Preprint Cambridge Centre for Computational Chemical Engineering ISSN 1473 – 4273

A Smart Contract-based agent marketplace for the J-Park Simulator – a Knowledge Graph for the process industry

Xiaochi Zhou[1], Mei Qi Lim[1], Markus Kraft[1,2,3]

released: 17 September 2019

[1] CARES
Cambridge Centre for Advanced Research and Education in Singapore,
1 Create Way,
CREATE Tower, #05-05,
Singapore, 138602

[2] Department of Chemical Engineering and Biotechnology
University of Cambridge
New Museums Site
Pembroke Street
Cambridge, CB2 3RA
United Kingdom
E-mail: mk306@cam.ac.uk

[3] Nanyang Technological University,
School of Chemical and Biomedical Engineering,
62 Nanyang Drive,
Singapore,
637459

Preprint No. 240

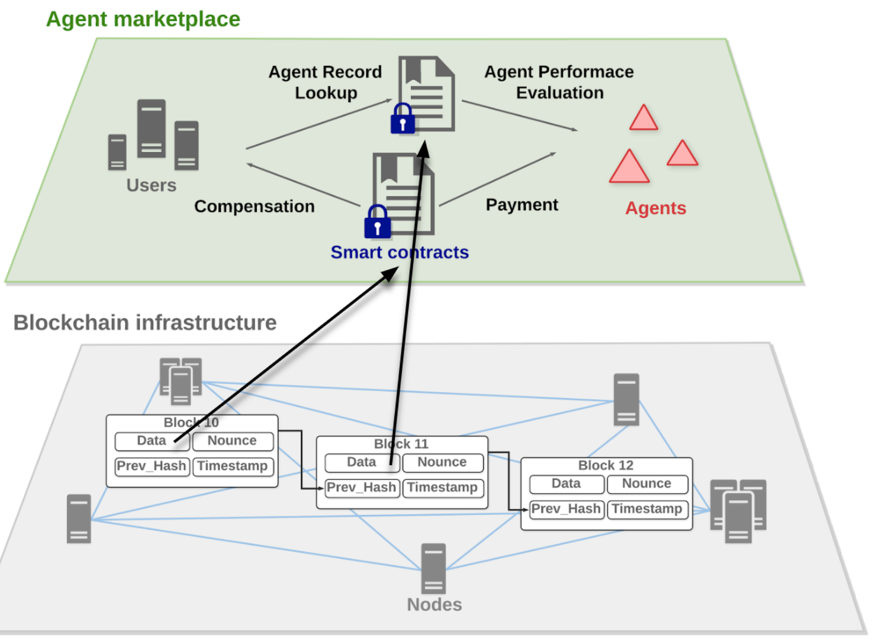
Keywords: Blockchain, Smart Contracts, agent marketplace, Knowledge Graph

Edited by
Computational Modelling Group
Department of Chemical Engineering and Biotechnology
University of Cambridge
West Cambridge Site
Philippa Fawcett Drive
Cambridge CB3 0AS
United Kingdom

Fax:	+ 44 (0)1223 334796
E-Mail:	c4e@cam.ac.uk
World Wide Web:	`https://como.ceb.cam.ac.uk/`

Abstract

The chemical industry is increasingly relying on agents for data acquisition, optimization, and simulation. In order to enable efficient management of agents, Knowledge Graphs (KG) together with agent composition frameworks are therefore applied. However, a method to assess the reliability of agents for such systems is absent. Therefore, this paper proposes a Smart Contract-based agent marketplace for composition frameworks to estimate the reliability of agents. In this agent marketplace, we improved the feedback-based reputation system by leveraging Smart Contracts to eliminate fraudulent ratings and to enable automation. The marketplace incorporates a rating-dependent payment mechanism as well, to further enhance the trust. The paper also illustrates how this marketplace is integrated into the J-Park Simulator (JPS) agent composition framework for the automated agent selection and transaction.



Highlights

- A novel decentralized and automated agent marketplace is built on top of blockchain-based Smart Contracts.
- The agent marketplace is integrated into an agent composition framework, to support automated agent selection and payment.

1 Introduction

The progress in information technology changes the field of chemical engineering at an increasing pace. For example, the number of web services or agents that provide functions relevant to chemical engineering over the internet or intranet is increasing. They are used for data acquisition, simulation, and optimization. For instance, the two thermodynamic databases, NIST chemistry webBook [21] and Mol-Instincts [2] provide agents that allow retrieval of thermodynamic data.

