, price per *cache-megabyte*, and price per *transfer-megabyte* values. The *commitmentBlockDur* argument is an *uint32* value containing the duration of the committed prices. Note that *msg.sender* is linked to the provider's registered record.
- For each job, the *processPayment()* function maintains the payment process between the provider and the requester for each submitted job. It charges the requester for the calculated amount incurred for computation, data transfer, and cache usage services. It also transfers the funds to the service provider and returns, if any, changes back to the requester. The *processPayment()* function is called immediately when the provider completes a job. It checks whether the job's

deposit was transferred to the provider and whether the job's ending time exceeds its deployed block's timestamp value. After this, it applies the `Overlap-Check` algorithm, as explained in Section 3.5.3. The *processPayment()* function takes the following arguments: *key* and *index*, which are used as mapping keys to a structure that contains the job's information, elapsed (real) time of the job in minutes (*elapsedTime*), the hash of output files generated by IPFS (*resultIpfsHash*), the Unix timestamp of when the job ended (*endTimestamp*), MB is used to download the requester's job from cloud storage into the provider's storage (*dataTransferIn*), and MB is used to upload output files generated by the requester's job into the cloud storage (*dataTransferOut*).

Table 3.5. List of eBlocBroker smart contract's fundamental and event functions with their descriptions.

Function	**Description**
registerRequester	Registers or updates the information of a requester.
updateProviderInfo	Updates the information of a provider, i.e., Gmail and IPFS address.
updateProviderPrices	Updates the number of available CPU cores, the provider's prices, and their commitment block duration.
removeRegisteredData	Deregisters a data file using its *sourceCodeHash*.
registerData	Registers a data file using *sourceCodeHash* with a given price and commitment block duration.
updataDataPrice	Updates the price of the registered data file using its *sourceCodeHash*.
suspendProvider	Suspends a provider that will deny receiving jobs to run.
resumeProvider	Resumes a suspended provider to receive jobs.
depositStorage	Transfers the requested data file's storage deposit to its owner if its *storageDuration* is expired.
refundStorageDeposit	Refunds the storage deposit to the requester if the *storageDuration* of the given *sourceCodeHash* has expired.

Table 3.5. List of eBlocBroker smart contract's fundamental and event functions with their descriptions. (cont.)

Function	**Description**
setDataVerified	Sets the corresponding job's data files' received block number with the current block number and also sets those data files verification status as true. This transaction can be performed only by the provider requested to run the job.
setJobStateRunning	Sets the state of the job as *running* and also records its start time. It takes the following arguments: *key* and *index*, which are map keys to retrieve the job's data from eBlocBroker and the Unix timestamp of when the job started (*startTimestamp*).
LogReceipt	Logs the completed job's data processed by the *processPayment()* function.
LogJob	Logs the data processed by the *submitJob()* function related to the submitted job.
LogProvider	Logs the registered providers' data processed by the *registerProvider()* function.
LogRequester	Logs the registered requesters' data processed by the *registerRequester()* function.
LogRefund	Logs the cancelled or refunded job's data recorded by the *refund()* function
LogSetJob	Logs the data processed by the *setJobStateRunning()* function related to the updated jobs.
LogDepositStorage	Logs the storage deposit transferred to the corresponding provider for the requested job, processed either by the *submitJob()* or *depositStorage()* function.

Lastly, the event functions store the arguments passed in a transaction log when it was emitted. The `brokerd` listens and subscribes to these triggered events through an Ethereum RPC node provided by Bloxberg. During filter-event handling, the indexed parameter is used, which allows users to search events using the indexed parameters as filters. The indexed parameters are configured as *provider* and *owner* variables, which are the Ethereum address of the provider requested to run the job and the requester who submitted the job, respectively. Note that in Solidity, mapping types use syntax, `mapping(key => value)`, which is similar to hash tables. Mapping does not store the key, instead, its *keccak256* hash is used to look up the value.

3.5.1. Job Submission Conditions

As of June 5th, 2023, when eBlocBroker was deployed, since the time interval between blocks in Bloxberg is nearly six seconds, eBlocBroker assumes that the duration of 10 (1 minutes/B^{time}), 60 (1 hours/B^{time}), and 14400 (1 days/B^{time}) consecutive blocks will take a minute, an hour, and a day, respectively, to complete.

The *submitJob()* function accepts the requested job if all of the following conditions hold true:

- The owner of the job and the requested provider are registered to eBlocBroker, and their ORCID iD is authenticated.
- The requested provider must resume accepting jobs.
- The deposit in *USDmy* tokens is enough to pay for the cost of the job calculated using our cost model as described in Section 3.3.
- The given value in the *core* field is positive and less than or equal to the requested provider's available CPU cores.
- The given value in the *runTime* field is positive and less than or equal to 1 days/B^{time}. The primary reason for this check is to limit the run time.
- Length of the arrays corresponding to *sourceCodeHash* is equal to each other.
- Items of *cloudStorageID*, except the first one, must be equal to the first item of

cloudStorageID, IPFS, or IPFS with GPG, illustrated in Figure 3.7.

- If the data that has been requested is already stored by the provider, it cannot be requested from any other cloud storage.

3.5.2. Provider's Price Commitment

When a provider registers itself or a dataset, it commits to the price for the predefined period, thus protecting the requester from any price change. This approach enables eBlocBroker to handle conflicts where a provider attempts to change its previously set prices within the commitment duration. In such a case, the cost is calculated with the price before the provider's update takes place. If the provider changes its prices, then the prices for the next commitment cycle will be updated.

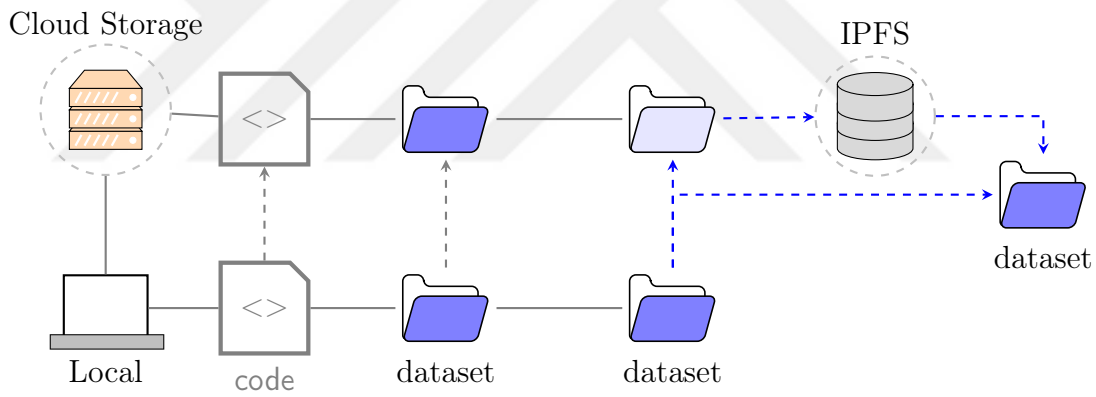


Figure 3.7. Transferring source code and datasets from users to providers through remote cloud storage and IPFS.

Figure 3.8 portrays the following scenario: Let's assume all the submitted transactions are deployed in the submitted block number. Here, Bob (provider) registers at block number 50 with a commitment duration of 150 blocks, where eBlocBroker pushes the block number into a mapping T. Every element of T points to a structure of the provider's registered information and other price details. Next, at block 250, Bob updates the price with a commitment duration of 100 blocks. The contract adds up the last committed provider's block number with its commitment duration iteratively until it is greater than the block number and appends this number to the list. In our

example, this would be 350, which is 50 + 150 + 150. When a requester submits a job, our algorithm iterates through the T, proceeding from tail to head until it reaches a committed block where the latest block number is greater. Once Alice (requester) submits a job at block 300 since it was submitted within the time frame of block 50 to 350 exclusive (illustrated as the commitment duration number one in Figure 3.8), the applied price will be the one set on block 50. Afterward, when Alice submits another job at block 500, the updated prices at block 250 (which has 350 as a committed block) will be applied.

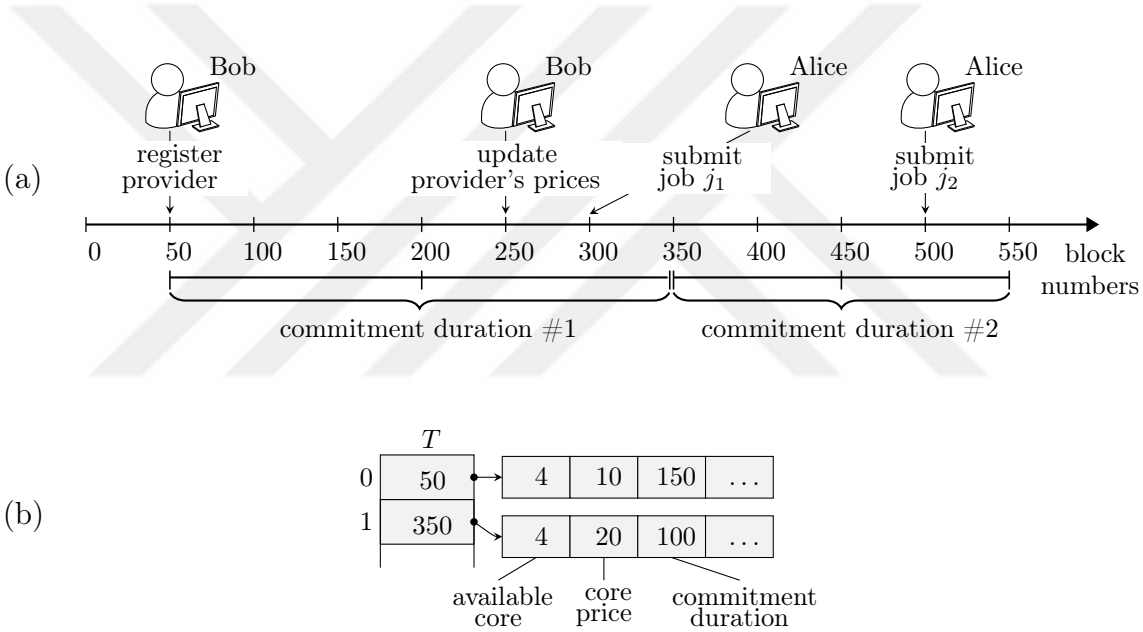


Figure 3.8. Example of price commitment of a provider. (a) Timeline of when Bob commits prices, and in its commitment duration; and when Bob updates the price while Alice submits jobs. (b) Mapping T and its values when Bob updates provider's prices at block 250.

3.5.3. Overlap Check

Intervals are useful for representing jobs that each occupy a continuous period of time. We can represent a job interval $[t_j^b, t_j^e]$ as an object x, with attributes $x.low = t_j^b$ (the low endpoint) and $x.high = t_j^e$ (the high endpoint). We express that intervals x and x' overlap if $x \cap x' \neq \emptyset$, which is, if $x.low \leq x'.high$ and $x'.low \leq x.high$. For

the `Overlap-Check` algorithm to work correctly, each job's precise Unix timestamp of when it started and ended is required. The providers supply this information, which is obtained from Slurm's active database in date format and converted into a Unix timestamp.

The `Overlap-Check` algorithm operates under the *processPayment()* function, which takes *startTimestamp* (t_j^b), *endTimestamp* (t_j^e), and *core* (w_j) arguments for each completed job j, and checks between t_j^b and t_j^e whether the overlapping intervals' aggregated *core* value exceeds the *availableCore* of the provider when the job is submitted. If the check returns false, the (t_j^b) and (t_j^e) are archived as time intervals for the job. If the check returns true, eBlocBroker will deny the transfer of funds to the provider in order to prevent unfair gains by claiming to provide computational resource usage beyond the number of available CPU cores. A full refund for the cache, computation, and data transfer deposit of the job is paid; hence the storage and provider's dataset usage deposit for the data files of the refunded job remains intact, which should be reclaimed by the user using the *refundStorageDeposit* function.

The `Overlap-Check` algorithm is implemented by a *sorted* singly linked list L using a mapping M. Each element of M points to an interval object that contains an endpoint of a job that is t_j^b or t_j^e (*endpoint*), the number of CPU cores requested to run the job (*core*), and an index points to the next node on the linked list (*next*). Note that if the endpoint of a job is t_j^b, then the *core* is considered negative. The algorithm will insert each interval into the linked list based on its endpoint in ascending sorted order to keep the linked list sorted at insertion since sorting would bring prohibitive gas costs. First, the algorithm iterates through the linked list proceeding from tail to head until it reaches an index where the interval's endpoint is greater than or equal to t_j^e. If it's equal, then the *core* is added to the interval's core field; if not, then the given interval is inserted into the linked list, and an accumulator variable is set (*carriedSum*) to the *core*. Then the algorithm continues to iterate towards the head of the linked list from where it left off while adding *core* at each interval structure into (*carriedSum*) until it reaches an index where the interval's endpoint is less than or equal to t_j^b. If equality holds,

then the *core* is added to the interval's core field; if not, the interval, which contains the job's endpoint as t_j^b, is inserted into that index. While iterating, if *carriedSum* exceeds the *availableCore* of the provider, then the *processPayment* transaction call, which handles the transfer of the job's deposit to the provider, will be rejected and rolled back using Solidity's *revert()* function. Since the linked list is already sorted, the `Overlap-Check` algorithm runs in $O(m)$ time, where m is the number of endpoints. The worst-case time complexity for the checking mechanism is linear in the number of intervals.

The `Overlap-Check` algorithm is demonstrated with the following example. A provider is registered with 128 available CPU cores. Afterward, the *processPayment()* function is called with the following transaction calls (10, 40, 16), (10, 40, 16), (7, 50, 32), (10, 55, 32), and (15, 70, 2) respectively. All transactions are deployed successfully until the latest transaction call. At the latest transaction call, the interval containing the job's endpoint (endpoint: 70, core: 2) is inserted into the linked list, and *carriedSum* is assigned to 2, which is the inserted interval's number of CPU cores. Next, during the algorithm's iteration, *carriedSum* reaches 130, which exceeds the *availableCore* of the provider when the job is submitted. For that purpose, the transaction will be rolled back; hence the inserted interval on the latest transaction call is removed from the linked list. Figure 3.9 illustrates the generated linked list after all the transactions are deployed on the given example.

