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

Ph.D. Thesis Defense

Alper Alimoğlu

Advisor: Prof. Can Özturan

Defense Committee:
Prof. Cem Ersoy
Prof. Arda Yurdakul
Prof. D. Turgay Altılar
Prof. Öznur Özkasap

Department of Computer Engineering
September 29, 2025

Table of Contents

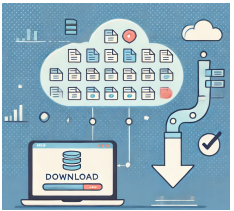
Motivation

- Centralized resource allocation $\rightarrow$ Decentralized resource allocation
- Decentralized Autonomous Organization (DAO)
- Provide an ecosystem where providers can sell their idle computing power to others and earn additional income.
- Supporting Slurm-managed clusters since the academicians and researchers commonly use Slurm.
- Computation + Data

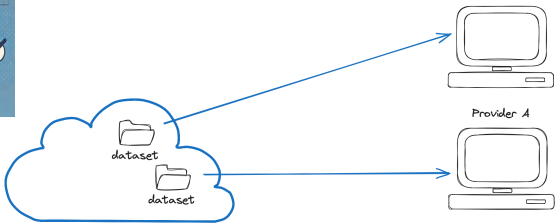
Problem

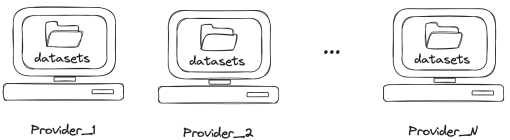
How can we facilitate an efficient way to share and allocate computational and data resources within research communities?

Motivation



DAO
eBlocBroker

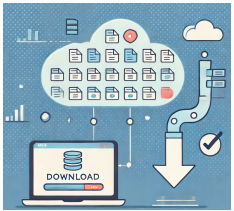


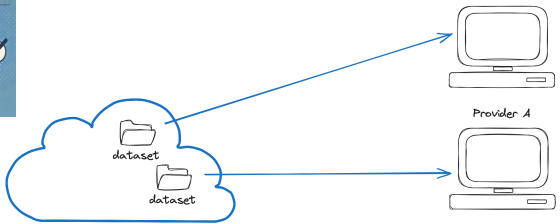


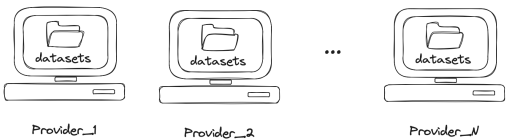
Motivation

Computation

DAO
eBlocBroker





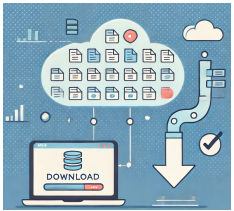


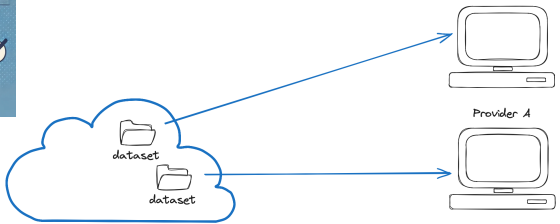
Motivation

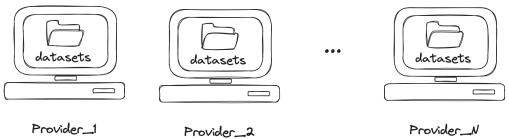
Computation

DAO
eBlocBroker

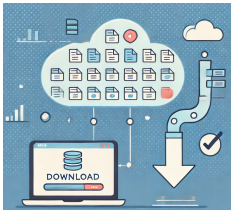
Data

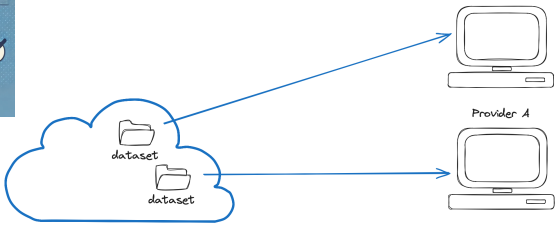


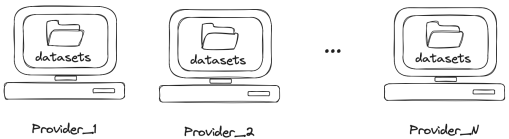




Motivation


DOWNLOAD


dataset
dataset
Provider A
Provider B


datasets
datasets
...
datasets
Provider_1
Provider_2
Provider_N

Computation
DAO
eBlocBroker
Cost
Model
Data

Major Components of the Thesis

1. EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science

Major Components of the Thesis

1. EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science
2. An Autonomous Blockchain-Based Workflow Execution Broker for e-Science

Major Components of the Thesis

1. EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science
2. An Autonomous Blockchain-Based Workflow Execution Broker for e-Science
3. Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

Major Components of the Thesis

1. EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science
2. An Autonomous Blockchain-Based Workflow Execution Broker for e-Science
3. Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

Major Components of the Thesis

1. EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science
2. An Autonomous Blockchain-Based Workflow Execution Broker for e-Science
3. Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

Major Components of the Thesis

1. EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science
2. An Autonomous Blockchain-Based Workflow Execution Broker for e-Science
3. Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

Major Components of the Thesis

1. EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science
2. An Autonomous Blockchain-Based Workflow Execution Broker for e-Science
3. Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

Contributions of the Thesis

1 EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science

eScience

- 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.
- Data staging solutions for data stored on centralized cloud storage facility (B2DROP, Google Drive), and decentralized storage (distributed IPFS file system).
- Implementation of a cost model in the eBlocBroker smart contract, which calculates and records computation, data transfer, storage, and cache usage costs.
- An ERC20 token standard-based payment system for purchasing computing and data resources.

Contributions of the Thesis

1 EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science

eScience

- 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.
- Data staging solutions for data stored on centralized cloud storage facility (B2DROP, Google Drive), and decentralized storage (distributed IPFS file system).
- Implementation of a cost model in the eBlocBroker smart contract, which calculates and records computation, data transfer, storage, and cache usage costs.
- An ERC20 token standard-based payment system for purchasing computing and data resources.