However, due to the heterogeneity and the increasing number of agents, it is challenging to enable their efficient discovery and coordination. A solution to this problem can be achieved by embedding agents into a Knowledge Graph (KG) which enables automated management of agents [11, 31]. Knowledge Graphs are sets of inter-connected classes, relations, and instances that are semantically described, *i.e.,* each distinct information is denoted by an unique Uniform Resource Identifier (URI)[1]. Due to the unique mapping from URIs to classes or instances, semantic descriptions are explicit and machine-readable. A collection of the semantic concepts providing vocabularies to build KGs is called Ontology. Within the KGs, information such as the I/O signatures and prices of agents are described on top of agent Ontologies such as OntoAgent [31] so that functions, request formats, and other properties of agents can be interpreted by computer programs. Thus, systems interacting with agents automatically, such as agent composition frameworks and agent registries, are enabled in KGs.

The J-Park Simulator (JPS) [11, 31] is an example of such a KG and serves as a research platform to explore how internet technologies can be used to achieve interoperability between different domains. It contains the semantic descriptions, based on OntoAgent, of a set of agents across multiple domains. On top of the agent descriptions, an agent composition framework has been implemented for the automated creation of composite agents for complex tasks consisting of interconnected sub-tasks. A composite agent selects atomic agents from the KG and put into a sequence based on their I/O signature (*i.e.,* the data type of their inputs and outputs). After creating a composite agent, the composition framework executes the composite agents.

Any composition framework faces the problem of assessing the atomic agents' performance and reliability. In other words, the framework needs to know whether an agent can be trusted. For example, the consistency and the comprehensiveness of thermochemical data for chemical species, provided by a thermodynamic database agent, significantly affects the accuracy, predictive performance and quality of the models that utilize this data [4]. In addition, agent performance and reliability are important selection criterion within the agent composition framework when there are multiple functionally identical agents available. Therefore, it is critical to provide credible information of the performance and reliability of the agents.

A most traditional way to endorse the trustworthiness of an entity (*e.g.* an agent) is to establish an authority to qualify and monitor the agents so that their quality and integrity is guaranteed. However, such a solution can not handle the vast number of agents in KGs because investigation and examination are time-consuming. Another solution is the

[1] https://www.w3.org/Addressing/URL/uri-spec.html

contract-based solution, where contracts define the rights and duties of the parties in advance, and the violation of the terms will lead to consequences. However, the enforcement of the contracts is challenging due to the large scale of KG. Besides the scalability problem, the two aforementioned solutions rely on human-intervention, which is too slow to cope with the highly automated nature of the agent composition framework.

Among all the solutions to access performance and reliability, feedback-based reputation systems are considered as the most cost-effective and scalable one. A typical reputation system is administrated by a single party (*e.g.* a hotel review website). It collects users' ratings for a vendor or product after a transaction and calculates a cumulative rating upon all the historical ratings, forming a quantified reputation score. Such a score allows the users to assess the reliability of the vendor or the quality of the product before the purchase, and hence establishes the confidence of the users for this purchase [14, 22]. The feedback-based reputation system is scalable and cheap, as it does not rely on designated institutes to evaluate the quality of products. As a result, it is adopted by virtually all electronic commerce platforms. Meanwhile, barely any human intervention is required to manage such a system [14, 22] so that the system can operate automatically. Such a solution is scalable and compatible with automated systems.

However, the current implementation suffers from persistent problems of rating frauds, from both users and the administrators [13, 16, 24], which discredit the reputation system. Firstly, the higher ratings lead to more profit [29] so the users (*e.g.,* vendors on e-commerce platforms and agent providers in KGs) may insert unjustly high ratings to promote a product or service and inject unfairly low ratings to demote competitors. Therefore, the ratings will fail to reflect the quality of the products. One of the existing solutions is to analyze the ratings and filter out the malicious ones. Many filters have been built [17, 26, 28]. However, when the profit for creating a fraudulent rating surpasses its cost, the well-funded dishonest rating could be deliberately masked as regular ones, making it harder to distinguish malicious ratings. For the case of KGs, the commercial agents involved are usually of high value. For example, in our use case the ADMS agent is built on top of the Atmospheric Dispersion Modeling System (ADMS)[2], a proprietary software with a substantial cost per license. Consequently, the profit per call will be high too. Therefore, if a feedback-based reputation system is implemented for the the composition framework, it is even more likely to encounter fraudulent ratings from agent providers. Secondly, the mainstream designs of reputation systems are centralized, which means the functions of cumulative score calculation and score look up, and score storage are controlled by designated administrators, which could also behave dishonestly. For example, an administrator may take bribery from an agent provider to tamper the scores of their agent. Clearly, the countermeasures against frauds from agent providers are no longer applicable.