3.6. Data Transfer and Reusability

Our automated script program, `brokerd`, coordinates linking the source code to the relevant datasets and ensures reusability by storing the requested data files for the received job. The data file of the received job could be a source code with its Slurm job script to run the job or a data file. Disk size and network usage are calculated in MB. Suppose the data files of the received jobs are received through B2DROP or Google Drive. In that case, they are stored privately within the job owner's home directory, publicly for another requester's access within a shared directory, or not stored based on

the requester's claim. Providers maintain the downloaded data files in their disk until their storage duration expires. If a data file, which has a remaining time for its storage duration, is requested to be used again by its owner, then no additional cache, data transfer, and storage cost will be applied. If different users want to use that data file, its owner should share it publicly, which the provider should verify. The `brokerd` runs a process called `broker-gc` as a garbage collector in the background that continuously checks the *storageDuration* of the stored data files every hour and removes them if their *storageDuration* has expired.

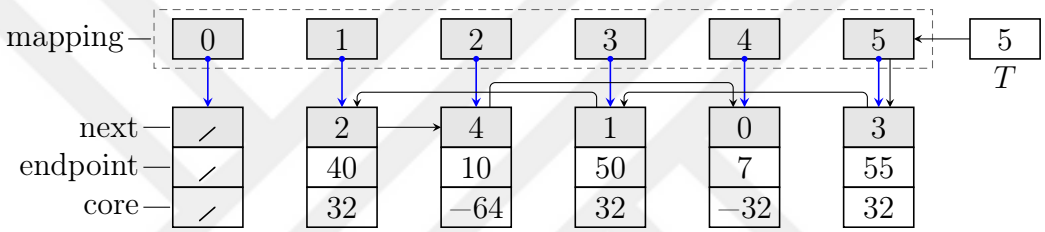


Figure 3.9. Linked list using a mapping after all the given transactions on the given example are deployed. Stored keys correspond to the mapping indices at the top; the arrows show how to interpret them. The variable T keeps the index of the tail.

Our script programs are based on the Python client library to communicate with each of the B2DROP, IPFS, and Google Drive storage facilities, as described as follows:

- *IPFS*: Files stored on an IPFS node will be available to the rest of the P2P network as long as that node is running and publicly accessible. Requesters wishing to ensure the privacy of files shared via IPFS may encrypt the files using GPG with the fingerprint of the desired provider before uploading to IPFS. Providers will also encrypt the output files with the fingerprint of the job owner before uploading them into IPFS to ensure file sharing using end-to-end encryption. IPFS nodes can consume high memory usage as they continuously connect to more nodes. For the purpose of isolating the IPFS node from the rest of the network, the IPFS daemon is launched with the `--routing=none` flag to make the node undiscoverable by other nodes via disconnected from the distributed hash table that supports routing and discovering contents and peers on the IPFS network.

Hence, the IPFS node can still form outgoing connections to other nodes using the `ipfs swarm connect <ipfsAddress>` command performed by our script program before submitting a job.

- *B2DROP*: The B2DROP service is used to store and exchange data. Requesters share their folder, which contains data of the submitted job, through B2DROP with each other using requester's (i.e., provider's) Federated Cloud ID using pyocclient [64], which is a Python client library for ownCloud.
- *Google Drive*: The Google Drive CLI client is used to access files on Google Drive, allowing requesters to share their folder containing the data of the submitted job through Google Drive using the provider's Gmail address. It also permits providers to download the shared folder and upload files onto it. Thus, only the provider has access to the folder, which is shared with itself, to download the requester's source code and data files and upload the changes made by the corresponding job through B2DROP or Google Drive.

3.7. Results

Before testing in real environments, the eBlocBroker smart contract was developed and tested using Brownie, and the contracts were implemented using Solidity version 0.7.6 and Ethereum Virtual Machine in Istanbul version. 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 is tested using a requester node to submit jobs and three provider nodes to obtain and run the jobs. The requester node and all the provider nodes are created on the Digital Ocean cloud platform, each of which has 4 vCPUs and 8 GB memory and allows file sharing using B2DROP, IPFS, or Google Drive cloud storage. All provider nodes run our `brokerd` daemon process to receive jobs, coordinate their access to other shared resources, and upload their results into cloud storage and submit them to real-machine Slurm in order to execute them. In our test, all providers have the same fixed prices for all except the fee for the datasets.

We first deployed eBlocBroker smart contract on Bloxberg and tested it using two types of synthetic CPU workloads, as described below.

(i) The first workload (Workload-1) tests to submit a job that only requires the source code of the job. It uses the NAS Parallel Benchmarks [65], a small group of programs targeting the performance evaluation of parallel supercomputers. Since providers' prices are the same, all providers' calculated costs for NAS jobs will be the same. One of the NAS serialized benchmarks in Class B (Block Tridiagonal solver, Scalar Pentadiagonal solver, Unstructured Adaptive mesh, and Lower-Upper Gauss-Seidel solver) is selected randomly.

(ii) The second workload (Workload-2) helps to evaluate executing source code together with pre-cached and not cached datasets on the provider. It also tests to select the cheapest provider with the cached dataset on its storage. As the source code, the *cppr* (colored parallel rush-relabel algorithm) [66] is used, which runs with additional datasets. All three providers have the same 12 medium-size datasets, of which only four distinct ones from each other have lower prices. During each job submission using this workload, its script is generated as follows:
    - Three *cppr* processes run consecutively with different randomly selected datasets.
    - Three datasets were selected: two from the provider's registered data and one from the requester's local storage.

Note that Workload-1 and Workload-2 are similar to the example given in Section 3.3.1. The script program we utilize to submit jobs from the requester node executes the following task:

- One hundred synthetic requesters within the requester node continually submit one of the workloads randomly for 22 hours.
- Job submissions are made in a batch; on each batch, two jobs are submitted to each provider. Between batches, the script sleeps for an interval randomly chosen between 240 to 480 seconds.

- All source codes contain a Slurm script that is used to execute them.
- Each source code and data file are compressed separately before being uploaded into the cloud storage.
- The cloud storage service is randomly selected for each job submission, with each job being shared between the requester and the provider.
- Our cost model calculates the cost of the job for each provider and then chooses the provider with the lowest price.

Table 3.6. Description of the number of jobs submitted and completed for each workload during the test.

Provider	**No. Jobs Completed/Submitted**	
	Workload-1	**Workload-2**
A	84/86	63/68
B	76/77	75/79
C	84/84	70/71

Alongside testing the correctness of our smart contract, we also tested the robustness of our Python scripts and `brokerd`. The results of the tests show that the providers have successfully completed all the submitted tasks, except for some. The download process of large files or Python packages may experience failure during execution, and an unresponsive Bloxberg node for a short time, leading to unfinished jobs. Job owners receive a refund of the fees associated with these dangling jobs, thus they are able to continue running these jobs as their information is stored. Furthermore, for the completed jobs, their generated results as patches are uploaded into the cloud storage, and providers obtain the deposit using our smart contract. Information about the test results is shown in Table 3.6. The datasets generated during test runs are available here [67].

Figures 3.10-3.13 illustrate various analyzes of the test results. In Figure 3.12, the waiting time is from when the job request is sent via the *submitJob* function to when the job starts in Slurm. In Figure 3.13, the completion time is from when the job's request via the *submitJob* function is deployed until the job is completed in Slurm and its *processPayment* function request is deployed.

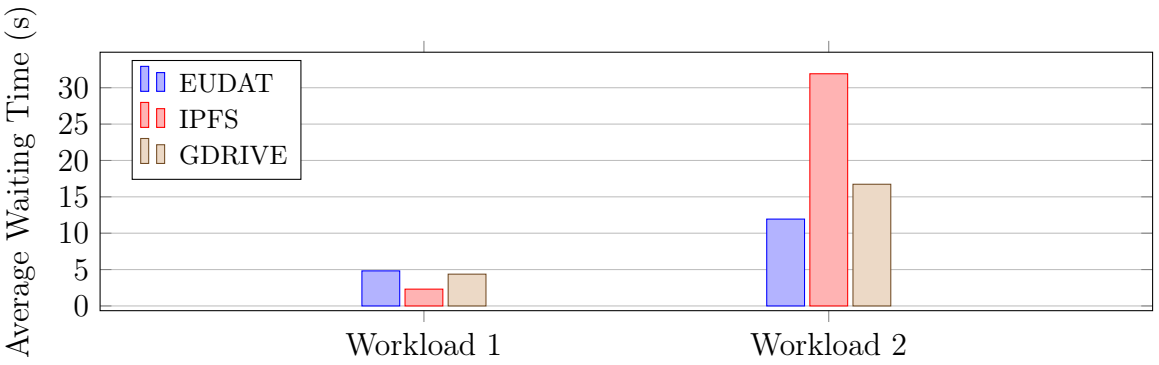


Figure 3.10. Comparison of job waiting times (in seconds) for our workloads.

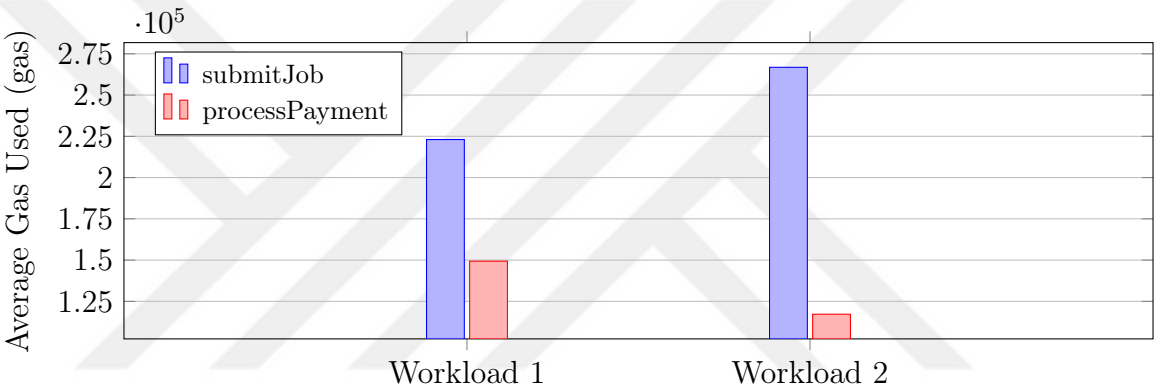


Figure 3.11. Comparison of gas cost for submitJob and processPayment.

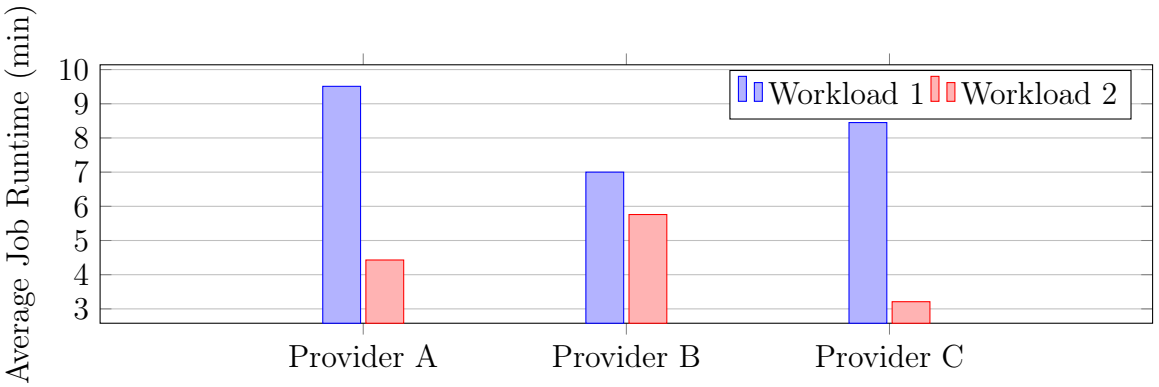


Figure 3.12. Comparison of job run times (in minutes) for our workloads in Slurm.

3.7.1. Gas Consumption Limitation

To increase efficiency, each block holds a set (group) of transactions, along with a fingerprint (hash) of the previous block. This creates a hash link to the previous group

and a timestamp. This interconnected, hash-linked structure forms the blockchain. The main blockchain is populated with blocks through a consensus procedure. The generation of valid blocks, known as mining, involves demonstrating proof of work. The blockchain's crypto-currency is distributed to the miners as a form of payment for their work. For instance, the Bitcoin blockchain produces new blocks approximately every 10 minutes through mining. Conversely, Ethereum has transitioned to a proof-of-stake mechanism, which involves validators instead of miners. As of February 18th, 2024, the Ethereum blockchain has an average block time of around 12 seconds [68].

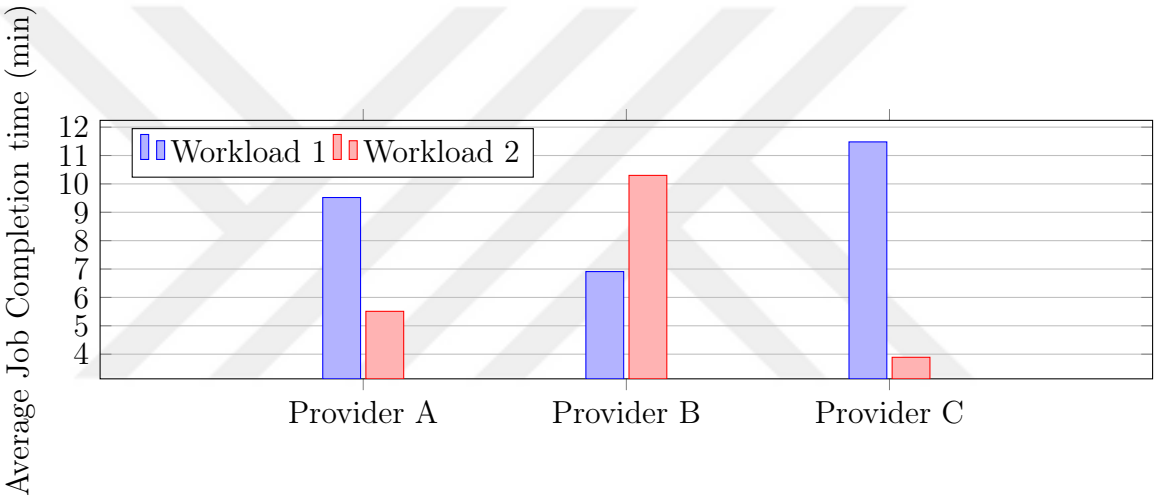


Figure 3.13. Comparison of job completion times (in minutes) for our workloads in Slurm.