Contributions of the Thesis

1 EBlocBroker: An Autonomous Blockchain-base Computational Broker For e-Science

eScience

- 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.
- Data staging solutions for data stored on centralized cloud storage facility (B2DROP, Google Drive), and decentralized storage (distributed IPFS file system).
- Implementation of a cost model in the eBlocBroker smart contract, which calculates and records computation, data transfer, storage, and cache usage costs.
- An ERC20 token standard-based payment system for purchasing computing and data resources.

Contributions of the Thesis

eScience

- 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.
- Data staging solutions for data stored on centralized cloud storage facility (B2DROP, Google Drive), and decentralized storage (distributed IPFS file system).
- Implementation of a cost model in the eBlocBroker smart contract, which calculates and records computation, data transfer, storage, and cache usage costs.
- An ERC20 token standard-based payment system for purchasing computing and data resources.

Contributions of the Thesis

2 An Autonomous Blockchain-Based Workflow Execution Broker for e-Science

- 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 a workflow engine on top of the eBlocBroker infrastructure that will be responsible for executing workflows on distributed providers through blockchain.
  - Partition workflows
  - Schedule workflows for providers
  - Enable 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.

Contributions of the Thesis

2 An Autonomous Blockchain-Based Workflow Execution Broker for e-Science

- 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 a workflow engine on top of the eBlocBroker infrastructure that will be responsible for executing workflows on distributed providers through blockchain.
  - Partition workflows
  - Schedule workflows for providers
  - Enable 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.

Contributions of the Thesis

2 An Autonomous Blockchain-Based Workflow Execution Broker for e-Science

- 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 a workflow engine on top of the eBlocBroker infrastructure that will be responsible for executing workflows on distributed providers through blockchain.
  - Partition workflows
  - Schedule workflows for providers
  - Enable 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.

Contributions of the Thesis

3 Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

- Design and implement a smart contract called AutonomousSoftwareOrg, which is a DAO-based smart contract. Purpose for e-Science:
  - Facilitate open-source software development
  - Ensure sustainability and reproducibility of software
- 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 software can be recorded to facilitate reproducibility checks for organizations
- 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 executions

Contributions of the Thesis

3 Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

- Design and implement a smart contract called AutonomousSoftwareOrg, which is a DAO-based smart contract. Purpose for e-Science:
  - Facilitate open-source software development
  - Ensure sustainability and reproducibility of software
- 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 software can be recorded to facilitate reproducibility checks for organizations
- 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 executions

Contributions of the Thesis

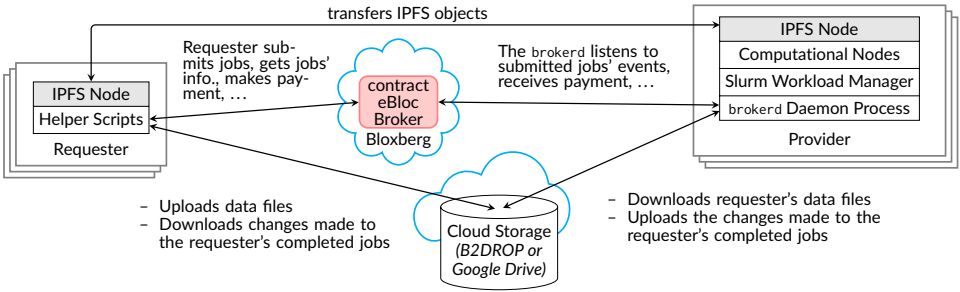
3 Design of a Smart Contract Based Autonomous Organization for Sustainable Software and e-Science

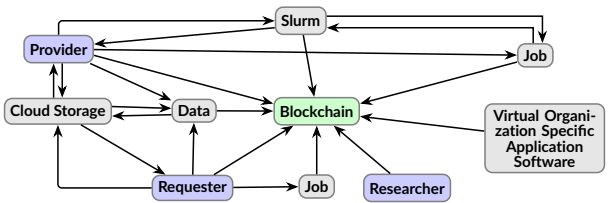
- Design and implement a smart contract called AutonomousSoftwareOrg, which is a DAO-based smart contract. Purpose for e-Science:
  - Facilitate open-source software development
  - Ensure sustainability and reproducibility of software
- Each software execution trace is recorded on the blockchain, showcasing the connection between input and output data files for each execution
  - The hash of the generated outputs from software can be recorded to facilitate reproducibility checks for organizations
- 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 executions

Table of Contents

Introduction

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





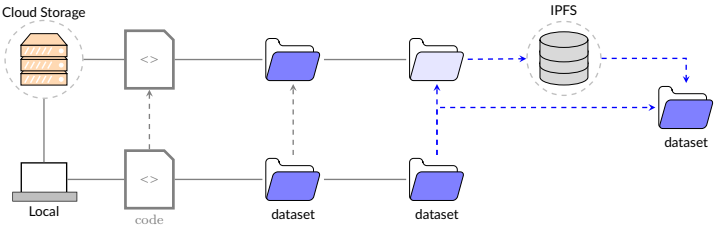
Related Work

Table: 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**	**Work-flow 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	✓	✓	✗	✗	✓	✓

Example: Deployment of a Job with Multiple Datasets

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



- **Compression Requirement:** Requesters should compress the source code of the job and data files.
- **Reproducible Tarballs:**
  - Use sort order during the tar process.
  - Do not preserve the timestamp and `.git` folder in the compressed file using gzip.
- **Hashing:** Use the MD5 hashing algorithm.