With the advent of the blockchain technology, some decentralized designs are proposed to address this problem. A blockchain, which will be further illustrated in Section 2.2, provides tamper-proof and decentralized storage of data. Since a blockchain is managed without a central authority, several blockchain-based decentralized reputation systems have been created. Dennis and Owen [9] proposed a blockchain-based P2P reputation system for file transaction. Carboni [7] designed an incentive-based feedback reputation

[2] `https://www.cerc.co.uk/environmental-software/prices.php`

model on top of the Bitcoin blockchain. These designs successfully established reputation systems without centralized control over them while guaranteeing the integrity of reputation records. However, the mentioned models could not implement functions such as score calculation and service searching without a centralized third party, as a blockchain could only provide decentralized management of data but not the implementation of functions. For example, the calculation of the cumulative score is exposed to manipulation by this third party. Consequently, the scores are still vulnerable to dishonest behaviour.

In order to implement a score calculation in decentralized ways, the blockchain-based Smart Contract, which will also be explained in details in Section 2.2, is proposed to provide decentralized control over the implementation of functions [5]. Calvaresi et al.[6] proposed a reputation system in which the cumulative performance score of an agent is automatically calculated and managed by the Smart Contracts and stored in the blockchain. This solution takes one step further; it not only guarantees that the performance records are tamper-proof but also secures the integrity of the calculation of the cumulative score. Klems et al.[15] also provided a Smart Contract-based solution for a decentralized service marketplace, which integrates functions such as match-making, transaction settlement, and dispute resolution. However, both solutions rely on the feedback from users. Consequently, although the Smart Contract-based solution prevents the risk of frauds from administrators, it still exposes the reputation system to the risk of rating frauds from the users.

In conclusion, to the best of our knowledge, a mechanism to provide credible agent performance record that could cope with the highly automated nature of an agent composition framework and the large scale KGs as well as resistant against fraudulent ratings from both users and administrators is absent.

The purpose of this paper is:

- To present a novel design of a Smart Contract-based feedback reputation system that allows an agent composition frameworks in a KG to assess the reliability of agents while ensuring that the design is scalable, compatible with highly automated systems, and invulnerable to rating frauds from neither users nor administrators.
- To demonstrate a use case which integrates the agent marketplace with the JPS agent composition framework, to facilitate its agent selection.

The remaining parts are structured as follows. Section 4 explains the technologies leveraged by the agent marketplace. Section 3 illustrates the design and implementation of the Smart Contract-based agent marketplace in detail. Section 4 demonstrates how we integrated this agent marketplace with the JPS agent composition framework. Section 6 discusses the limitations of the current work while Section 5 provides plans for the future improvement. Section 7 outlines the conclusions for this paper.

2 Background

In this section we provide some information on the J-Park Simulator (JPS) and Etheruem Smart Contracts. The JPS will provide the environment in which we built our use case and demonstrate the effectiveness of our solution. Etheruem Smart Contracts is the specific technology we have chosen to develop our solution and the section below summarises some key features useful to understand the technical aspects of the paper.

2.1 J-Park Simulator

Large cross-domain systems such as Industrial Symbioses, chemical plants, and cities are constituted by components such as power generators, storage tanks, and buildings, which are from diverse domains. In order to achieve complex tasks including running simulation and optimization and coordination of multiple components, the relevant data, knowledge and models must be integrated. However, the communication friction due to the heterogeneous conventions across domains hinders such an integration. Therefore, JPS is developed to provide a data management common ground for those components and enable semantic interoperability, so that cross-domain integration could be enabled.

For example, on top of JPS, Zhou et al. [30] proposed a methodology to use the ontologies for the modelling and management of eco-industrial parks (EIPs) and their components. They also applied such a methodology to increase the efficiency of intra-plant waste heat utilization. The waste heat utilization between chemical plants on Jurong island in Singapore is hindered due to the communication friction between them caused by the heterogeneous terminologies. With the explicitness of ontology descriptions, intra-plant waste heat utilization opportunities could be better found. In addition, Devanand et al. [10] gave an example of using the JPS cross-domain KG to access financial and geographical information of potential sites for the optimal placement of Small Modular Reactor (SMR). Such an approach is enabled by the KG's capability to effectively incorporate cross-domain data and provide convenient access to them.