In addition to its consensus mechanism, Ethereum introduces the concept of "gas" to manage transaction costs and computational resources within the network. The Ethereum blockchain uses a pay-to-play model, which charges senders of transactions with a small amount of Ether, i.e., gas, to execute a transaction on the network. Note that 1 Ether is 10^{18} Wei. Gas is the execution fee for every instruction executed to run a transaction. The *transaction* is a write-operation that will continually change the state of the Ethereum blockchain. The transaction is propagated to the network, processed by miners, and, if valid, is deployed on the Ethereum blockchain.

On the other hand, *view* functions do not require any gas to be executed. A *call* is a local invocation of a smart contract function defined by the *view* keyword. It

does not broadcast on the Ethereum blockchain and is a read-only call, and reads the return value of the smart contract function instantly. In addition, variables defined by the *memory* keyword are used to hold temporary values, which are erased between (external) function calls. On the contrary, variables defined by the *storage* keyword are persisted on the blockchain; hence the next time the smart contract executes some code, it has access to all the data it previously stored on its disk.

When executing a smart contract function, the global variable *msg.sender* contains the caller's address. The global variable *msg.value* contains the amount of payment received together with the contract call. Note that in order for a smart contract function to be able to receive payment, it should be declared as a *payable* function. In order to ensure accuracy, timestamps are not used as time units on the blockchain. Rather, we rely on block numbers to establish and monitor deadlines.

The deployment of transaction executions is bounded by the block gas limit [4], which changes dynamically. If a block fits only one transaction, then the gas limit per transaction is, in theory, the same as the block gas limit. The gas limit differs from blockchain to blockchain. As of September 26th, 2023, the Ethereum average gas limit is around 30M, and Bloxberg has a lower gas limit, which is fixed at 10M. In order to reduce gas costs, we have designed the eBlocBroker smart contract by applying the following approaches:

- Because events are not required on-chain, they are used to store most data; hence using them is considerably cheaper than contract storage.
- For complex functions, internal functions are implemented. When you call an internal function, references of those parameters are passed and not copied into memory again. Because of this reason, calling internal functions is cheaper.
- Copy data from storage to memory, apply all modifications to the memory variable, and finally update the storage with the changes.
- In certain data structure definitions, use mapping over arrays.
- Using the tight variable packing pattern that optimizes gas consumption when

storing or loading statically-sized variables.

In Equations 3.3 and 3.4, the estimated costs of gas are calculated from the Ganache platform. The gas cost estimation function for the *processPayment()* is given in Equation 3.3, where x is the size of D_j. The average gas consumption value of the *processPayment()* function without overlap is 146346. Every iteration of the `Overlap-Check` algorithm given in Section 3.5.3 will cost an additional 1000 gas based on its logic. Therefore, the gas cost will increase when the number of endpoints m increases. If m is more than approximately 30000 and 10000, then the transaction exceeds the gas limit for Ethereum and Bloxberg, respectively.

$$est_gas_processPayment(x) = 1000 \cdot x + 146346 \tag{3.3}$$

$$f(Y_{D_j}) = 196254 + \sum_{d \in D} 11636 \cdot [\![y_d = 0]\!] + 55065 \cdot [\![y_d > 0]\!] \tag{3.4}$$

The gas cost calculation function for the *submitJob()* function is given in Equation 3.4. If only a source code is submitted without a dataset, then the *submitJob()* function has average gas consumption of 196254. On top of that, each additional dataset has an average gas consumption of 11636. Suppose a new data file is requested to be stored on the provider. In that case, it costs an additional 43429 gas fee for storing its *receivedBlock* (block number when data is registered) and *storageDuration* in the contract. Due to this, our cost model will lead to the usage of more storage and additional instructions in the contract and, therefore, will incur higher gas fees. For this reason, it is not feasible to submit many data files requesting to be stored on the provider's storage.

Researchers can use Bloxberg at no cost and can get free Ether (called Berg) from Bloxberg's faucet [69]. As an alternative, Polygon is a layer 2 scaling solution created

to offer Ethereum developers more features and lower fees, making real costs much cheaper. eBlocBroker smart contract can also be deployed on Ethereum networks such as on Polygon. eBlocBroker is currently deployed and operational on the Bloxberg network. As of September 26th, 2023:

- The average gas price is 16.70 Gwei and the price of 1 ether (ETH) is 1590 USD [70].
- The average gas price is 140.50 Gwei and the price of 1 polygon (MATIC) is 0.52 USD [71].

Table 3.7. Average gas cost of the eBlocBroker's core functions obtained from the Bloxberg as a real test environment.

Method	**Gas Used (gas)**	**Ethereum USD cost**	**Polygon USD cost**
submitJob	264967	7.04	0.019
processPayment	143902	3.82	0.011
registerRequester	53248	1.41	0.004
registerProvider	263558	7.00	0.019
authenticateOrcID	44133	1.17	0.003
registerData	88566	2.35	0.006
setJobStateRunning	49495	1.31	0.004

Table 3.8. Average gas cost of the eBlocBroker's core functions obtained from the Bloxberg as a real test environment.

Method	**Gas Used (gas)**	**Ethereum USD cost**	**Polygon USD cost**
updateProviderInfo	33284	0.88	0.002
setDataVerified	42427	1.13	0.003
updateProviderPrices	76641	2.04	0.006
refund	67744	1.80	0.005
depositStorage	34472	0.92	0.003
refundStorageDeposit	26500	0.70	0.002
updataDataPrice	47359	1.26	0.003
removeRegisteredData	88566	2.35	0.006

In order to calculate the USD price of a gas fee, we multiplied the gas cost, given the Ethereum network's gas price and price in dollars together. The average gas consumption of the eBlocBroker's functions, along with their USD price in Ethereum Mainnet and Polygon, are given in Table 3.7 and 3.8, respectively.

3.8. Discussion

Since blockchain technology emerged, it has revolutionized not only the banking and financial sectors but also the field of e-Science. The formation of VOs for research, crowdfunding projects, citizen science, and resource management are examples of where blockchain technologies can be used. We demonstrate how a blockchain-based autonomous resource broker called eBlocBroker, which is a smart contract, can be used as blockchain technology to create a virtual market for computational and data resources for research communities. We also explain how our automated scripts assemble and share computational and storage resources. A configured Dockerfile file on our GitHub repository [72] is available for testing inside a Docker environment. The following list shows the features included in our future roadmap:

(i) Development of an ERC20 token on a separate smart contract.
(ii) Deployment of eBlocBroker smart contract on the Polygon main networks after making stress testing on Bloxberg.

4. AN AUTONOMOUS BLOCKCHAIN-BASED WORKFLOW EXECUTION BROKER FOR E-SCIENCE

4.1. Introduction

A scientific workflow [73] 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 [74] 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 through a single server, it leads to excessive network traffic, causing a performance bottleneck.

Our work aims to support blockchain through the implementation of decentralized autonomous organization-based workflows. 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) [75] scheduling and (*ii*) HEFT algorithm, as detailed in Section 4.3.5.

In this chapter, 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 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 [14] on computing clusters. 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.

The remainder of this chapter is organized as follows. In Section 4.2, we provide an overview of our system's architecture, including the interactions between components and the internal modules of the components. In Section 4.3, 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 Section 4.4, we present tests and experimental results from demonstrations of eBlocBroker on the Bloxberg scientific research blockchain. Section 4.5 concludes the chapter and discusses future work directions.

4.2. Blockchain-Based Autonomous Organization Model

Our approach (shown in Figure 4.2) consists of three phases: (*i*) inputting a workflow, (*ii*) partitioning it for scheduling, and (*iii*) submitting sub-workflows to computational and storage providers 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 Figure 4.1.

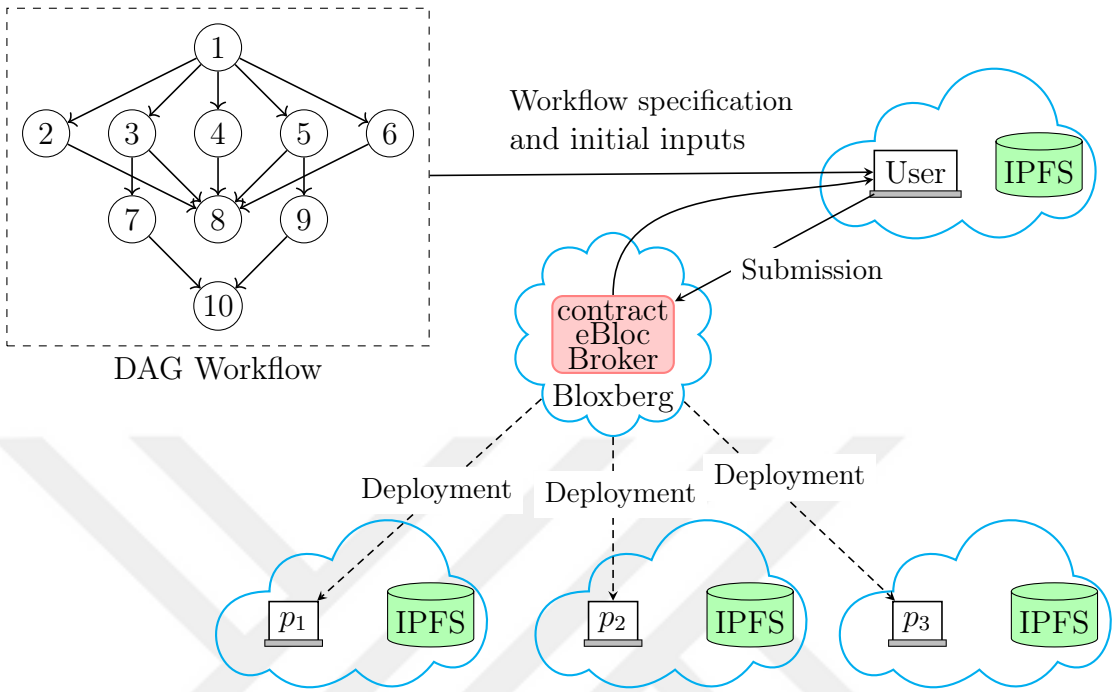


Figure 4.1. Overview of our distributed architecture and the interactions between the user and providers. Source code and data files are transferred between providers and users through IPFS.

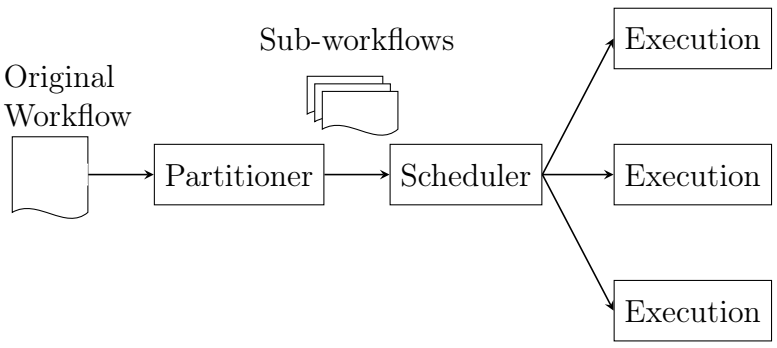


Figure 4.2. The process of dividing and organizing a workflow. The sub-workflows are allocated to various providers by the scheduler.

The engine utilizes our approach to execute the workflow by submitting transactions to the eBlocBroker smart contract deployed on the Bloxberg blockchain, 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. IPFS is used for data storage and sharing.

Figure 4.3 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.

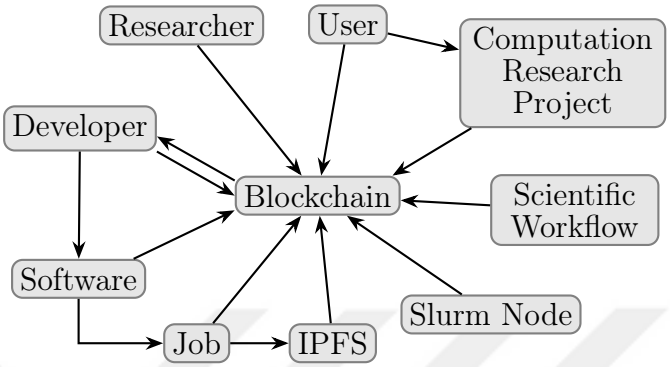


Figure 4.3. Interactions between diverse eBlocBroker entities.

The source codes of eBlocBroker and its script programs to interact with Bloxberg are available at [72] and source codes to carry out workflow engine tests are available at [76].

4.3. Workflow Support

We will investigate providing DAG workflow distribution support using eBlocBroker, which will enhance computational management automation. The scheduler optimizes cost and run-time performance by selecting the best resources for the sub-workflows. Our scheduler coordinates scheduling from the user side rather than the provider side, implementing both the HEFT and TGL algorithms. There is a difference from the original version because our experiments use a shared file system, and the data transfer cost within a sub-workflow is disregarded.

HEFT and TGL algorithms require that an application's characteristics, including run times, data sizes for communication between jobs, and job dependencies, are known a priori. Hence, a static model is used to represent the workflow, which is utilized by both HEFT and TGL algorithms. If variable output sizes are present, it is assumed that the user provides an estimate beforehand. In the scheduling phase for dependent

jobs submitted to the same provider, output data transfer costs are considered zero.

We model workflows using DAGs, where nodes represent computations and directed edges indicate data flow dependencies between nodes.

4.3.1. 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 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.

4.3.2. Random Graph Generator

In order to test the workflow engine algorithms and robustness of our system, random DAGs were generated and used, which were partitioned using HEFT and TGL algorithms. Figure 4.4 illustrates a randomly generated DAG.

NetworkX [77], a Python package, is employed to conduct graph analysis and create a workflow of jobs with 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 Section 4.3.5.