- Slurm script:

```
#!/bin/bash

BASE="../data_link/"
DATA1_DIR=$BASE"47b0f5f1a882d1c15661bc681de8230a"
DATA2_DIR=$BASE"45281dfec4618e5d20570812dea38760"
DATA3_DIR=$BASE"QmZYUCRTp3iKusMdQP1oWKHpkCDzinM4j2VdATxXSNEVZv"

g++ main.cpp -o main
./main $DATA1_DIR/data.csv $DATA2_DIR/data.csv $DATA3_DIR/data.csv
cat $DATA1_DIR/data1.txt >> completed.txt
cat $DATA2_DIR/data2.txt >> completed.txt
```

Example: Deployment of a Job with Multiple Datasets

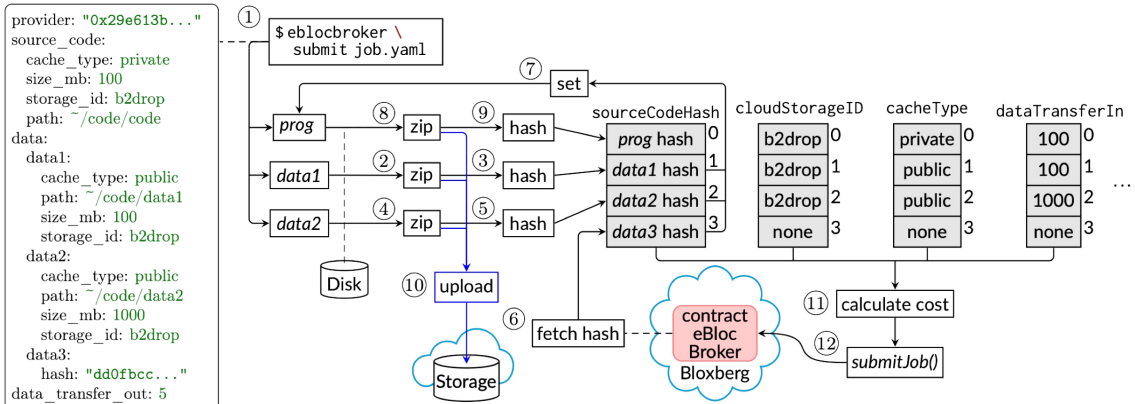


Figure: Creating a new job and sending it to the provider via eBlocBroker. The dashed arrows represent from which the data was fetched.

Example for Processing a Job on the Provider

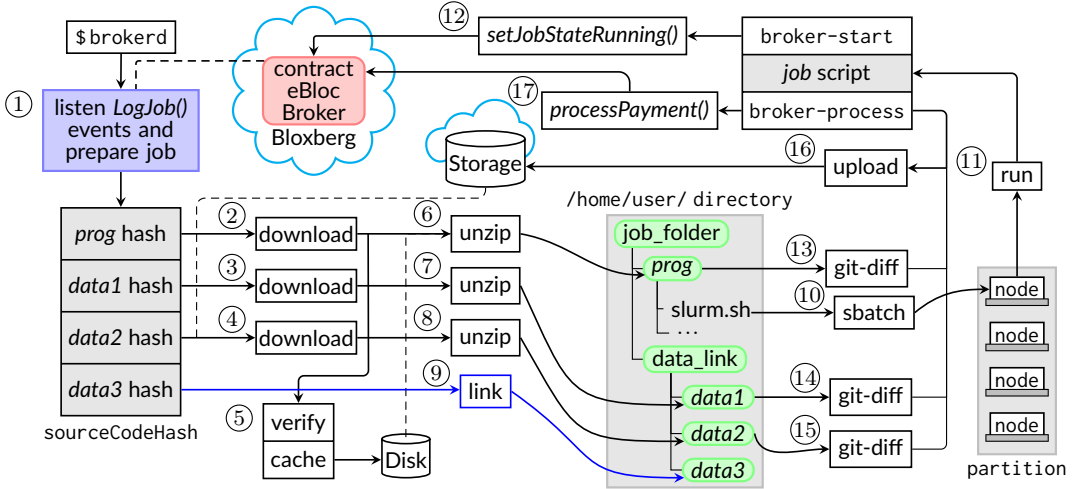
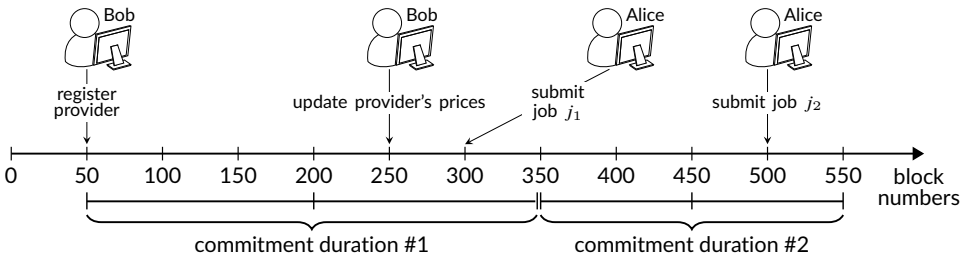


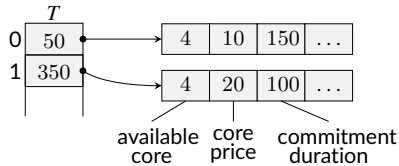
Figure: A breakdown of the steps the provider takes to receive a job, download the job's corresponding data, submit the job to Slurm, upload generated results to the cloud storage, and obtain the payments is outlined.

Provider's Price Commitment

(a)



(b)