To constitute the JPS cross-domain KG, a set of ontologies are developed or incorporated, which contains structured and connected knowledge and data that are represented semantically, *i.e.* concepts and individuals are denoted by Uniform Resource Identifiers (URIs). As each URI uniquely points to a distinct concept or individual (*e.g. dbr:Cambridge*[3] and *dbr:Cambridge,_Massachusetts*[4] each denotes "Cambridge" in the UK and the US), the data representation is explicit and unambiguous. The explicitness also makes the data and knowledge machine-readable. A collection of the explicitly declared concepts and individuals are referred to as an ontology [25]. Further, a collection of inter-connected ontologies integrated for a certain purpose is considered as a KG according to our understanding.

As shown by Figure 1, the JPS KG consists of the domain ontologies and the agent ontology. The domain ontologies are utilized to model knowledge, data, and entities in a wide range of fields. For instance, OntoCAPE [19] is integrated to describe concepts and individuals that are related to chemical process engineering. Starting from OntoCAPE, OntoEIP [30] is developed to model eco-industrial parks. Moreover, OntoCityGML, OntoKin, and OntoEngine [12] are included to cover the field of city modelling, chemical kinetics, and internal combustion engines. Those ontologies are used collectively to pro-

[3]http://dbpedia.org/resource/Cambridge
[4]http://dbpedia.org/page/Cambridge,_Massachusetts