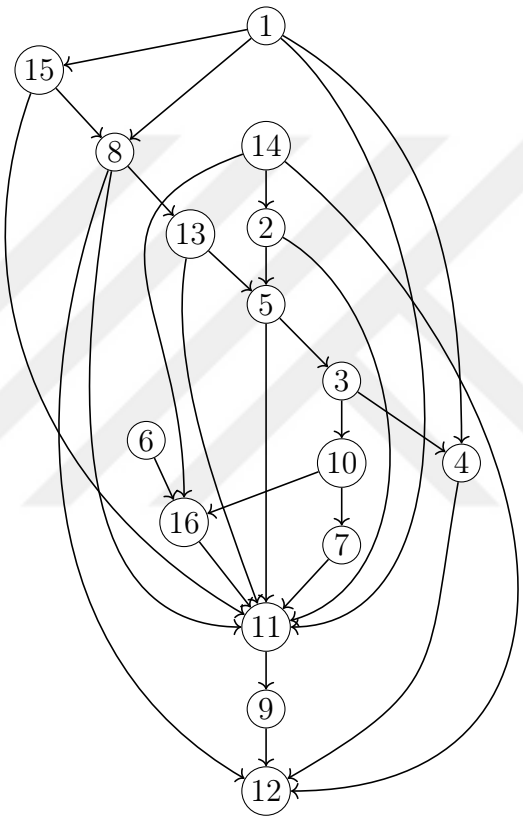


Figure 4.4. A randomly generated Directed Acyclic Graph (DAG) workflow that has 16 nodes and 28 edges.

4.3.3. 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 4.1. The total computation cost of a workflow W_i on provider p is defined in Equation 4.1. Data costs consist of data transfer, storage, and caching costs.

$$C_{W_i,p} = \sum_{j \in W_i} U_j \cdot X_j \cdot F_p^{\text{cpu}} \tag{4.1}$$

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.

Table 4.1. Main symbols used in the cost function given in Equation 4.1 and Figures 4.5-4.8.

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
I_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

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

Figure 4.5. Cost calculation in the *submitJob* function.

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 Figure 4.5. The cost calculation for computation is implemented in lines 2-4, while the cost calculation for data is implemented in lines 5-20.

4.3.4. 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.

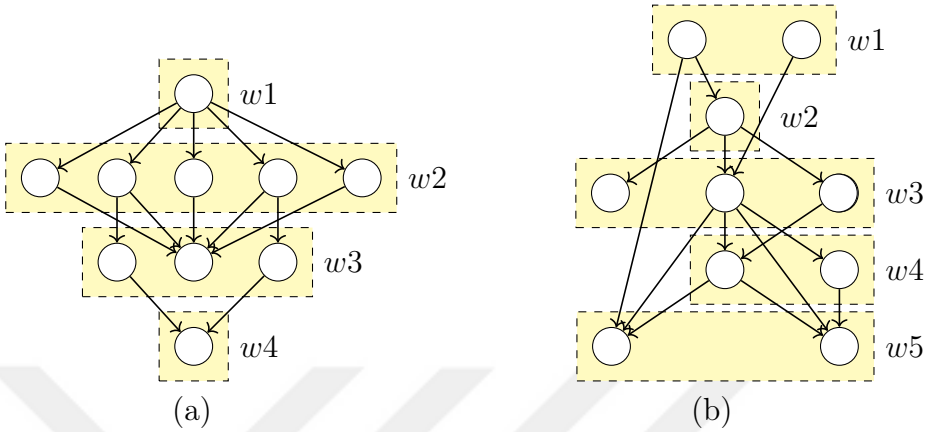


Figure 4.6. Example view of topological generations layering on randomly generated DAGs.

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 4.6 illustrates examples of partitioned workflows using the topological generations layering.

4.3.5. Workflow Engine Algorithms

For the tests, the workflow engine algorithms illustrated in Figures 4.7 and 4.8 were used to submit scheduled jobs. During the workflow submission, the state of all the submitted jobs is checked every 30 seconds and updated based on the state flow in Figure 4.9. Our algorithm distributes new sub-workflows to providers when state changes occur.

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

1: Generate a list of jobs to submit to each provider set ($p_k \in P$) using the HEFT algorithm.
2: Compute a sub-workflow ω_i, traversing graph starting from the sink node till there is no dependency from jobs in other providers.
3: Submit sub-workflow ω_i to its corresponding provider p_k.
4: **while** there are incomplete jobs in J **do**
5: Update the status of all the ongoing jobs
6: **if** there is a newly completed job in the list **then**
7: **for** each generated list of remaining jobs to submit to each provider p_k in the provider-set ($p_k \in P$) **do**
8: Traverse graph starting list of incomplete jobs till there is no dependency from jobs in other providers.
9: Compute sub-workflow ω_i to submit
10: Submit sub-workflow batch ω_i to its corresponding provider p_k.
11: **end for**
12: **end if**
13: **end while**

Figure 4.7. The workflow engine algorithm to distribute jobs using the HEFT [40, 78] algorithm.

1: Generate a list of layers of jobs to submit to each provider set ($p_k \in P$) using the TGL algorithm.
2: **for** each generated list in each layer that will be submitted to each provider p_k in the provider-set ($p_k \in P$) **do**
3: **while** layer's all jobs' dependent jobs are incomplete **do**
4: Update the status of all the ongoing jobs.
5: **end while**
6: Partition workload sub-workflow ω_i into number of available providers in $\bar{P}$ and submit them to its corresponding provider p_k.
7: **end for**

Figure 4.8. The workflow engine algorithm to distribute jobs using the topological generations [75] algorithm.

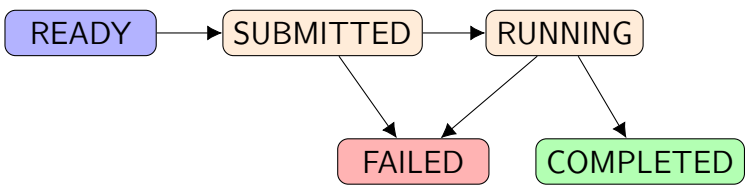


Figure 4.9. State changes of a submitted job.

4.3.6. Caching

In our model, IPFS is utilized for data transfer between providers as well as between requesters and providers. 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 repository. For that reason, providers maintain the corresponding data files to run the workflow and each job's generated outputs in their IPFS repository until the workflow's execution has been completed. For data reusability, output data files may also be stored longer on the provider's IPFS repository 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 rather than 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.

4.4. Experimental Results and Discussion

Before testing in real environments, the eBlocBroker smart contract was developed and tested using Brownie, and the contracts were implemented using Solidity version 0.8.20 and Ethereum Virtual Machine in Istanbul version. 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 [78] and TGL is implemented in [75].

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 4.2. In addition, in Table 4.3, 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 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 Table 4.2 and 4.3, 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 [79].
- 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 minutes.

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.

Table 4.2. Results of the HEFT and TGL algorithms.

Test	**Workflow** $\lvert V\rvert$	$\lvert E\rvert$	**Method**	**Completion Time (min)**	**Gas Used for *submitJob* (gas)**	**Gas Used for *processPayment* (gas)**	**Actual/Complete Cost (cents)**	**Failed**
T_1	16	28	HEFT	27	1722983	1942770	0.827/0.827	0
T_2	32	56	HEFT	44	3738070	3986219	2.037/2.037	0
T_3	64	112	HEFT	52	6745934	8037017	3.234/3.234	0
T_4	128	224	HEFT	100	13858812	16524333	6.867/7.025	2
T_5	256	448	HEFT	191.5	27859947	33574732	13.909/14.060	3
T_6	16	28	TGL	48.5	3533072	1932928	1.768/1.768	0
T_7	32	56	TGL	64	5709622	3965208	3.169/3.169	0
T_8	64	112	TGL	96	10192826	8017429	5.410/5.410	0
T_9	128	224	TGL	222	21100398	15922550	12.243/12.243	0
T_{10}	256	448	TGL	266	29775297	32933948	19.511/19.528	1

Table 4.3. Results of the HEFT and TGL algorithms taking provider loads into account.

Test	**Workflow** $\lvert V\rvert$	$\lvert E\rvert$	**Method**	**Completion Time (min)**	**Gas Used for *submitJob* (gas)**	**Gas Used for *processPayment* (gas)**	**Actual/Complete Cost (cents)**	**Failed**
T_{11}	128	224	HEFT	102	13591822	16278401	6.972/7.009	1
T_{12}	128	224	HEFT	111	13072131	16125526	6.961/7.025	1
T_{13}	32	56	TGL	59.5	5777102	4089870	3.167/3.167	0
T_{14}	32	56	TGL	58.3	5762126	4082402	3.170/3.170	0

In order to simulate the execution of workloads, the Slurm emulation mode is used to create a cluster, and the sleep programs are served 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 4.4. Note that once the workflow is submitted through eBlocBroker, the cost will remain unchanged until the workflow is finished, regardless of any price fluctuation.

Table 4.4. Description of prices on provider p.

provider	F_p^{cpu} **(cents)**	$F_p^{storage}$ **(cents)**	F_p^{cache} **(cents)**	F_p^{trans} **(cents)**
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

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 Table 4.2 and 4.3, 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. The completion time of each workflow

is recorded in minutes. Since we ran Slurm in emulation mode, the run-time of jobs is simulation times. Across all given node values, TGL has a longer completion time than HEFT. There may be a single job within multiple layers that can slows down the completion of workflow.

Furthermore, the TGL algorithm is more aggressively distributing jobs to each provider. Due to the fact that it evenly distributes job sets among providers, there is an extra cost for data transfer. The HEFT algorithm distributes jobs to keep data transfer costs at a minimum. Consequently, the TGL algorithm is not an effective choice for workflow partitioning.

4.5. Discussion

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

The list below highlights the features included in our future roadmap:

(i) Larger clusters will be used for testing workflow submissions.

(ii) A system for feedback reviews could be implemented for computation providers, just like the one used by 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.

(iii) The RISC Zero [80] 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.

5. DESIGN OF A SMART CONTRACT BASED AUTONOMOUS ORGANIZATION FOR SUSTAINABLE SOFTWARE AND E-SCIENCE

5.1. Introduction

In the field of e-Science, the development of open-source software like OpenStreetMap, Linux kernel, TensorFlow, and Blender usually takes place within the duration of a funded project. In general, the software is uploaded to a public code repository, with selected developers are financially supported throughout the project timeline to maintain the software. Once a project is completed, and if the project team is not given a new project, the responsibility for maintaining and advancing the software falls on volunteer developers. This method has resulted in the creation of successful developer communities for various software. On the other hand, there is a potential risk that follow-up communities may not be formed; ultimately, all the work into development may be wasted because no one is left to maintain and further develop the software. The primary focus of this chapter is addressing this issue, i.e., can we develop an unstoppable virtual organization that will: (i) represent a open-source software, (ii) keep execution records of software by computing service providers, and provide transactional data for analysis and reproducibility checks, (iii) continue to exist even after the related projects have ended, (iv) offer developers decision-making, crowdfunding, and citation mechanisms.

Before the rise of blockchain and smart contracts, a common method for ensuring software sustainability was creating a company and generating revenue through software sales or support, thereby securing funds for ongoing development efforts. Al-

though effective, this method demands significant financial investment and bureaucratic procedures. It can only truly succeed if there is a large demand for the software in the market. Despite this, there may be better approaches than establishing a company when it comes to open-source and free software developed by communities. A virtual organization operating through a smart contract could be a feasible alternative.

DAOs can coordinate the work of crowds of users, developers, and researchers by utilizing smart contracts on blockchains. This chapter proposes utilizing blockchain and smart contract-based solutions for improving software sustainability and execution traceability:

(i) *Development and community:* A distributed autonomous software organization model and its Ethereum smart contract implementation provide a funding mechanism and decision-making process based on voting as an unstoppable virtual software organization for the software community.

(ii) *Professionalization:* Since blockchains are open public data, citations of software by others, records of funding, membership in software organizations, and usages are stored in a single location (blockchain) and updated continuously. Blockchain can be monitored by funding agencies and employers, who can utilize this information to offer incentives, promotions, and grants. The use of blockchain data in this approach results in higher quality sustainable software since the efforts of developers are better assessed quantitatively.

(iii) *Credit:* The blockchain not only allows for crypto-currency donations but also enables the recording of citations from paper authors and software usage by other software. Hence, a new credit and citation ecosystem can be developed using blockchain technology.

(iv) *Software publishing:* Anyone can utilize blockchains to record and announce software-related activities, such as forming autonomous organizations, announcing new versions, funding records, and citations as events. Developing standardized smart contracts for software and interfaces can aid in software discoverability and reuse.

(v) *Software reproducibility:* Software execution records can be stored on the blockchain at the request of conferences, journals, and funding agencies. The blockchain has the ability to store hashes of both inputs and outputs. Due to its immutable nature and notary capabilities, blockchain allows software executors to guarantee that a specific input will consistently yield a corresponding output.

The importance of software reproducibility is becoming more evident as artificial intelligence (AI) projects advance. In April 2021, the European Commission proposed an EU regulatory structure on AI. The draft AI act [81] is the first attempt to enact horizontal AI regulations. In this stage, the data files utilized for training an AI model and the generated data hold significant importance.

This chapter is an extension of our conference paper [17]. Our primary contributions are as follows:

(i) Design and implement a smart contract called AutonomousSoftwareOrg, which is a DAO-based smart contract for open-source software development, sustainable and reproducible software, and advancements in e-Science.
(ii) Implementation of the reproducibility component of software executions, which is essential for keeping software sustainable [17]. Each software execution trace is recorded on the blockchain, showcasing the connection between input and output data files for each execution. The hash of the generated outputs from jobs can be recorded to facilitate reproducibility checks for organizations.
(iii) Recording of software execution traces on the blockchain. Each trace consists of a linkage of input and output data files of each software execution.
(iv) An AND/OR graph-based model for software executions and input/output data files has been developed, along with software tools for the analysis of these executions.

Note that blockchain-based computation service platforms like Ethernity Cloud, Golem, iExec, and Fluence do not provide the above features (i-iii).