The eBlocBroker Smart Contract Implementation

```
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: Signatures of the eBlocBroker smart contract's fundamental functions.

Gas Consumption Limitation

The deployment of transaction executions is bounded by the block gas limit [1]. The gas limit differs from blockchain to blockchain.

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.
- 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.

Cost Model

The total cost includes computation, data transfer, storage, and caching costs.

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: $J = \{j_1, j_2, \ldots, j_{\lvert J \rvert}\}$
D_j	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
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
$M_p(a_{jr}, h_d)$	Hours remaining to store with h_d on provider p

$cost \leftarrow I_j \cdot X_j \cdot F_p^{\text{cpu}}$
$cost \leftarrow cost + \delta_{D_j} \cdot F_p^{\text{trans}}$
for each data in data set ($d \in D_j$) **do**
    if data d is paid for and stored on provider p **then**
        continue
    end if
    if data d is registered on provider p **then**
        $cost \leftarrow cost + F_{p,d}^{\text{data}}$
    else
        if data d is requested to be stored on provider p **then**
            $cost \leftarrow cost + Z_d \cdot F_p^{\text{store}}$
        else
            $cost \leftarrow cost + Z_d \cdot F_p^{\text{cache}}$
        end if
        $cost \leftarrow cost + Z_d \cdot F_p^{\text{trans}}$
    end if
end for

Cost Model

The total cost includes computation, data transfer, storage, and caching costs.

$$R(n,a,h,y,z,u)=\begin{cases}\left(\underbrace{z\cdot F_p^{\text{trans}}}_{\substack{\text{transfer cost}\\ [\mathbf{C}]}}+\underbrace{z\cdot y\cdot F_p^{\text{storage}}}_{\substack{\text{storage cost}\\ [\mathbf{D}]}}+\underbrace{z\cdot F_p^{\text{cache}}\cdot[\![y=0]\!]}_{\substack{\text{cache cost}\\ [\mathbf{E}]}}\right)\cdot[\![M_p(a,h)=0]\!] & \text{if } u \text{ is false}\\ \underbrace{F_{p,h}^{\text{data}}}_{\substack{\text{dataset cost}\\ [\mathbf{F}]}} & \text{otherwise.}\end{cases} \tag{1}$$

$$C\left(n,a_{jr},H_{D_j},Y_{D_j},Z_{D_j},U_{D_j},\delta_{D_j},I_j,X_j,D_j\right)=\underbrace{I_j\cdot X_j\cdot F_p^{\text{cpu}}}_{\substack{\text{computation cost}\\ [\mathbf{A}]}}+\underbrace{\underbrace{\delta_{D_j}\cdot F_p^{\text{trans}}}_{[\mathbf{C}]}+\sum_{d\in D_j}R\left(n,a_{jr},h_d,y_d,z_d,u_d\right)}_{\substack{\text{data cost}\\ [\mathbf{B}]}} \tag{2}$$

Data Structure used in eBlocBroker Smart Contract

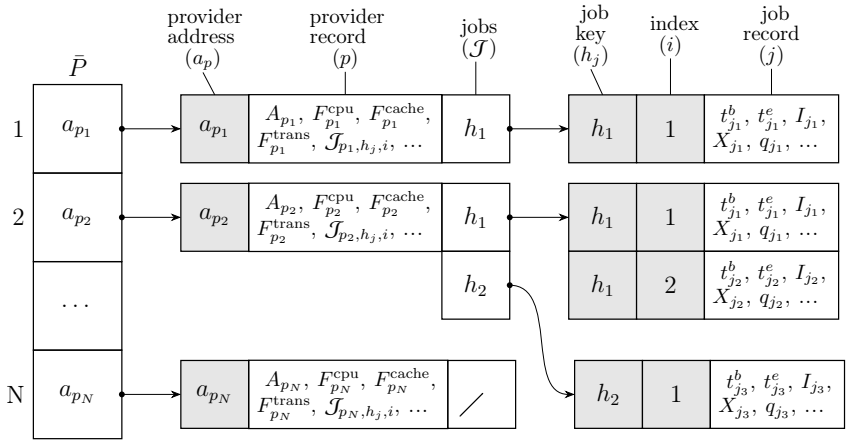


Figure: The data structure for storing provider records p on each provider's storage array, which includes job records. Nodes that are lookup values are colored gray.

Tests and Results

Before testing in real environments, the eBlocBroker smart contract was developed and tested using Brownie. Afterward, we deployed eBlocBroker smart contract on Bloxberg and tested it using two types of synthetic CPU workloads explained as follows:

- *Workload-1* tests to submit a job that only requires the source code of the job. It uses the NAS Parallel Benchmarks [2].
- *Workload-2* evaluates executing source code with pre-cached and non-cached datasets on providers, selecting the cheapest provider.
  - As the source code, the parallel network flow application [3] 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 processes run consecutively with different randomly selected datasets.
  - Selected three datasets are decided as follows: two datasets are the provider's registered data, and one is from the requester's local storage.

In our test, all providers have the same fixed prices for all except the fee for the datasets.

Tests and Results