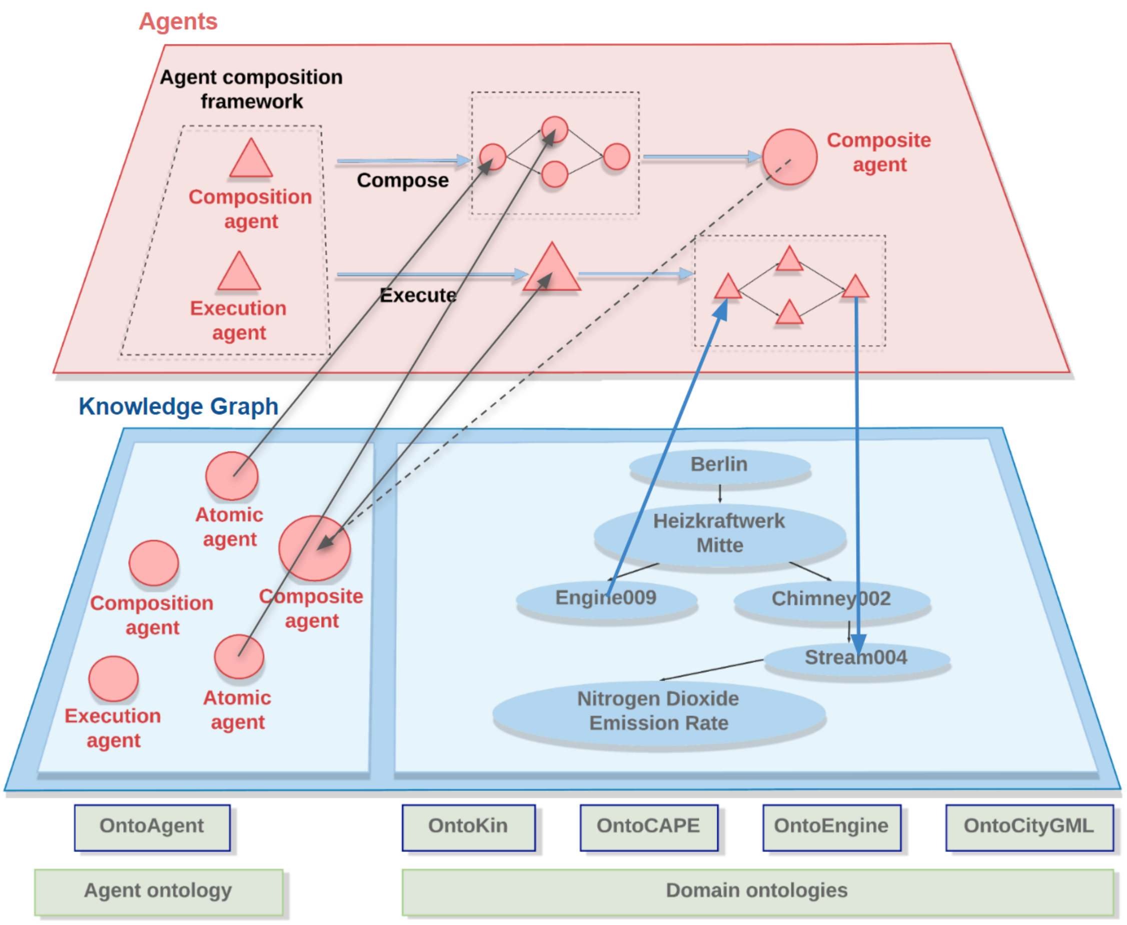


Figure 1: *The JPS KG and agents: a) the JPS KG (blue layer) contains both the agent ontology (left) and the domain ontologies (right). b) each ontology includes the terminology (green boxes) and the instances (pink nodes for agent instances and blue nodes for domain ontologies). c) the agent layer (pink layer) includes an agent composition framework, which creates and executes composite agents. d) agents in action are triangles. The black solid arrows represent the mapping between the same agents on the two layers and the blue arrows denote the data stream between the agents and the domain ontologies.*

vide knowledge and data for cross-domain simulation and optimization.

However, those ontologies must be continuously managed and updated due to the dynamical nature of the real-world components such as cities, eco-industrial parks and industrial cooperation systems. Therefore, agents for data acquisition, optimization, simulation, *etc.* are implemented to update and maintain the KG. To enable semantic access of those agents for other components on the JPS platform, semantic individuals of the agents are described by the OntoAgent [31] ontology. For an agent instance, OntoAgent describes its I/O signature by assigning domain ontology concepts to its inputs and outputs, so that machines can interpret the agent's function, discover agents according to I/O requirements, and also make I/O based matchmaking between agents.

On top of the semantic description, an agent composition framework is implemented in the JPS KG. Such a framework can automatically discover, select, and arrange agents from the KG in order to generate a composite agent to fulfill complex tasks. Given a user defined I/O requirement for the composite agent, the framework will iteratively discover and match agents based on their I/O to fill the gap between the given I/O requirement. For example, the upper part of Figure 7 demonstrate the structure of a composite agent that takes reaction mechanism and region as inputs and produces air dispersion as the output. The atomic agents, which provide the intermediate steps in this complex task, are connected according to their I/O to constitute a composite agent that simulates air dispersion within a particular area.

However, in a composite agent generated by the framework, it is possible that there are multiple agents providing the same functions. Therefore, an optimization module is also implemented in the framework. The optimization module implemented for the framework select out the optimal agent among functionally-identical ones by ranking the agents with regard to their performance scores. However, due to the lack of scalable and secure approach to provide reference for agent selection, those functionally identical agents are assigned with arbitrary performance scores before we successfully implemented the outcome of this paper.

In addition, the framework comes with an execution agent, which is able to automatically execute the composite agent after the optimization process. It executes the atomic agents by constructing and sending HTTP request according to the semantically described grounding information of the agents. By feeding the outputs of an upstream agent to the connected downstream agents, the execution agent executes all the atomic agents in sequence to produce the output(s) of the composite agent.

2.2 Blockchains, Etheruem Smart Contracts and Oraclize

In this paper, we implement the Smart Contract-based agent marketplace on top of the Ethereum blockchain to address the problem of supplying a credible agent performance record in the JPS. Ethereum [5, 27] is a blockchain that designated to support the deployment of Smart Contract-based decentralized applications (DAPPs). The same as other mainstream blockchains, the Ethereum blockchain is a chain of data blocks shared on a P2P network. Each of the blocks contains the hashed transaction record or other general data within a specified period as well as the hashed previous block, namely the previous hash. As a result, if the data within a block is modified, its hash will then fail to match to the subsequent block's previous hash. Therefore, by iteratively verifying whether a block's previous hash accord with the prior block, which only takes minimal computation power, a user on the blockchain could verify the integrity of all the record stored on a blockchain. Consequently, to modify the data on a specific block while not failing the integrity check, one must re-calculate the previous hash of all the subsequent blocks.

Besides, Ethereum and many other blockchains implemented proof-of-work systems to increase the difficulty to re-calculate hashes of the whole blockchain. The proof-of-work systems increase the amount of computation required to create a new block so that an enormous amount of computational power or time is necessary to re-calculate the blockchain. As a result, the historical records in the blockchain are secured.

Also, each node on the P2P blockchain network keeps a full copy of the blockchain, and a new block can only be appended to the blockchain if the majority of the nodes have verified the block. Therefore, the authority of validating new records can not be easily seized by a malicious party.