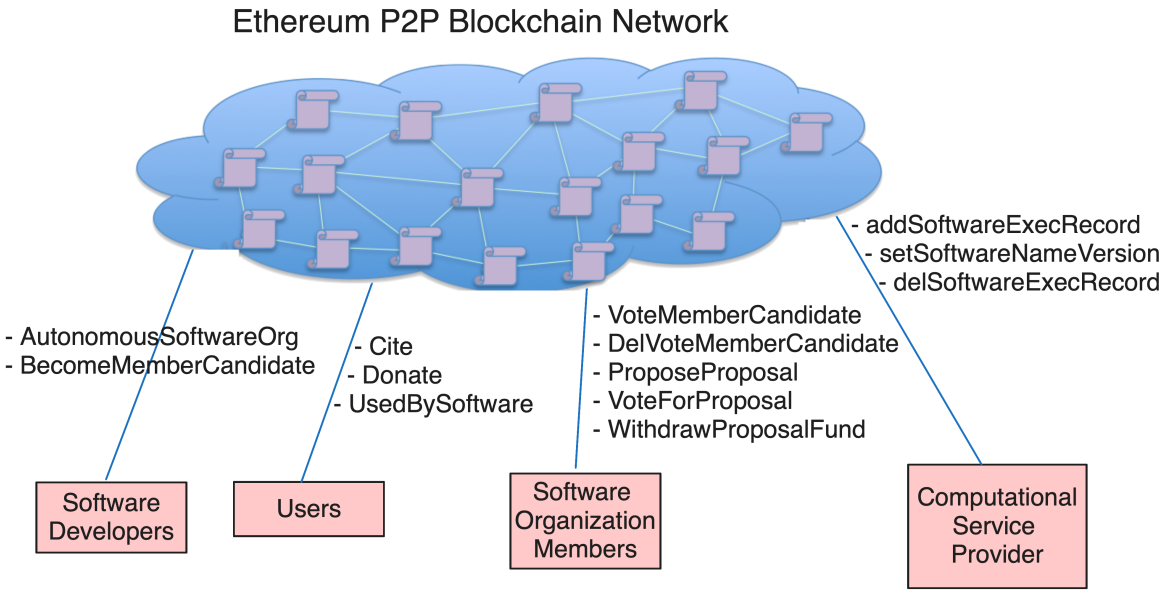


Figure 5.1. Blockchain-based autonomous organization model.

In Section 5.2, the blockchain-based autonomous software organization model is covered. In Section 5.3, details of the smart contract that implements the software organization model are given. In Section 5.4, demonstration of the smart contract on an Ethereum-based private blockchain network (Bloxberg) is presented. In Section 5.5, we first define the data and software execution graph and then describe the procedures used to analyze this graph. In Section 5.6, we conclude the chapter.

5.2. Blockchain-Based Autonomous Organization Model

The distributed autonomous software organization that has been designed and implemented in this chapter aims to provide a virtual organization for the software developers, enables them to collect funding for the software development, and makes decisions through a voting process. In the virtual organization's initial setup, decisions must be made by at least the M/N majority rule. This means that a minimum of M/N members must vote for a decision to be approved. The design of our autonomous organization is illustrated in Figure 5.1. Our system is comprised of eight core components:

(i) *Users* use the software developed by the AutonomousSoftwareOrg. Users who like the software can donate to the organization by invoking the *Donate()* function. Companies also have the option to use the *Donate()* function to offer financial grants to software developers. Researchers who publish in journals or conferences can cite the software using the *Cite()* function, including the DOI number of their publications. Suppose the user is a developer for a different software, which is controlled by another smart contract. In that case, they can use the *UsedBySoftware()* function to register the smart contract address for the other software. The *UsedBySoftware()* function links two software using smart contracts while the *Cite()* function links a publication and the software using a smart contract. The *setSoftwareNameVersion()* function can be used to map different software versions to the software name. The software name is passed to this function. Alternatively, the smart contract address for software can be passed as the name. Computational service providers who execute users' jobs can also use the *addSoftwareExecRecord()* function to notarize and record hashes of input and output files used by the software in the users' jobs, which can be useful for reproducibility checks.

(ii) The *Computational Service provider* offers computational and storage resources, as well as datasets.

(iii) *Software Developers* desire to contribute to software development efforts within an organization. The initial creation (deployment) of the AutonomousSoftwareOrg smart contract is done by a sole developer who invokes the constructor function of the AutonomousSoftwareOrg smart contract. The organization's first member is designated as the person in charge of deploying the contract. The process to become a member candidate involves calling the *BecomeMemberCandidate()* function and providing personal details. Membership is determined through a voting process conducted by current members.

(iv) *Members of the Software Organization* are the software developers who are formally associated with the software organization and have voting rights. They can propose development proposals and ask for funding from the organization by calling the *ProposeProposal()* function. If there are software developers who

wish to vote for candidates to become members, they can call the *VoteCandidateMember()* function. They can also call the *DelVoteCandidateMember()* function to delete their vote for a candidate or a member. Once the voting majority is achieved, a candidate is granted membership. If a member fails to secure a voting majority, he or she will lose their membership. Members vote in order to obtain financial support for software development proposals. They cast their votes by using the *VoteForProposal()* function. Once a voting majority is achieved for a proposal, the member who proposed it has the option to withdraw funding by calling the *WithdrawProposalFund()* function.

(v) The *Ethereum blockchain infrastructure*, as Bloxberg, runs on a P2P network and executes smart contracts. Anyone can utilize the Bloxberg blockchain by accessing their RPC API.

(vi) *AutonomousSoftwareOrg Smart Contract* can be found on the Bloxberg blockchain network and operates autonomously when its functions are invoked. It is implemented as a Solidity programming language and contains the operation rules of the software organization. It has the rules for proposal funding and majority voting decisions for membership. It also provides getter and event functions for the number of votes, proposals, and members.

(vii) *eBlocBroker infrastructure* records the hash of each submitted source code and data files, along with their generated output's hashes into the eBlocBroker smart contract. Hence, it maintains a trace of data and software usage. Figure 5.2 illustrates the interactions between the components of the eBlocBroker infrastructure. Figure 5.3 illustrates a general AND/OR graph of software executions, where datasets are used among various software.

(viii) *Decentralized Storage* using IPFS allows users to publicly share data. Decentralized storage using IPFS is where the users' source code is read, and the provider writes the changes made by the completed jobs.

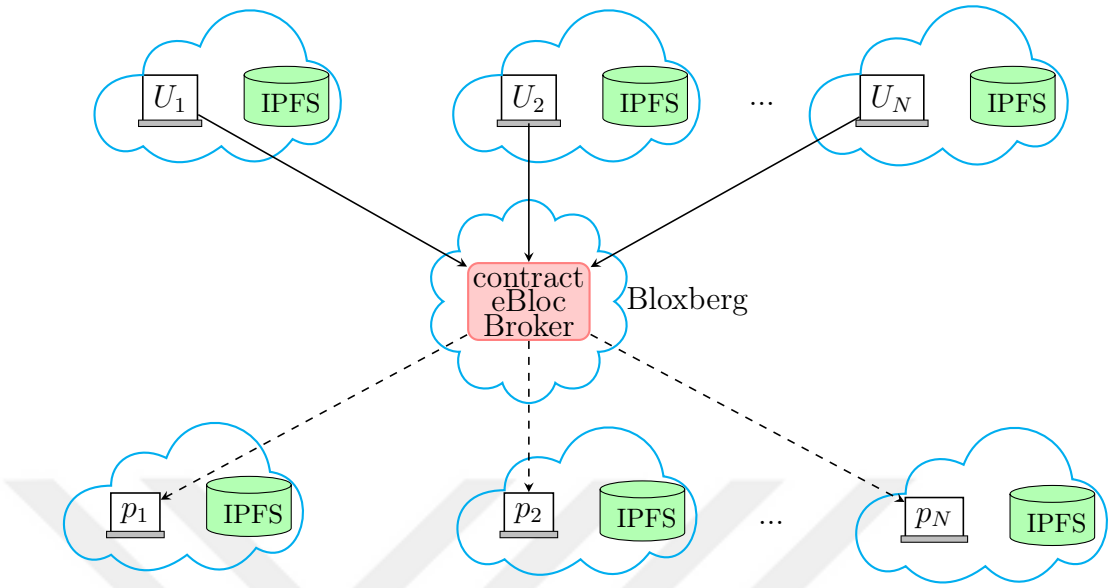


Figure 5.2. Interactions between the users and providers through eBlocBroker smart contract. Note that U stands for requester and p stands for provider.

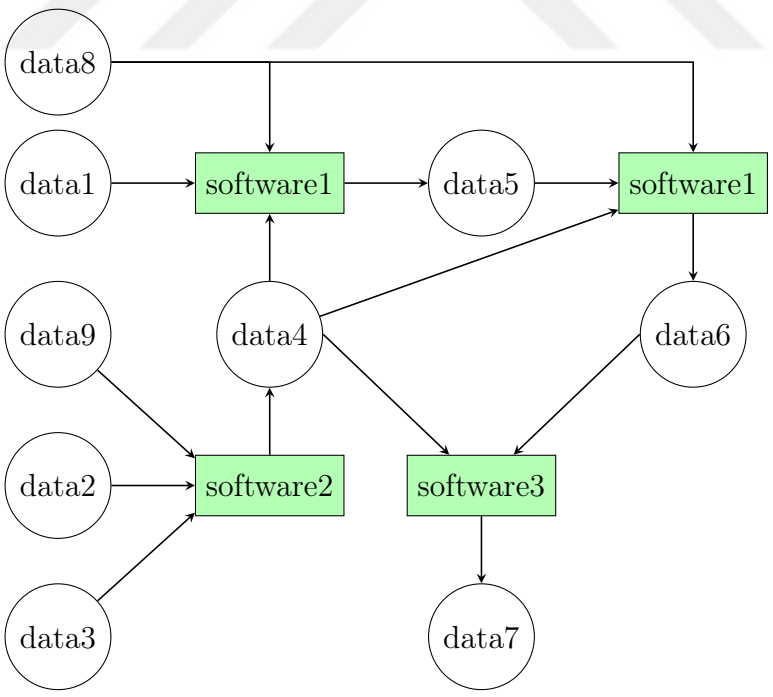


Figure 5.3. Example of a software execution AND/OR graph.

```
contract AutonomousSoftwareOrg {
  ... // Variable definitions
  ... // Conditional modifier definitions
  ... // Event definitions
  function constructor(string memory name, uint8 m, uint8 n, string memory url,
    address _eBlocBrokerAddress, address _ResearchCertificateAddress)
  function ProposeProposal(string memory title, string memory url, bytes32
    propHash, uint requestedFund, uint deadline) public member(msg.sender)
    validDeadline(deadline)
  function VoteForProposal(uint propNo) public validProposalNo(propNo)
    withinDeadline(propNo) member(msg.sender) notVotedForProposal(propNo)
  function WithdrawProposalFund(uint propNo) public validProposalNo(propNo)
    withinDeadline(propNo) member(msg.sender) enoughFundBalance(propNo)
    proposalOwner(propNo) proposalMajority(propNo)
  function BecomeMemberCandidate(string memory url) public notMember(msg.sender)
  function VoteMemberCandidate(uint memberNo) public validMemberNo(memberNo)
    member(msg.sender) notVotedForMember(memberNo)
  function DelVoteMemberCandidate(uint memberNo) public validMemberNo(memberNo)
    member(msg.sender) votedForMember(memberNo)
  function Donate() payable public nonZeroPaymentMade
  function Cite(bytes32 doiNumber) public
  function UsedBySoftware(address addr) public
  function setNextSoftwareExecutionRecordCounter(bytes32 sourceCodeHash) public
    member(msg.sender) validEblocBrokerProvider() returns (uint32)
  function addSoftwareExecRecord(bytes32 sourceCodeHash, uint32 index, bytes32[]
    memory inputHash, bytes32[] memory outputHash) member(msg.sender)
    validEblocBrokerProvider() returns (uint32)
  function delSoftwareExecRecord(bytes32 sourceCodeHash, uint32 index)
    member(msg.sender) public member(msg.sender) softwareOwnerCheck(index)
  function setSoftwareNameVersion(bytes32 sourceCodeHash, string memory name,
    string memory version) public member(msg.sender) validEblocBrokerProvider()
  function getSoftwareExecutionCounter() public view returns(uint32)
  function getAutonomousSoftwareOrgInfo() public returns (string memory, uint,
    uint, uint, uint)
  function getMemberInfoLength() public view returns (uint)
  function getMemberInfo(uint memberNo) member(membersInfo[memberNo-1].memberAddr)
    public view returns (string memory, address, uint)
  function getCandidateMemberInfo(uint memberNo)
    notMember(membersInfo[memberNo-1].memberAddr) public view returns (string
    memory, address, uint)
  function getProposalsLength() public view returns (uint)
  function getProposal(uint propNo) public view returns (string memory, string
    memory, uint256, uint, uint, bool, uint)
  function getDonationInfo(uint donationNo) public view returns (address, uint,
    uint)
  function getCitation(uint citeno) public view returns (bytes32)
  function getUsedBySoftware(uint usedBySoftwareNo) public view returns (address)
  function getNoOfIncomingDataArcs(bytes32 sourceCodeHash, uint32 index) public
    view returns(uint)
  function getNoOfOutgoingDataArcs(bytes32 sourceCodeHash, uint32 index) public
    view returns(uint)
  function getIncomingData(bytes32 sourceCodeHash, uint32 index, uint i) public
    view returns(uint)
  function getOutgoingData(bytes32 sourceCodeHash, uint32 index, uint i) public
    view returns(uint)
  ... // the remaining getter functions
}
```

Figure 5.4. The AutonomousSoftwareOrg smart contract function signatures.

5.3. Smart Contract Details

The programming language Solidity provides conditional modifiers that can be used in function bodies. We use the Condition Oriented Programming (COP) style proposed by [82]. When the conditional modifiers are true, the function bodies are executed. The full source code of the AutonomousSoftwareOrg smart contract and graph tools are available at `https://github.com/ebloc/AutonomousSoftwareOrg`. Figure 5.4 illustrates the signatures of the functions in the AutonomousSoftwareOrg smart contract, which also defines and emits events when transactions such as proposal submissions or voting are performed, allowing anyone to watch these events. Getter functions, which return the values of various variables, are also provided and are defined as *public view* functions. Note that the descriptions of conditional modifiers of the AutonomousSoftwareOrg smart contract are given in Table 5.1.

5.3.1. Security Model

5.3.1.1. Unbounded Loops Vulnerability. Unbounded loops in Solidity can cause denial-of-service attacks and run out of gas issues. For that reason, we have only used loops with defined endpoint in the *addSoftwareExecRecord()* function.

5.3.1.2. Oracle Manipulation. Protocols sometimes need additional information from outside the blockchain to function properly. This off-chain information is supplied by oracles which in our case are computational providers. They will write hashes of both the inputs and outputs of software executions into the AutonomousSoftwareOrg smart contract by calling the *addSoftwareExecRecord()* function. The check to verify if a valid provider performs this call is made by the *validEblocBrokerProvider()* modifier. Note that eBlocBroker is built on the assumption that all providers are trusted. Users should select providers based on their website and performance evaluations.