The script program we utilize to submit jobs from the requester node executes the following tasks:

- 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 4 to 8 minutes.
- All source codes contain a Slurm script that is used to execute it.
- Each source code and data file are compressed separately before being uploaded into the cloud storage.
- Cloud storage service is randomly selected for each job submission, which will be shared between the requester and the provider.
- Our cost model calculates the job's cost for each provider and selects the provider with the lowest price.

Tests and Results

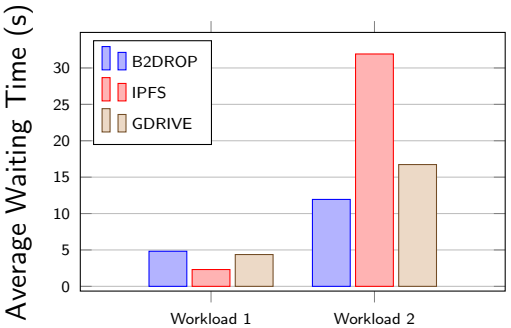


Figure: Comparison of job waiting times (in seconds) for our workloads.

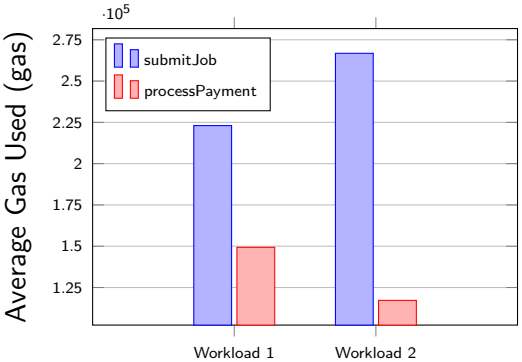


Figure: Comparison of gas cost for submitJob and processPayment.

Tests and Results

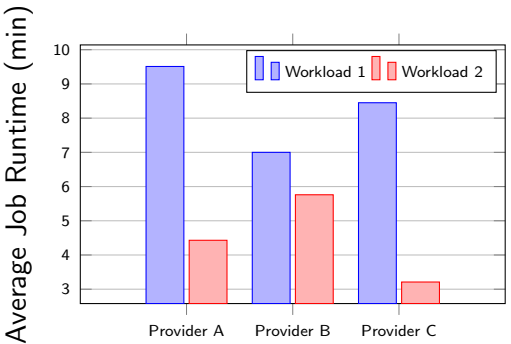


Figure: Comparison of job run times (in minutes) for our workloads in Slurm.

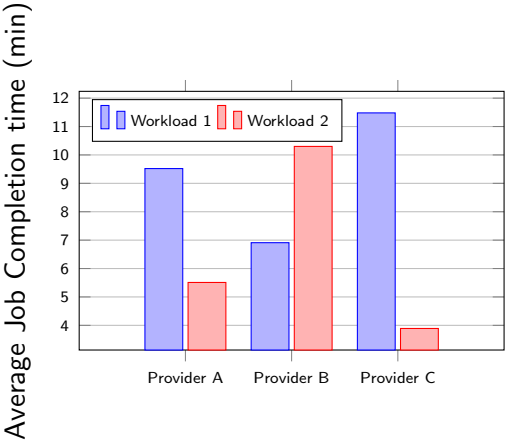


Figure: Comparison of job completion times (in minutes) for our workloads in Slurm.

Tests and Results

- Tested the robustness of our smart contract, Python scripts, and `brokerd`.
- Results showed successful task completion by providers, except for some failures in downloading large files or packages and occasional unresponsive Bloxberg nodes.
- Job owners receive refunds for these unfinished jobs and can rerun them since their information is stored.

Table: 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

Table of Contents

Motivation

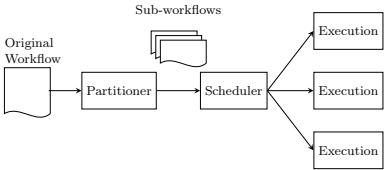
- **Usage:**
  - Workflows are widely used by scientific computing communities.
- **Need:**
  - Researchers and academicians require a workflow execution service.
- **Motivation:**
  - The demand for a workflow execution service was the primary motivation for integrating it into the eBlocBroker infrastructure.
- **Novelty:**
  - While distributed workflow execution exists, our work is unique.
  - It is the first to offer a blockchain-based workflow execution service.
  - No other known works provide a similar blockchain-based solution.

Task-Scheduling Algorithms Used

- Used Heterogeneous Earliest Finish Time (HEFT) Algorithm Topcuoğlu et al. [4].
  - Task Prioritizing Phase: Set the priorities of the tasks and select the tasks based on their priorities.
  - Processor Selection Phase: Map and schedule each selected task onto a processor.
- Used Topological generations layering (TGL) [5] scheduling by NetworkX [6].
- Many scheduling heuristics, such as HEFT, MinMin [7], Max-Min, and MCT [8], attempt to solve the workflow mapping problem.

Our Approach

- Our approach consists of three phases:
  - inputting a workflow
  - partitioning it for scheduling
  - submission to computational providers for execution



- HEFT and TGL algorithms require that an application's characteristics are known a priori, including:
  - Run times
  - Data sizes for communication between jobs
  - Job dependencies
- The partitioner takes the original workflow as input, which includes:
  - Information about the estimated time of each job
  - The DAG of the workflow
  - The calculated cost of each job
- The partitioner then produces various sub-workflows utilizing a given scheduler (e.g., HEFT or TGL) and executes them based on their dependencies on each other.

Our Approach

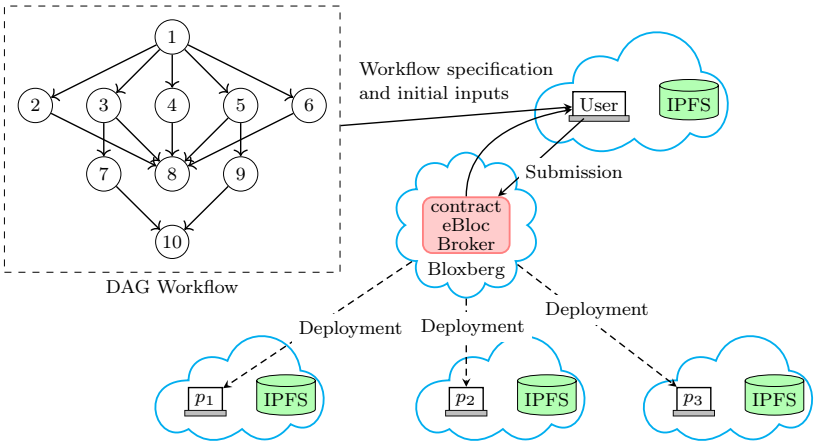


Figure: Overview of our distributed architecture and the interactions between the user and providers. Source code and data files are transferred between providers and users through IPFS.

Directed Acyclic Graph (DAG)

- A workflow is represented by a DAG, $G = (V, E)$:
  - V is the set of v jobs.
  - E is the set of e edges between the jobs.
- Each edge $(i, m) \in E$ defines the precedence constraint:
  - Job j_i must complete its execution before job j_m starts.
- $data_{i,m}$ is the quantity of data that needs to be transferred from job j_i to job j_m.

NetworkX, a Python package, is employed to conduct graph analysis and create a workflow of jobs with dependencies between them.

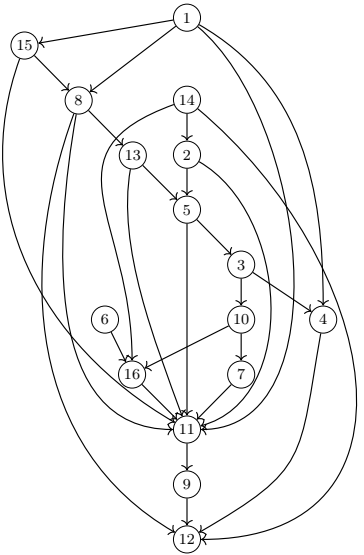


Figure: A randomly generated DAG workflow that has 16 nodes and 28 edges.

Layer Explanation

- Characteristics of topologically generated sets of nodes:
  - Ancestors of any node are included in the previous layer.
  - Descendants of a node are part of the following layer.
  - Nodes are assigned to the earliest layer level they can be a part of.

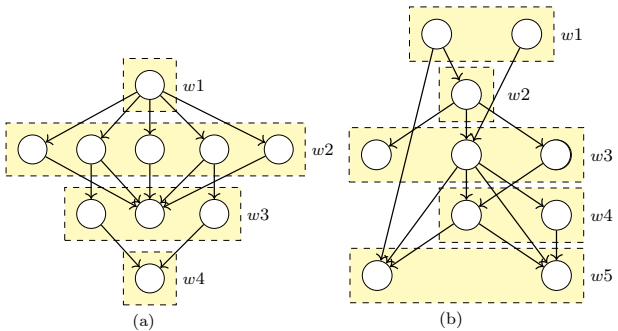


Figure: Example view of topological generations layering on randomly generated DAGs.

Revised Cost Model for Workflow

The total cost is composed of computation costs which involve running multiple workflow partitions on various providers.

Symbol	Meaning
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: $D_j = \{d_1, d_2, \ldots, d_{\lvert D_{W_i} \rvert}\}$
$\delta_{D_{W_i}}$	Size of the output files as patches in MB generated by the workflow W_i on each D_{W_i} required to be uploaded

$$C_{W_i,p} = \sum_{j \in W_i} U_j \cdot X_j \cdot F_p^{\text{cpu}} \tag{3}$$

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: $\quad \ldots$
8: **end for**

Workflow Engine Algorithms

Our scheduler coordinates scheduling from the user side rather than the provider side.

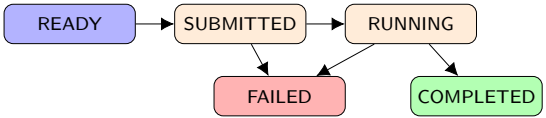


Figure: State changes of a job.

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 P and submit them to its corresponding provider p_k.
7: **end for**

Figure: The workflow engine algorithm to distribute jobs using the topological generations [5] algorithm

Workflow Engine Algorithms

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: The workflow engine algorithm to distribute jobs using the HEFT [4, 9] algorithm

Experimental Results of the HEFT and TGL algorithms

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:

- 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.

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: Description of prices on provider p.

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

Experimental Results of the HEFT and TGL algorithms

We submit workflows in the simulated real environment using HEFT and TGL scheduling algorithms.

Test	Workflow		Method	Completion Time (min)	Gas Used for *submitJob* (gas)	Gas Used for *processPayment* (gas)	Actual/Complete Cost (*Cent*)	Failed
	$\|V\|$	$\|E\|$						
T_1	16	28		27	1722983	1942770	0.827/0.827	0
T_2	32	56		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		100	13858812	16524333	6.867/7.025	2
T_5	256	448		191.5	27859947	33574732	13.909/14.060	3
T_6	16	28		48.5	3533072	1932928	1.768/1.768	0
T_7	32	56		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		222	21100398	15922550	12.243/12.243	0
T_{10}	256	448		266	29775297	32933948	19.511/19.528	1

Experimental Results of the HEFT and TGL algorithms

During these 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.

- Every provider has a Flask web server that will provide information on the state of Slurm nodes and partitions, such as available cores.

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

Test	Workflow		Method	Completion Time (min)	Gas Used for *submitJob* (gas)	Gas Used for *processPayment* (gas)	Actual/Complete Cost (*Cent*)	Failed
	$\|V\|$	$\|E\|$						
T_{11}	128	224	HEFT	102	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

Discussion

- **TGL Algorithm**
  - More aggressively distributes jobs to each provider
  - Equally splits job sets to providers
  - Results in additional data transfer costs