Thus, by its design, the data stored on a blockchain is tamper-proof, and a blockchain could be implemented without a centralized authority. Figure 3 illustrates the structure of a blockchain. The tamper-proof and decentralized data storage of the blockchains enabled a series of applications in the chemical industry. For instance, Sikorski et al. [23] proposed a machine-to-machine electricity market in the context of the chemical industry built on the MultiChain blockchain.

However, blockchains only provide the secure and decentralized storage of data. In order to enable more complex blockchain-based applications, blockchains must also support secured and decentralized code implementation. Therefore, on top of these features of proof-of-work blockchains, some blockchains start to support blockchain-based Smart Contract. The major ones are Bitcoin [20] and Ethereum [18]. However, the Bitcoin blockchain only provides a limited set of functionalities for their Smart Contracts [8]. Therefore, we chose to implement the agent marketplace on top of the Ethereum blockchain, which is of higher versatility. On the Ethereum blockchain, Smart Contracts are bytecodes that are published on the blockchain via transactions; therefore, same as other data on the blockchain, the bytecodes are inherently tamper-proof. Currently, Solidity[5] is the main language used to develop Ethereum Smart Contracts. The Ethereum blockchain inludes Ethereum Virtual Machines (EVM). The EVMs, which are installed on each node locally, assures that for the same code executed, the same result is produced. Therefore, with EVMs and the tamper-proof nature of the Smart Contracts code on the blockchain, carrying out of the functions of the Smart Contract can not be intervened. As a result, Smart Contracts serve as trustworthy and autonomous nodes to enforce activities on the blockchain.

Identical to other nodes on the blockchain, Smart Contracts are assigned with blockchain addresses (in the form of a hexadecimal number) so that they can receive transactions from other nodes. Smart Contracts can possess Ether, the cryptocurrency on the Ethereum blockchain, and make transactions to other nodes too. This a feature allows Ethereum Smart Contracts to carry out tasks that involve financial transactions.

[5]https://solidity.readthedocs.io/en/v0.5.11/

Another feature of the Ethereum Smart Contract is that they can store data. Each variable declared in the code of the Smart Contract is assigned with an address on the blockchain. By making a transaction, the Smart Contract updates the variable value, and the nodes with permissions can read the value of the variable locally through the Smart Contract. This feature allows the Smart Contracts to manage a database that stores general data. For example, the performance scores of agents.

However, compared with traditional applications, the Smart Contract-based DAPPs have some restrictions due to the blockchain infrastructure. For instance, Smart Contracts are not allowed to make direct HTTP requests to the Internet, to guarantee the predictability of activities on the blockchain. One of the criteria for the credibility of a blockchain is that all the change of states on it can be precisely reproduced based on the transactions records. However, an HTTP request may return different results with the same input. As a result, the Smart Contracts can not receive any data, including the result of an HTTP call, other than explicit transaction from other nodes on the blockchain.