Table 5.1. Conditional modifiers in the AutonomousSoftwareOrg smart contract.

Modifier	**Condition**
enoughFundBalance(propNo)	there is sufficient balance to pay for proposal *propNo* funding
validProposalNo(propNo)	*propNo* is a valid proposal number
member(addr)	*addr* is the Ethereum blockchain address of a member
notMember(addr)	*addr* is not the Ethereum blockchain address of a member
validMemberNo(memberNo)	*memberNo* is a valid member/candidate number
validDeadline(deadline)	*deadline* is a valid block number
withinDeadline(propNo)	deadline for proposal *propNo* has not passed
notVotedForProposal(propNo)	function invoker did not vote for proposal *propNo*
notVotedForMember(memberNo)	function invoker did not vote for member with *memberNo*
votedForMember(memberNo)	function invoker voted for member with *memberNo*
proposalOwner(propNo)	function invoker is the owner of proposal *propNo*
proposalMajority(propNo)	at least M/N of the members voted affirmatively for the proposal *propNo*
membershipMajority(memberNo)	at least M/N of the members voted affirmatively about membership of *memberNo*
nonZeroPaymentMade()	non-zero payment is also sent to the contract when the function is invoked
validEblocBrokerProvider()	*msg.sender* is a valid provider at eBlocBroker smart contract
softwareOwnerCheck(index)	*msg.sender* matches with the address on the given *index*

5.3.1.3. Withdraw Funds for each Proposal. The *WithdrawProposalFund()* function in Figure 5.5 follows the *Checks-Effects-Interactions* pattern, where the recipient is responsible for claiming their funds by initiating a withdrawal transaction. The conditional modifiers will initially perform some checks, which should be done first. As the second step, if all checks pass, changes to the state variables of the current contract should be made. In line 5, the fund balance is set to zero before transferring the funds to *msg.sender* to prevent reentrancy attacks.

```
1 function WithdrawProposalFund(uint propNo) public proposalOwner(propNo)
    proposalMajority(propNo)
2     validProposalNo(propNo) withinDeadline(propNo) member(msg.sender)
        enough_fund_balance(propNo) {
3     uint fund = proposals[propNo].requestedFund;
4     weiBalance -= fund;
5     proposals[propNo].requestedFund = 0;
6     (bool success, ) = msg.sender.call{value: fund}("");
7     require(success, "Transfer failed.");
8     emit LogWithdrawProposalFund(propNo, fund, block.number, msg.sender);
9 }
```

Figure 5.5. Withdraw fund of a proposal using the *Checks Effects Interactions* pattern.

5.3.2. The Research-Certificate smart contract

For testing purposes we copied and modified the Bloxberg ResearchCertificate smart contract [83], and added new functions. Our ResearchCertificate smart contract can link every version of software or data hash to a unique token ID, which later proves that this specific data hash was certified earlier. Afterward, a unique token ID for each software's source code hash or data file hash could be used to identify them. Figure 5.6 shows the signatures of the functions in the ResearchCertificate smart contract. Note that the ResearchCertificate smart contract inherits from ERC721, which is a Non-Fungible Token (NFT) that is used to uniquely identify an entity.

```
contract ResearchCertificate is ERC721 {
  ... // Variable definitions
  function createCertificate(address recipient, bytes32 dataHash) public returns
    (uint256)
  function getTokenIndex(bytes32 dataHash) public view returns(uint256)
  function getDataHash(uint256 index) public view returns(bytes32)
  function getDataHashLen() public view returns(uint256)
}
```

Figure 5.6. ResearchCertificate [83] smart contract function signatures.

5.4. Tests

The contract was implemented using Solidity version 0.8.22 and Ethereum Virtual Machine in Istanbul version. Brownie was used for developing and testing the smart contract of AutonomousSoftwareOrg smart contract. In order to test it in a real

environment, it was deployed, tested, and debugged on Bloxberg.

The scenario we are testing is the following: One developer deployed the smart contract and registered ten candidate members into the AutonomousSoftwareOrg smart contract using the *BecomeMemberCandidate()* function. Then, the contract deployer uses the *VoteMemberCandidate()* function to vote for one randomly selected new candidate who becomes a valid member. Afterward, valid members repeat this process for candidate members until all candidates become valid members. Numerous users use the *Donate()* function to donate Ether to the AutonomousSoftwareOrg smart contract. A randomly selected member proposes a proposal using the *ProposeProposal()* function, requesting 10 Ethers for funding and setting a deadline. Members use the *VoteForProposal()* function to vote for the proposal until the required majority is reached. Voting for proposals can be watched using the *LogProposalVote()* event function. After the deadline passes the latest block number, the proposal proposer withdraws his requested fund using the *WithdrawProposalFund()* function.

5.4.0.1. Gas Consumption. The gas usage for the AutonomousSoftwareOrg smart contract deployment was 2,878,669 gas units. The Table 5.2 provides the amount of gas used for executing each smart contract function, where the estimated gas costs are calculated from the Ganache platform. As an alternative, Polygon is a layer 2 scaling solution created to offer Ethereum developers more features and lower fees, making real costs much cheaper. The AutonomousSoftwareOrg smart contract can also be deployed on Ethereum networks such as Polygon. AutonomousSoftwareOrg is currently deployed and operational on the Bloxberg network. As of February 18th, 2024

- The average gas price is 22.60 Gwei, and the price of 1 Ether (ETH) is 2826 USD [70].
- The average gas price is 88.10 Gwei, and the price of 1 Polygon (MATIC) is 0.97 USD [71].

We multiplied the gas cost to calculate the USD price of a gas fee, given the Ethereum network's gas price and price in dollars. The average gas consumption of eBlocBroker's functions, along with their USD price in Ethereum Mainnet and Polygon, are given in Table 5.2.

Note that under the hood, the *addSoftwareExecRecord* function calls the ResearchCertificate's *createCertificate* function for each given data hash, which causes additional gas usage equal to around 116000 gas. Hence, the gas cost of *addSoftwareExecRecord()* will increases based on the array size of the input and output hashes, which will be multiplied by 116000 gas and added to the *addSoftwareExecRecord()* based gas consumption, which is around 178444.

Table 5.2. Average gas cost of the AutonomousSoftwareOrg's core functions.

Method	**Gas Used (gas)**	**Ethereum USD cost**	**Polygon USD cost**
ProposeProposal	156486	9.33	0.012
VoteForProposal	73712	4.71	0.0063
WithdrawProposalFund	62601	4.00	0.0053
BecomeMemberCandidate	74384	4.75	0.0064
VoteMemberCandidate	107250	6.85	0.0092
DelVoteMemberCandidate	29412	1.88	0.0025
Donate	125700	8.03	0.0107
Cite	62643	4.00	0.0054
UsedBySoftware	63438	4.05	0.0054
setSoftwareNameVersion	73952	4.72	0.0063
setNextExecutionCounter	75257	4.81	0.0064
addSoftwareExecRecord	178444	11.40	0.0152

5.5. Analyzing Software Executions and Data Usages

We model software executions and their input data usages and output data productions with AND/OR graphs. An AND/OR graph is a directed bipartite graph, $G(V_o, V_a, A)$, with two disjoint sets of vertices, V_o the set of OR nodes and V_a the set of AND nodes. The arc set A is given by $A = \{< o, a >: o \in V_o, a \in V_o \; and \; o \in incoming(a)\} \bigcup \{< a, o >: a \in V_a, o \in V_o \; and \; o \in outcoming(a)\}$. In our model, the software execution graph, $G(D, S^{\text{exec}}, E)$, is formed as an AND/OR graph with V_o being

input/output data files and V_a being the software execution transactions and the arcs, A, representing the incoming/outcoming input/output files from software executions. This software execution AND/OR graph can be downloaded and analyzed in order to gain insight into the importance and criticality of software and data files among all users and developers. In the following subsection, our first step involves defining the data and software execution graph, followed by a description of the procedures employed to analyze it.

Figure 5.7 illustrates a general example of a trace of the software executions, where the hash of the data file and software with a specific version is stored in the same way as the unique token ID; hence, the software's token ID is followed by the execution index after the dot.

5.5.1. Data and Software Execution Graph

The list of symbols, their meanings, and definitions for defining data and software execution graph is represented in Table 5.3.

The hierarchy of a software is depicted in Figure 5.8, where each software can have various versions with different execution indexes. Note that the software version is stored in the AutonomousSoftwareOrg smart contract using a mapping from hash, as key, to version, as value. In order to store the name of the software, Solidity's two-dimensional mapping, where hash and version are keys, is used. Different versions indicate the software has a different source code hash since the source code differs.

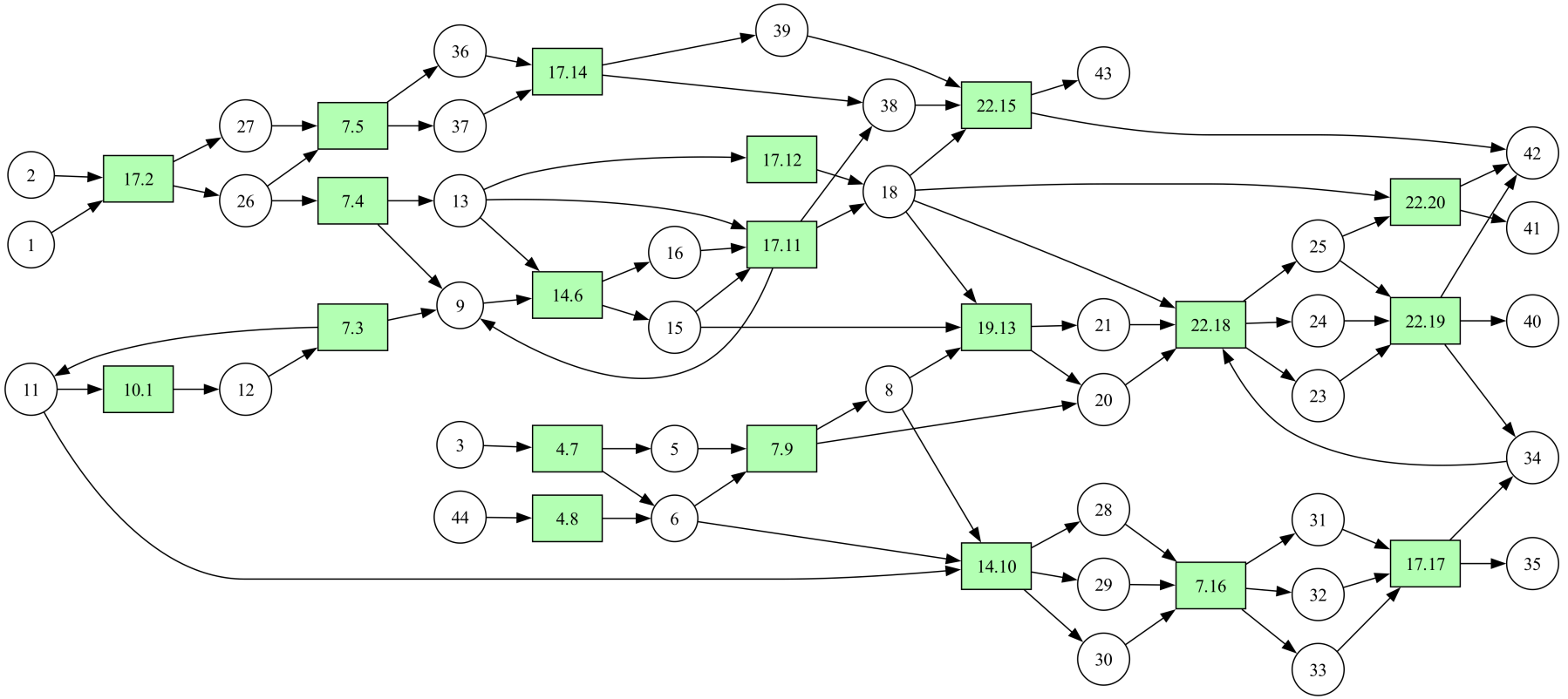
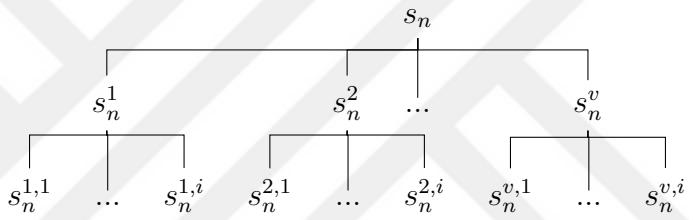


Figure 5.7. Example of a general AND/OR graph of software executions. Rectangle nodes represent software executions, and circle nodes represent input/output data files.

Table 5.3. Main symbols used defining data and software execution graph.

Symbol	Meaning
D	Set of data files: $D = \{d_1, d_2, \ldots, d_{\|D\|}\}$
S	Set of software: $S = \{s_1, s_2, \ldots, s_{\|S\|}\}$
s_n^v	Software s_n with version v
$s_n^{v,i}$	Software s_n with version v and execution index i
S^{exec}	Set of software executions
E	Set of edges: $E = \{e_1, e_2, \ldots, e_{\|E\|}\}$
$G(D, S^{\text{exec}}, E)$	AND/OR graph with D as the set of data files, S^e as the set of software executions and E as the set of edges representing data inputs and outputs

Figure 5.8. Example of a software hierarchy.



5.5.2. Analysis Algorithms

The list of the graph of software execution analysis algorithms used and developed and their descriptions are given in Table 5.4.

Table 5.4. Graph of Software Execution Analysis Algorithms.

Algorithm	Description
PageRankSE	Pagerank on software version execution graph
PageRankSV	Pagerank on software version graph
PageRankS	Pagerank on software graph
DAGify	Construct a directed acyclic graph (DAG) that shows which software initially generated the data files
KnockedDown	Graph formed by deleting a single file node and all the nodes that transitively depend on it
MaxKnockedDown	Output nodes that produces the maximum number of the knocked down nodes
SWExecMinInput	Software execution that required data files of minimum total size

5.5.2.1. PageRank. We examine the PageRank algorithm on the given example graphs. Table 5.5 illustrates the PageRank of software nodes based on order in Figures 5.7 and 5.9.