  - Multiple layers may contain a single job that slows down workflow completion
- **HEFT Algorithm**
  - Distributes jobs to minimize data transfer costs
- **Conclusion**
  - Completion time takes longer for TGL than HEFT for all given node values.
  - TGL is not an efficient algorithm for workflow partitioning due to higher data transfer costs.

Table of Contents

Problem

Can we develop an unstoppable virtual organization that will:

- Represent a open-source software
- Keep execution records of software by computing service providers, and provide transactional data for analysis and reproducibility checks
- Continue to exist even after the related projects have ended
- Offer developers decision-making, crowdfunding, and citation mechanisms

Establishing a company and selling the software requires a lot of capital and bureaucratic work. A virtual organization that runs as smart contract can be reasonable alternative to this.

Proposal

Smart contract solutions for enhancing software sustainability and execution traceability:

- *Development and Community:* Funding mechanism and decision-making process based on voting as an unstoppable virtual software organization for the software community.
- *Professionalization:* Efforts of developers are better assessed quantitatively using the blockchain data.
- *Credit:* Blockchain based credit and citation ecosystem. Citations by paper authors and software usages by other software.
- *Software publishing:* Help software discoverability and reuse.
- *Software reproducibility:* Conferences, journals and funding agencies can require software execution records to be stored on the blockchain. Hashes of inputs and outputs can be recorded on the blockchain.
  - The importance of software reproducibility is increasing with AI advancements. In April 2021, the European Commission proposed an EU regulatory structure on AI.
  - The draft AI act [10] is the first attempt to enact horizontal AI regulations.
  - The data files utilized for training an AI model and the generated data hold significant importance.

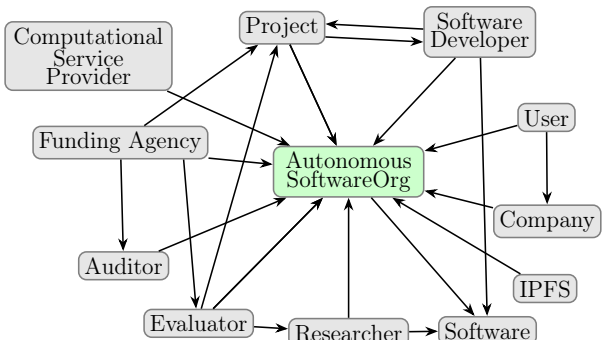
Previous Work

- Crowdsourced development has become increasingly significant in building successful platforms like OpenStreetMap, Instagram, Weather Underground, and Kickstarter.
- 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).
- The first Distributed Autonomous Organization on the Ethereum platform called The DAO was implemented in Spring 2016.
- Drips [11], an application developed on the Ethereum platform, enables flexible support of open-source projects in GitHub by sending funds with built-in dependency splitting.
- There has been an approach [12] 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.

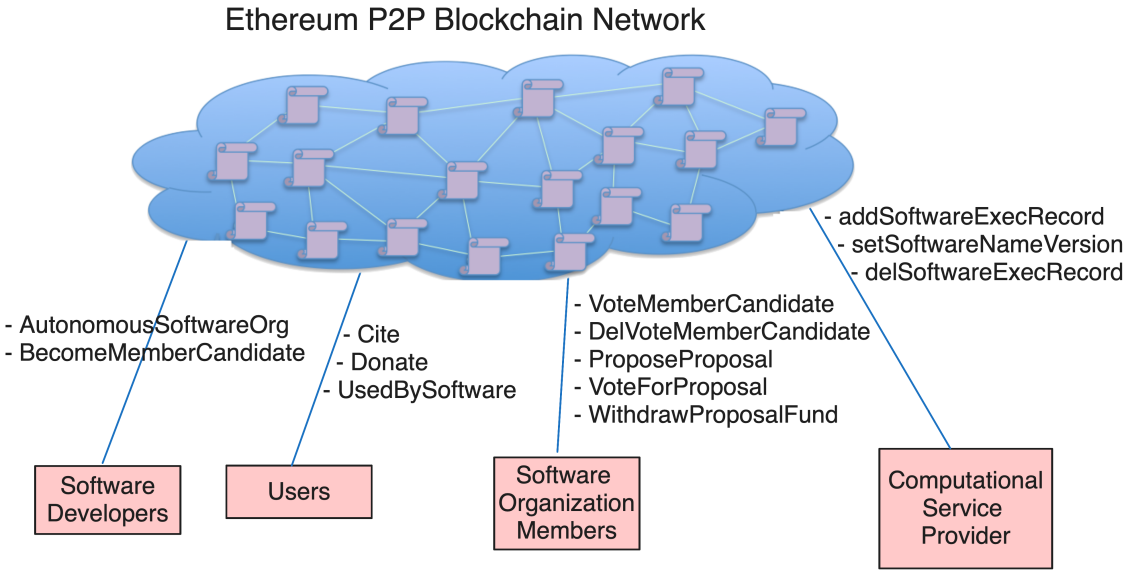
Autonomous Software Organization

A smart contract running continuously on a blockchain can be the autonomous entity that represents the software infrastructure. AutonomousSoftwareOrg presents a unstoppable, uninterruptible, smart contract. AutonomousSoftwareOrg smart contract provides:

- a funding mechanism based on crypto-currencies
- a democratic mechanism for decision making mechanism based on voting
- record keeping for software usage citations and executions



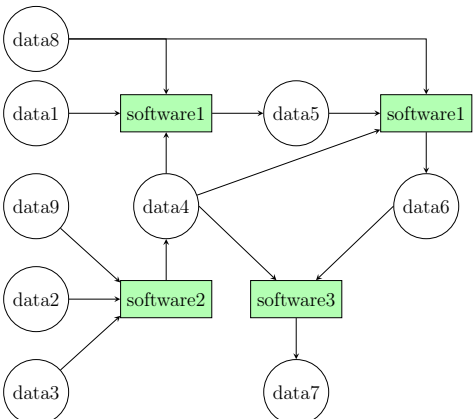
Blockchain-based autonomous organization model



AND/OR graph model for data and software execution trace

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.

Figure illustrates a general AND/OR graph of software executions, where datasets are used among various software.



The AutonomousSoftwareOrg Smart Contract Implementation

For testing purposes we copied and modified the Bloxberg ResearchCertificate smart contract [13], and added new functions.