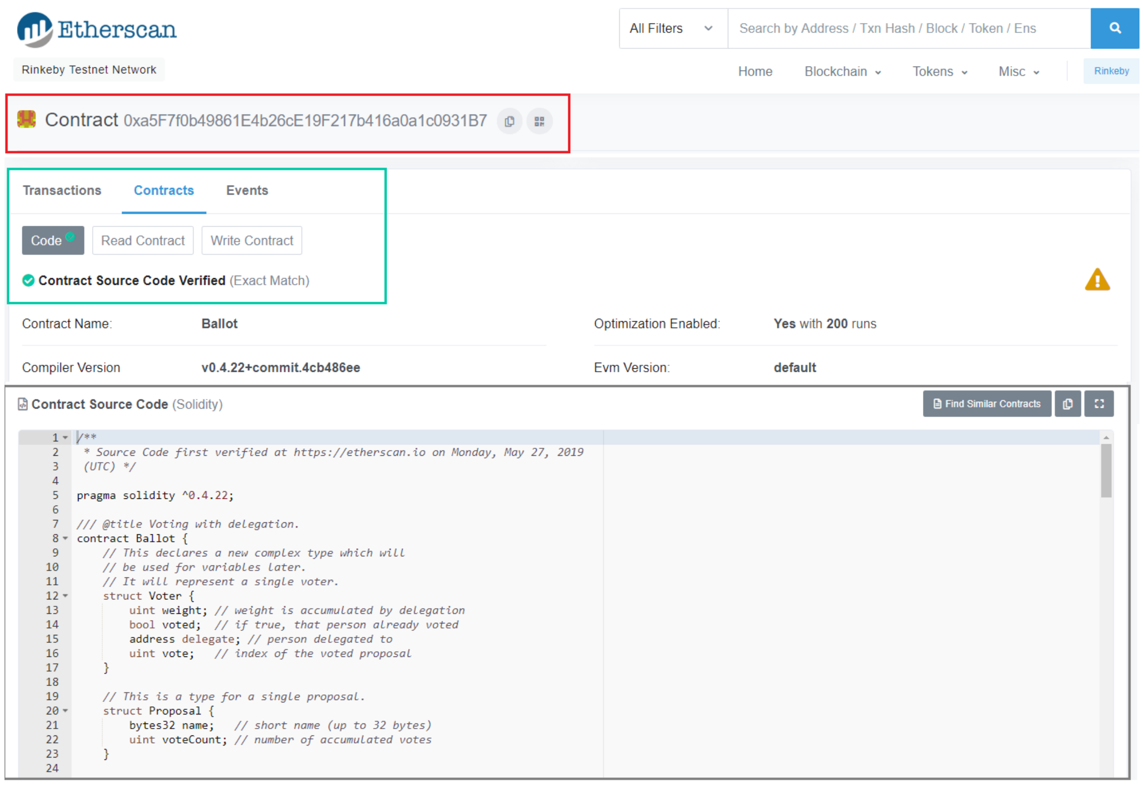
Therefore, the Oraclize [3] service is used to make delegated HTTP requests for the Smart Contracts. The Oraclize service sets up Smart Contracts to receive the call for making HTTP requests, from other Smart Contracts. The Oraclize Smart Contract will then pass the requests through transactions to the Oraclize servers, which are also nodes on the blockchain, to make the HTTP requests off the blockchain. After the result of the HTTP request is returned, the Oraclize server will deliver it to the Oraclize Smart Contract. Finally, the Oraclize Smart Contract will return the result to the calling Smart Contract. As a result, the Smart Contract could make HTTP request indirectly via the Oraclize service.

In addition to the features of Smart Contracts themselves, some tools are implemented to test and inspect Smart Contracts. Besides the Ethereum main blockchain network, Ethereum also provided a test network named Rinkeby[6]. On the main network, Ethers (the cryptocurrency used on the Ethereum) are acquired from mining and purchasing; therefore, the Ethers are of real-world market value. To avoid financial losses while testing Smart Contracts, we deploy and hence test our Smart Contract on the Rinkeby test network. On the Rinkeby network, Ethers are arbitrarily assigned to accounts and therefore of no real-world value.

Moreover, websites such as Etherscan are tools for inspecting Smart Contracts. As shown in Figure 2, through the Etherscan website, one can investigate the address and the bytecode of the Smart Contract. If the developer has uploaded the source code of the Smart Contract, it also verifies whether the source code accord with the bytecode published.

We have now introduced all elements to present our solution to the above defined problem of a secure agent scoring system within our development platform JPS.

[6]`https://www.rinkeby.io`



Contract 0xa5F7f0b49861E4b26cE19F217b416a0a1c0931B7

Contract Source Code Verified (Exact Match)

Contract Name: **Ballot**

Optimization Enabled: **Yes** with **200** runs

Compiler Version **v0.4.22+commit.4cb486ee**

Evm Version: **default**

Contract Source Code (Solidity)

```
/**
 * Source Code first verified at https://etherscan.io on Monday, May 27, 2019
 (UTC) */

pragma solidity ^0.4.22;

/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

```

Figure 2: *The screenshot of a Smart Contract information page: a) in the red box is the address of the contract, which is a hexadecimal number. Knowing this address, other nodes on the blockchain can make transactions to the contract or call its functions. b) within the green box, it shows that this Smart Contract is verified, in the sense that its compiled bytecode published on the blockchain accord with the source code. c) in the gray box is the source code of this contract submitted by the author, which is written in Solidity language.*

3 Agent marketplace

In order to address the aforementioned fraudulent rating problem of the current feedback-based reputation systems, we augment Smart Contracts, to provide quantified data with the aim to, firstly, evaluate the trustworthiness of the agent, secondly, to select the optimal agent among functionally identical ones and, thirdly, to introduce a payment mechanism making financial transaction between participants after invocation of the chosen agent. The two systems then constitute an agent marketplace that provides most of the core functions for agent selection, evaluation, and employment as shown in Figure 3. The Smart Contracts making up this agent marketplace is developed in the Solidity[7] language and published on the Ethereum Rinkeby test network.