Table 5.5. Analyzing PageRank on example graphs, where software nodes are ordered based on perceived importance.

Algorithm	**Results**
PageRankSV	22.18, 22.19, 22.15, 17.17, 7.16, 17.11, 14.6, 17.14, 19.13, 14.10, 22.20, 7.9, 7.5, 7.3, 17.2, 10.1, 7.4, 4.8, 4.7, 17.12
PageRankS	22, 17, 7, 14, 19, 4, 10

5.5.2.2. DAGify. Different software executions can produce identical output data files. For example, the same algorithm implemented in different programming languages could generate the same output. Our goal is to determine which software initially generated the data file.

Table 5.6. List of software in execution order and their initial generated data files.

Exec Order	$s_n^{v,e}$	**Files created**
1	10.1	12
2	17.2	26, 27
3	7.3	9, 11
4	7.4	13
5	7.5	36, 37
6	14.6	15, 16
7	4.7	5, 6
8	4.8	none
9	7.9	8, 20
10	14.10	28, 29, 30
11	17.11	18, 38
12	17.12	none
13	19.13	21
14	17.14	39
15	22.15	42, 43
16	7.16	31, 32, 33
17	17.17	34, 35
18	22.18	23, 24, 25
19	22.19	40
20	22.20	41

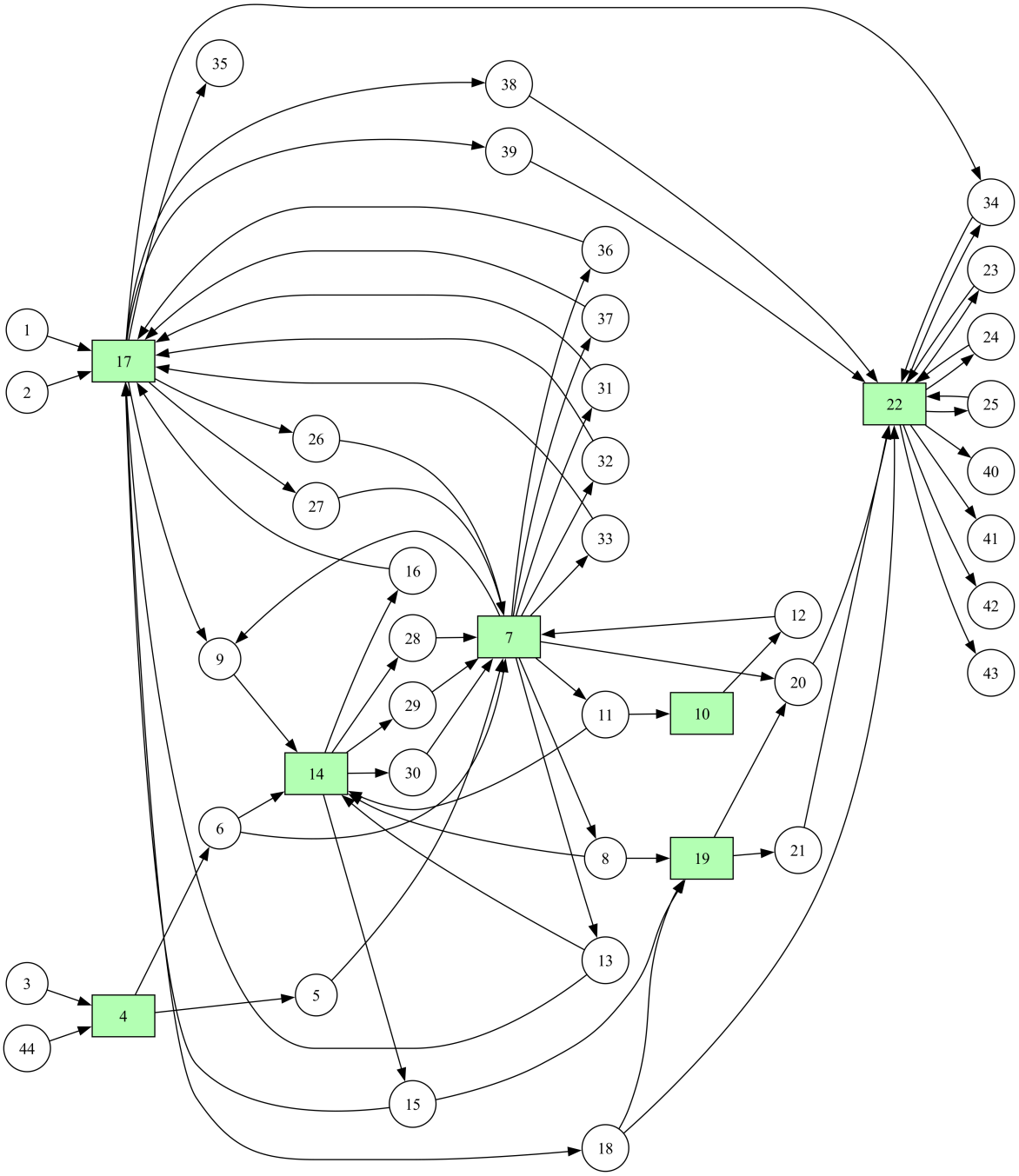


Figure 5.9. Graph of merged software version executions.

The global execution counter assigned to each software can determine the execution order, as it follows a sequential pattern. The original graph shown in Figure 5.7 can be reconstructed by following the software's order of execution. Throughout this procedure, we maintain a record of the software that first created each data file. For example, Table 5.6 illustrates the order of the software executions and their initially

generated data files.

5.5.2.3. Merging same software. We merged executions of the same software, which has the same source code hash, as illustrated in Figure 5.9.

5.5.2.4. MaxKnockedDown. In order to calculate the maximum number of knocked down nodes due to the removal of a single data file, we loop through all data files and evaluate their impact on the graph. Removing a single node affects its out-dependency nodes, where we keep them in a queue. We iterate through nodes inside the queue and add their out-dependency nodes into the queue. The iterated nodes will be removed from the queue, and this process continues until no nodes are left in the queue. In this process, we maintain a record of the nodes that have been knocked down.

As an example, in the main graph, if data node 11 is knocked down, then it directly affects the creation of 36 nodes. Figure 5.10 illustrates the remaining nodes after data node 11 is knocked down.

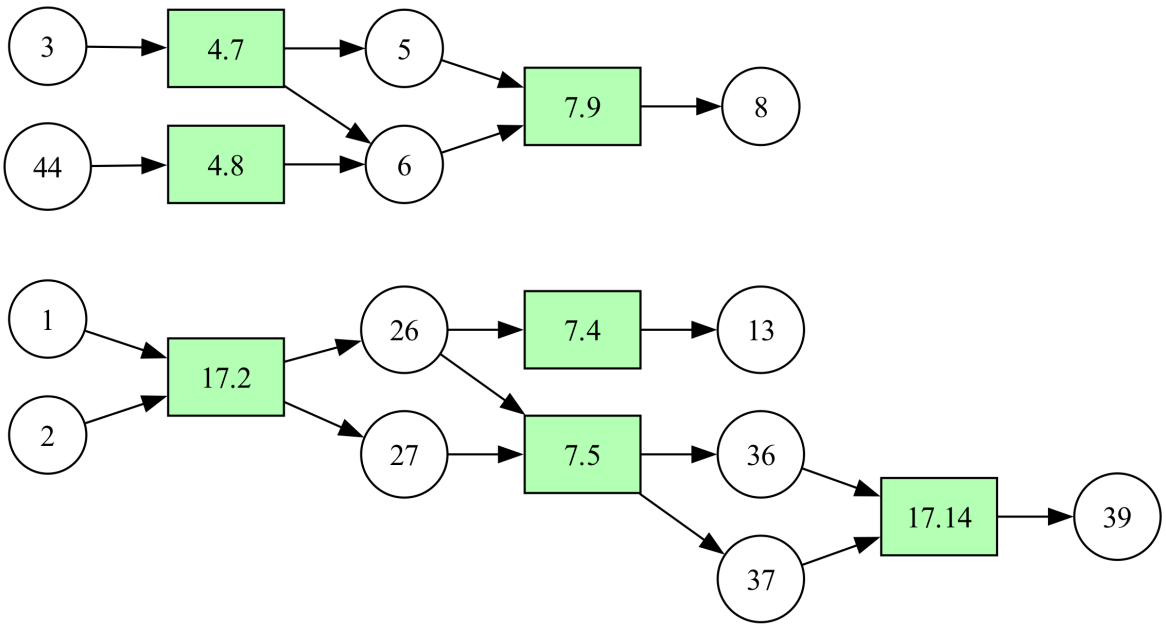


Figure 5.10. Remaining nodes after the maximum number of nodes have been knocked down, with only data node 11 is removed.

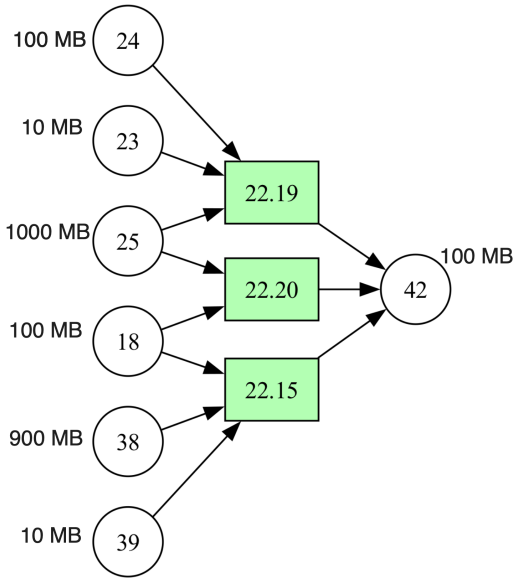


Figure 5.11. Software executions to generate data node 42. The size of each data node is indicated next to it.

5.5.2.5. SWExecMinInput. Here, if a data file is generated by multiple software programs, we determine which software requires the minimum total size of input data files. As illustrated in Figure 5.11, in order to generate node 42, we can take three different approaches where 22.19, 22.20, or 22.15 could generate it. Software execution 22.15 requires data files of minimum total size.

5.6. Discussion

If the AutonomousSoftwareOrg, as proposed in this paper, is deployed on the Ethereum blockchain, the software can transform into a living entity embodied by its unstoppable smart contract. In addition, the AutonomousSoftwareOrg smart contract enables the collection and distribution of funds to both current and future developers who join the autonomous software organization, promoting long-term sustainability. The software has the capability to record usage citations and execution records, which can be utilized to evaluate its impact. The data is notarized and continuously updated, as it is replicated on thousands of blockchain P2P nodes, ensuring widespread availability. The centrally-managed services do not offer all of these capabilities. Furthermore,

it has the ability to monitor software executions and store their corresponding input and output data files, which can then be utilized to create software execution graphs for analysis.

The following list shows the features included in our future roadmap:

(i) An accessible web interface will be created using Web3.js.

(ii) The establishment of software assessment frameworks and seal of approvals is becoming increasingly crucial [84]. Functionality to record the seal of approvals and assessment framework results.

6. CONCLUSION

In this thesis, we present the integration of blockchain technology into scientific workflow management, which represents a paradigm shift in how computational resources are allocated, managed, and utilized in e-Science. This thesis has explored the development and application of the eBlocBroker infrastructure, an autonomous, blockchain-based resource broker designed to enhance scientific workflows' efficiency, transparency, and security. Setting up blockchain-based computational and workflow services involves many different areas of system software, optimization, decentralized autonomous smart contract programming, storage management, and system integration. We have also explored smart contracts to enhance software sustainability in e-Science.

The first part of this study, demonstrates how a blockchain-based autonomous resource broker, eBlocBroker, can be utilized to create a virtual market for computational and data resources within research communities. Our study has shown that eBlocBroker, through its smart contract, offers a decentralized, transparent, and secure platform for managing and sharing computational and storage resources. The experiments carried out confirm the efficiency of eBlocBroker in handling job submissions and executions, showcasing its potential as a groundbreaking tool for researchers. The findings from this research provide a strong foundation for future exploration into the integration of blockchain technology in scientific workflows.

The second part of this study, dives into the detailed technical architecture and mathematical models that form the foundation of the eBlocBroker infrastructure. It highlights the connections among the blockchain, computing providers, and users, demonstrating different scheduling algorithms that optimize workflow execution. Furthermore, it provides an in-depth cost model that considers computation and data costs, along with exploring using IPFS for decentralized data storage to enhance the system's scalability and robustness.

Third, we proposed a distributed autonomous software organization model and its implementation of the Ethereum smart contract, known as AutonomousSoftwareOrg. This model provides a continuously operating virtual organization for software development communities, facilitating a funding mechanism based on cryptocurrencies and a democratic decision-making process through voting. By keeping track of software usage, citations, and executions on the blockchain, AutonomousSoftwareOrg ensures transparency and sustainability, enabling software to persist and endure individual projects' lifespan. One of the key contributions of this research is recording software execution traces on the blockchain. By connecting input and output data files for each software execution, we ensure a transparent and immutable record of all activities. This enhances the reproducibility of scientific results and builds trust within the research community. Furthermore, we developed an AND/OR graph-based model for software executions and the associated input/output data files. This model allows for a comprehensive analysis of software executions, providing valuable insights into the performance and behavior of various software tools. The developed software tools facilitate the analysis process, making it easier to identify patterns, optimize performance, and ensure the reliability of software systems.

The integration of blockchain technology into scientific workflow management in this thesis leads to the development of a blockchain-based alternative workflow execution service for researchers and academicians. The eBlocBroker infrastructure showcases the possibilities of decentralized, transparent, and efficient resource management in e-Science, establishing a new benchmark for future research methods. Through solving current challenges and introducing a scalable framework for complex scientific workflows, this innovative method not only enhances current practices but also opens doors for future developments in the field.

In conclusion, integrating blockchain technology and smart contracts offers a promising solution for achieving sustainable software development and efficient workflow execution in e-Science. Our contributions demonstrate that blockchain-based systems can provide decentralized, transparent, and secure platforms for managing soft-

ware development and scientific workflows. The integration of blockchain technology into scientific workflows, as demonstrated by the eBlocBroker infrastructure, holds significant promise for the future of e-Science. This study provides a foundational understanding of how blockchain can be leveraged to improve resource management, offering a robust and scalable solution that addresses current and future scientific research needs. The findings of this thesis underscore the significant potential of blockchain in e-Science, heralding a new era of efficient, transparent, and secure scientific workflows. Future research should address scalability issues and refine the algorithms and models presented in this thesis to enhance their applicability and performance in diverse scientific domains.