```
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)
}
```

```
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)
```

The AutonomousSoftwareOrg Smart Contract Implementation

```
  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
}
```

The addSoftwareExecRecord() Function

```
function addSoftwareExecRecord(bytes32 sourceCodeHash, uint32 index, bytes32[] memory inputHash, bytes32[] memory outputHash)
    public member(msg.sender) validEblocBrokerProvider() returns (uint32) {
    if (index == 0) {
        globalIndexCounter += 1;
        softwareExecutionRecordOwner.push(msg.sender);
        index = globalIndexCounter;
    }
    else {
        require(softwareExecutionRecordOwner[index] == msg.sender);
    }

    softwareExecutionNumber += 1;
    ResearchCertificate(ResearchCertificateAddress).createCertificate(msg.sender, sourceCodeHash);
    for (uint256 i = 0; i < inputHash.length; i++) {
        uint256 tokenIndex = ResearchCertificate(ResearchCertificateAddress).createCertificate(msg.sender, inputHash[i]);
        incoming[sourceCodeHash][index].push(tokenIndex);
    }
    incomingLen[sourceCodeHash][index] = incomingLen[sourceCodeHash][index] + inputHash.length;
    for (uint256 i = 0; i < outputHash.length; i++) {
        uint256 tokenIndex = ResearchCertificate(ResearchCertificateAddress).createCertificate(msg.sender, outputHash[i]);
        outgoing[sourceCodeHash][index].push(tokenIndex);
    }
    outgoingLen[sourceCodeHash][index] = outgoingLen[sourceCodeHash][index] + outputHash.length;
    emit LogSoftwareExecRecord(msg.sender, sourceCodeHash, index, inputHash, outputHash);
    return index;
}
```

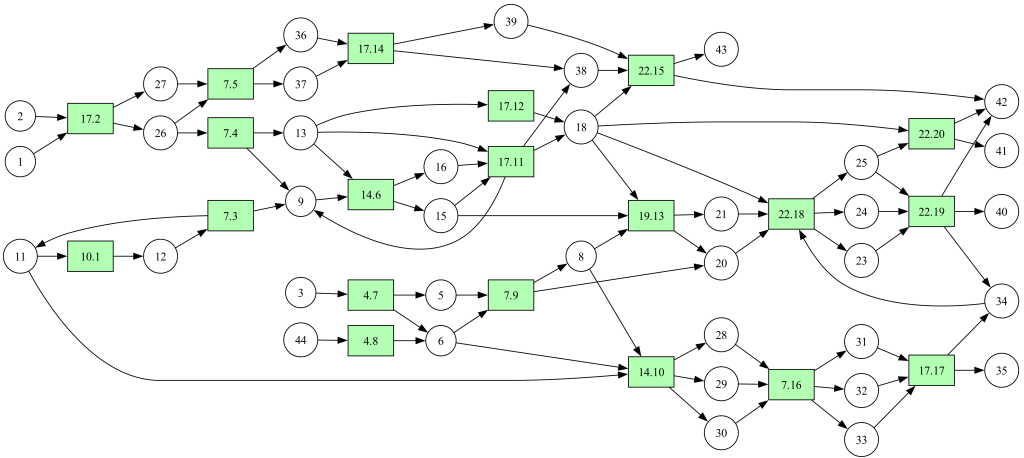
Security Model

- *Withdraw Funds for each Proposal:* The *WithdrawProposalFund()* function follows the *Checks-Effects-Interactions* pattern, where the recipient is responsible for claiming their funds by initiating a withdrawal transaction.

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

- *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.
- *Unbounded Loops Vulnerability:* Unbounded loops in Solidity can lead to denial-of-service attacks and run out of gas issues. For that reason, we have only used loops with defined endpoint

Example of a general AND/OR graph of software executions



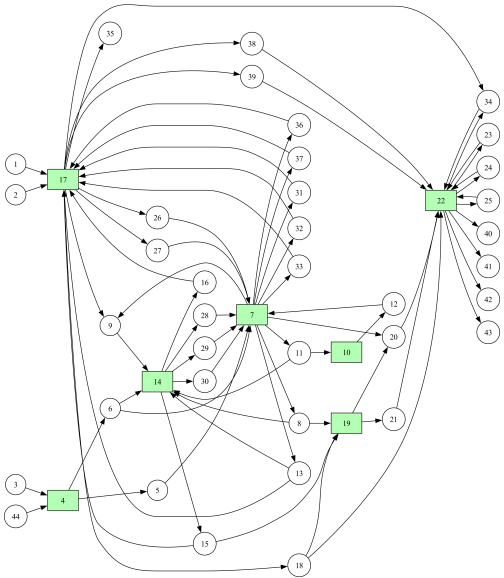
The hash of the data file and software with a specific version is stored in the same way as the unique token ID; hence, the software's token ID is followed by the execution index after the dot.

Analysis Algorithms

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 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

Merging same software



DAGify

Construct a DAG that shows which software initially generated the 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

(a)

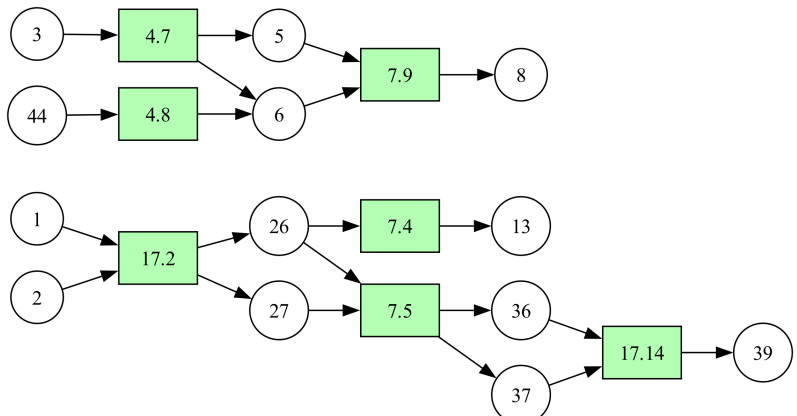
Exec Order	$s_n^{v,e}$	Files created
11	17.11	18, 38
12	17.12	none
13	19.13	21
14	17.14	39
15	22.15	42, 43
16	7.16	31, 32, 33
17	17.17	34, 35
18	22.18	23, 24, 25
19	22.19	40
20	22.20	41

(b)

MaxKnockedDown

Output nodes that produces the maximum number of the knocked down nodes.

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



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.

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.

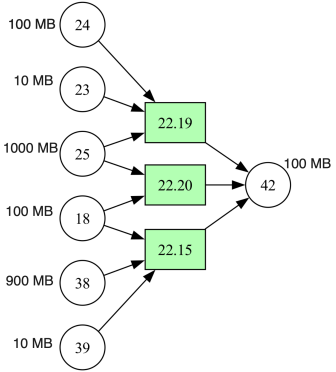


Figure: Software executions to generate node 42. The size of each data node is indicated next to it.

Table of Contents

Discussion and Future Roadmap

- Development of an ERC20 token on a separate smart contract.
- Deployment of eBlocBroker smart contract on the Polygon main networks after making stress testing on Bloxberg.
- 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.
- The RISC Zero [14] 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.
- Larger clusters will be used for testing workflow submissions.
- An accessible web interface will be created using Web3.js.

Publications

- 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.
- Alimoğlu, A. and C. Özturan, "An autonomous blockchain-based workflow execution broker for e-science", Cluster Computing, May 2024.
- 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. 471476, IEEE Computer Society, 2017.
  - An extended journal version has been submitted and will be revised and resubmitted in the future.
- Our implementations are available online as open source with permissive licenses;
  - `https://github.com/ebloc/AutonomousSoftwareOrg`.
  - `https://github.com/ebloc/ebloc-broker`

Any Questions?

THANK YOU

All animated images in this presentation are generated by DALL-E.

REFERENCES

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, and H. Simon, "The nas parallel benchmarks," 1991.

[3] G. Kara 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.

[4] H. Topcuoglu, 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, Mar. 2002.

[5] "Topological generations," accessed on May 4, 2024. [Online]. Available: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.dag.topological_generations.html

[6] "Networkx," accessed on May 4, 2024. [Online]. Available: https://networkx.org

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

[8] T. D. Braun, 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, Jun. 2001.

[9] M. Rocklin, D. Nikel-Shepherd, and C. Rodrigues, "Heterogeneous earliest finish time," accessed on May 4, 2024. [Online]. Available: https://github.com/mrocklin/heft

[10] "Artificial intelligence act," accessed on May 4, 2024. [Online]. Available: https://www.europarl.europa.eu/RegData/etudes/BRIE/2021/698792/EPRS_BRI(2021)698792_EN.pdf

[11] "Drips," accessed on May 4, 2024. [Online]. Available: https://docs.drips.network

[12] R. Hasan, S. Purawat, C. Olschanowsky, and I. Altintas, "Preserving file provenance using principles of blockchain to ensure scientific reproducibility," in *2023 IEEE 19th International Conference on e-Science (e-Science)*, 2023, pp. 1–7.

REFERENCES

[13] "Bloxberg researchcertificate," Mar. 2024. [Online]. Available: https://blockexplorer.bloxberg.org/address/0x3fb704dfDB72Fc06860D9F09124C30919488f13C/contracts#address-tabs

[14] "The general purpose zero-knowledge vm," accessed on May 4, 2024. [Online]. Available: https://www.risczero.com