REFERENCES

1. Nakamoto, S., *Bitcoin: A peer-to-peer electronic cash system*, 2008, `https://bitcoin.org/bitcoin.pdf`, accessed on August 8, 2024.

2. Merkle, R., "DAOs, Democracy and Governance", *Cryonics Magazine*, Vol. 37, No. 4, pp. 28–40, 2016.

3. *Ethereum Project*, `https://ethereum.org`, accessed on August 8, 2024.

4. Wood, G., "Ethereum: A secure decentralised generalised transaction ledger", *Ethereum project yellow paper*, Vol. 151, No. 2014, pp. 1–32, 2014.

5. *Solidity*, `https://docs.soliditylang.org`, accessed on August 8, 2024.

6. Friederike Kleinfercher, J. L., Sandra Vengadasalam, *Bloxberg Whitepaper, the blockchain for science, v2.0*, `https://bloxberg.org/wp-content/uploads/2022/07/bloxberg_whitepaper_2.0.pdf`, accessed on August 8, 2024.

7. *Parity*, `https://github.com/openethereum/parity-ethereum`, accessed on August 8, 2024.

8. *Aura*, `https://openethereum.github.io/Aura.html`, accessed on August 8, 2024.

9. Benet, J., "IPFS - Content Addressed, Versioned, P2P File System", *CoRR*, Vol. abs/1407.3561, 2014.

10. von St. Vieth, B., "B2DROP - EUDAT's Personal Cloud Storage", *CS3 Workshop - Cloud Services for Synchronisation and Sharing*, 2015.

11. *B2DROP*, `https://eudat.eu/services/b2drop`, accessed on August 8, 2024.

12. *EUDAT - Research Data Services, Expertise and Technology Solutions*, `https://eudat.eu`, accessed on August 8, 2024.

13. Gentzsch, W., D. Lecarpentier and P. Wittenburg, "Big Data in Science and the EUDAT Project", *2014 Annual SRII Global Conference*, pp. 191–194, 2014.

14. Yoo, A. B., M. A. Jette and M. Grondona, "SLURM: Simple Linux Utility for Resource Management", D. Feitelson, L. Rudolph and U. Schwiegelshohn (Editors), *Job Scheduling Strategies for Parallel Processing*, pp. 44–60, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

15. Anderson, D., "BOINC: a system for public-resource computing and storage", *Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 4–10, 2004.

16. *European commission's digital single market policy*, `https://ec.europa.eu/commission/presscorner/detail/en/IP_16_1408`, accessed on August 8, 2024.

17. Alimoglu, A. and C. Özturan, "Design of a Smart Contract Based Autonomous Organization for Sustainable Software", *13th IEEE International Conference on e-Science, e-Science 2017, Auckland, New Zealand, October 24-27, 2017*, pp. 471–476, IEEE Computer Society, 2017.

18. Alimoğlu, A. and C. Özturan, "An autonomous blockchain-based computational broker for e-science", *Concurrency and Computation: Practice and Experience*, Vol. 36, No. 13, p. e8087, 2024.

19. Alimoğlu, A. and C. Özturan, "An autonomous blockchain-based workflow execution broker for e-science", *Cluster Computing*, 2024.

20. Foster, I., C. Kesselman and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *The International Journal of High Performance Computing Applications*, Vol. 15, No. 3, pp. 200–222, 2001.

21. Foster, I. and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*, The Elsevier Series in Grid Computing, Elsevier Science, 2004.

22. Gagliardi, F., R. Jones, F. Grey, M.-E. Begin and M. Heikkurinen, "Building an infrastructure for scientific Grid computing: Status and goals of the EGEE project", *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, Vol. 363, pp. 1729–42, 2005.

23. Prnjat, O., A. Balaz, D. Vudragovic, I. Liabotis, C. Sener, B. Marovic, M. Kozlovszky and G. Neagu, "SEE-GRID eInfrastructure for Regional eScience", S. C. Lin and E. Yen (Editors), *Data Driven e-Science*, pp. 91–103, Springer New York, New York, NY, 2011.

24. Özturan, C., V. Kotroni and E. Atanassov, "Development of virtual organizations, applications and services for earth science on grid e-Infrastructures", *Earth Science Informatics*, Vol. 3, pp. 197–198, 2010.

25. Antelmi, A., G. D'Ambrosio, A. Petta, L. Serra and C. Spagnuolo, "A Volunteer Computing Architecture for Computational Workflows on Decentralized Web", *IEEE Access*, Vol. 10, pp. 98993–99010, 2022.

26. *Ethernity Cloud*, `https://ethernity.cloud/ethernity_whitepaper.pdf`, accessed on August 8, 2024.

27. *Golem*, `https://golem.network/`, accessed on August 8, 2024.

28. *The Golem Project Crowdfunding - Whitepaper*, `https://golem.network/doc/Golemwhitepaper.pdf`, accessed on August 8, 2024.

29. *iExec*, `https://iex.ec/`, accessed on August 8, 2024.

30. *iExec Whitepaper, Blockchain-Based Decentralized Cloud Computing*, `https://github.com/iExecBlockchainComputing/whitepaper/raw/master/V3/iExec-W`

Pv3.0-English.pdf, accessed on August 8, 2024.

31. *SONM*, https://web.archive.org/web/20231206201401/https://sonm.com/, accessed on Dec 6, 2023.

32. *SONM Supercomputer Organized by Network Mining - Whitepaper*, https://web.archive.org/web/20230607153339/https://whitepaper.io/document/326/sonm-whitepaper, accessed on June 7, 2023.

33. *Fluence*, https://fluence.network, accessed on August 8, 2024.

34. *Polygon*, https://polygon.technology, accessed on August 8, 2024.

35. *Golem Service Bus*, https://github.com/golemfactory/ya-service-bus, accessed on August 8, 2024.

36. *gftp*, https://github.com/golemfactory/yagna/tree/master/core/gftp, accessed on August 8, 2024.

37. Lodygensky, O., E. Urbah and S. Dadoun, *Xtremweb-HEP: Designing desktop grid for the EGEE infrastructure*, pp. 79–98, Chapman and Hall/CRC, 2012.

38. *The Cocaine Project*, https://github.com/Cocaine/, accessed on August 8, 2024.

39. *Samba (software)*, https://www.samba.org, accessed on August 8, 2024.

40. Topcuoglu, H., S. Hariri and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 3, pp. 260–274, 2002.

41. Thain, D., T. Tannenbaum and M. Livny, "Distributed computing in practice: the Condor experience", *Concurrency and Computation: Practice and Experience*,

Vol. 17, No. 2-4, pp. 323–356, 2005.

42. Ludäscher, B., I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao and Y. Zhao, "Scientific workflow management and the Kepler system", *Concurrency and Computation: Practice and Experience*, Vol. 18, No. 10, pp. 1039–1065, 2006.

43. Papadimitriou, C. and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover Books on Computer Science, Dover Publications, 1998.

44. Jaradat, W., A. Dearle and A. Barker, "Towards an autonomous decentralized orchestration system", *Concurrency and Computation: Practice and Experience*, Vol. 28, No. 11, pp. 3164–3179, 2016.

45. Blythe, J., S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal and K. Kennedy, "Task scheduling strategies for workflow-based applications in grids", *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, Vol. 2, pp. 759–767, 2005.

46. Braun, T. D., H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen and R. F. Freund, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems", *Journal of Parallel and Distributed Computing*, Vol. 61, No. 6, pp. 810–837, 2001.

47. Chen, W. and E. Deelman, "Partitioning and Scheduling Workflows across Multiple Sites with Storage Constraints", *Parallel Processing and Applied Mathematics*, Vol. 7204, pp. 11–20, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, series Title: Lecture Notes in Computer Science.

48. Pawlik, A., J. Segal, H. Sharp and M. Petre, "Crowdsourcing scientific software documentation: a case study of the NumPy documentation project", *Computing*

in Science & Engineering, Vol. 17, No. 1, pp. 28–36, 2015.

49. Mao, K., L. Capra, M. Harman and Y. Jia, "A survey of the use of crowdsourcing in software engineering", *Journal of Systems and Software*, Vol. 126, pp. 57–84, 2017.

50. Jentzsch, C., *Decentralized autonomous organization to automate governance*, 2016, `https://download.slock.it/public/DAO/WhitePaper.pdf`, accessed on August 8, 2024.

51. Daian, P., *Analysis of the DAO exploit*, 2016, `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`, accessed on August 8, 2024.

52. *Drips*, `https://docs.drips.network`, accessed on August 8, 2024.

53. *OpenZeppelin Governance*, `https://docs.openzeppelin.com/contracts/4.x/governance`, accessed on August 8, 2024.

54. Kouzes, R. T., J. D. Myers and W. A. Wulf, "Collaboratories: Doing science on the Internet", *Computer*, Vol. 29, No. 8, pp. 40–46, 1996.

55. Chue Hong, N., "Software: the hidden infrastructure behind the world's largest scientific facilities", *Innovation Into Success - The Quarterly Journal of UKSPA*, Vol. 31, pp. 28–29, 2013.

56. Niemeyer, K. E., A. M. Smith and D. S. Katz, "The challenge and promise of software citation for credit, identification, discovery, and reuse", *ACM Journal of Data and Information Quality (JDIQ)*, Vol. 7, No. 4, p. 16, 2016.

57. Hasan, R., S. Purawat, C. Olschanowsky and I. Altintas, "Preserving File Provenance Using Principles of Blockchain to Ensure Scientific Reproducibility", *2023 IEEE 19th International Conference on e-Science (e-Science)*, pp. 1–7, 2023.

58. *EIP 105 (Serenity): Binary sharding plus contract calling semantics 53*, https://github.com/ethereum/EIPs/issues/53, accessed on August 8, 2024.

59. *Horizontal scalability/Sharding 142*, https://github.com/EOSIO/eos/issues/142, accessed on August 8, 2024.

60. *On sharding blockchains*, https://github.com/ethereum/wiki/wiki/Sharding-FAQ, accessed on August 8, 2024.

61. Hauser, B., *Brownie*, https://github.com/eth-brownie/brownie, accessed on August 8, 2024.

62. *Web3.py*, https://github.com/ethereum/web3.py, accessed on August 8, 2024.

63. *Ganache*, https://trufflesuite.com/ganache/, accessed on August 8, 2024.

64. *ownCloud client library for Python*, https://github.com/owncloud/pyocclient, accessed on August 8, 2024.

65. Bailey, D. H., D. H. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, R. Fatoohi, S. Fineberg, P. O. Frederickson, T. Lasinski, R. Schreiber, H. D. Simon and V. Venkatakrishnan, "The Nas Parallel Benchmarks", *International Journal of High Performance Computing Applications*, Vol. 5, pp. 63 – 73, 1991.

66. Kara, G. and C. Özturan, "Algorithm 1002: Graph Coloring Based Parallel Push-relabel Algorithm for the Maximum Flow Problem", *ACM Transactions on Mathematical Software (TOMS)*, Vol. 45, No. 4, pp. 1–28, 2019.

67. *eBlocBroker Test Results*, https://github.com/ebloc/ebloc-broker/tree/v2.3.1/broker/results, accessed on August 8, 2024.

68. *Ethereum Block Time History*, https://etherscan.io/chart/blocktime, accessed on August 8, 2024.

69. *Bloxberg Faucet*, https://faucet.bloxberg.org/, accessed on August 8, 2024.

70. *Etherscan*, https://etherscan.io/chart/gasprice/, accessed on August 8, 2024.

71. *Polygonscan*, https://polygonscan.com/chart/gasprice/, accessed on August 8, 2024.

72. *eBlocBroker*, https://github.com/ebloc/ebloc-broker, accessed on August 8, 2024.

73. Taylor, I. J., E. Deelman, D. B. Gannon and M. Shields, *Workflows for E-Science: Scientific Workflows for Grids*, Springer-Verlag, Berlin, Heidelberg, 2006.

74. Deelman, E., J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi and M. Livny, "Pegasus: Mapping Scientific Workflows onto the Grid", *Grid Computing*, Vol. 3165, pp. 11–20, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, series Title: Lecture Notes in Computer Science.

75. *Topological Generations*, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.dag.topological_generations.html, accessed on August 8, 2024.

76. *eBlocBroker Workflow Scripts*, https://github.com/ebloc/ebloc-broker/tree/v2.3.5/broker/test_setup_w, accessed on August 8, 2024.

77. *NetworkX*, https://networkx.org, accessed on August 8, 2024.

78. Rocklin, M., D. Nikel-Shepherd and C. Rodrigues, *Heterogeneous Earliest Finish Time*, https://github.com/mrocklin/heft, accessed on August 8, 2024.

79. *eBlocBroker Workflow Test Results*, https://github.com/ebloc/ebloc-broker/blob/v2.3.6/broker/results/eblocbroker_wf_test-results_October_2

023.xlsx, accessed on August 8, 2024.

80. *The General Purpose Zero-Knowledge VM*, https://www.risczero.com, accessed on August 8, 2024.

81. *Artificial Intelligence Act*, https://www.europarl.europa.eu/RegData/etudes/BRIE/2021/698792/EPRS_BRI(2021)698792_EN.pdf, accessed on August 8, 2024.

82. Wood, G., *Condition Oriented Programming*, https://blog.ethcore.io/condition-oriented-programming-2, accessed on August 8, 2024.

83. *Bloxberg ResearchCertificate*, https://blockexplorer.bloxberg.org/address/0x3fb704dfDB72Fc06860D9F09124C30919488f13C/contracts#address-tabs, accessed on August 8, 2024.

84. Chue Hong, N., *Setting up a Software Seal of Approval*, 2017, https://figshare.com/articles/Setting-up-a-Software-Seal-of-Approval/4737178/1, accessed on August 8, 2024.

APPENDIX A: COPYRIGHT NOTICE

All visual content in this thesis book is produced by the author for his own publications [17–19] and then reused in this thesis book. Visual content produced in the scope of this thesis study whose copyrights are transferred to a publisher are used in accordance with the publisher's current policies found in the publisher's web site as of this writing in regards to the reuse of text and visual content produced by the authors themselves in this thesis book.