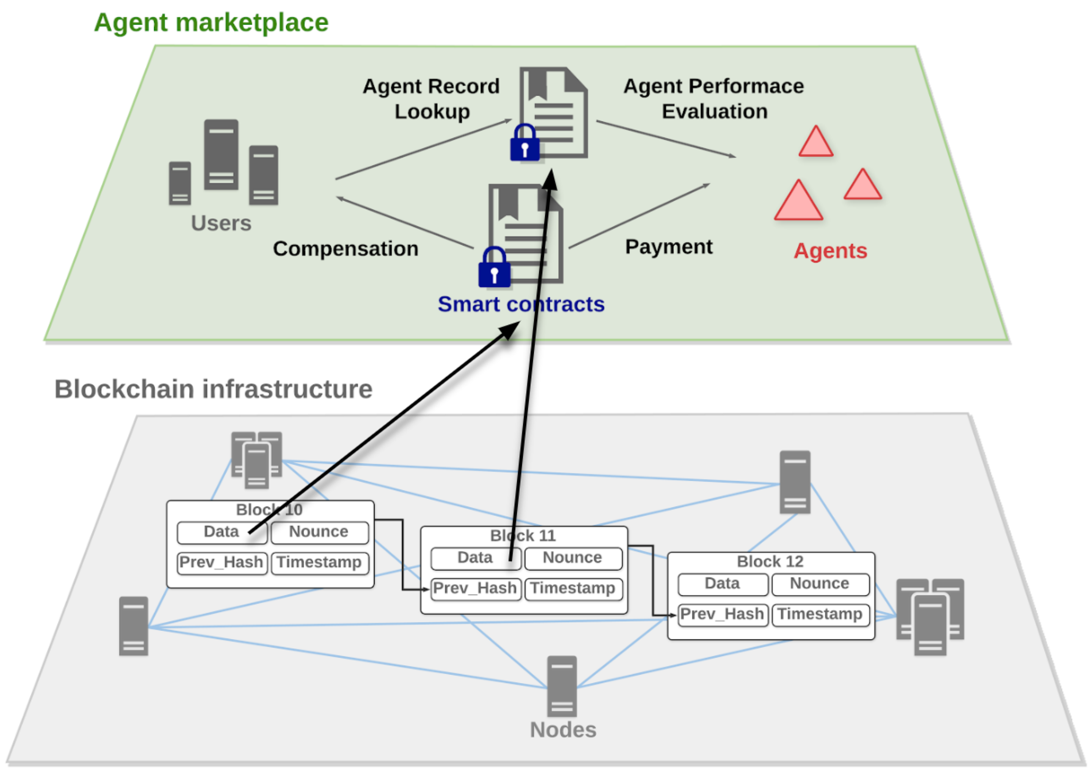


Figure 3: *The Smart Contract-based agent marketplace: a) the agent marketplace (the green layer) is established with Smart Contracts on top of a decentralized blockchain infrastructure (the gray layer), which is updated and validated by a number of connected nodes collectively. b) The Smart Contracts are compiled bytecode published on a blockchain (black arrows). In this agent marketplace, functions including transactions to users and agents, agent record lookup, and agent performance evaluation are implemented with Smart Contracts.*

[7]https://solidity.readthedocs.io/en/v0.4.24/

3.1 Reputation system

As mentioned above, building the reputation system on top of users' feedback may lead to frauds. However, the features of the tamper-proof code and the decentralized execution of the blockchain-based Smart Contracts enabled a solution to this problem. With these two features, Smart Contracts are used to call agents, to evaluate agent performance, and to manage reputation records independently and hence to prevent fraudulent feedback.

As illustrated in Figure 4, to employ an agent, the user (the agent composition framework) needs to call the ***invoke()*** function, providing the Etheruem address of the agent (a hexadecimal number that points to the agent provider's Ethereum account) and the input data for the request. Subsequently, the Smart Contract will check the user's deposit balance with the ***check_deposit()*** function. If the balance is sufficient, the Smart Contract will make an HTTP call through function ***_call()***, which will search for the agent URL according to the given Ethereum address, compose an HTTP request, and finally delegate the request to the Oraclize service. As introduced in the Background section, the Oraclize service allows Ethereum Smart Contracts to make HTTP requests for the agents through it. When the HTTP request for the agent returns the results, the reputation Smart Contract will receive it via the ***__callback()*** function, which will then return the result to the user.

Simultaneously, the ***evaluate_performance()*** function will be triggered to evaluate the performance of this agent invocation based on the result received. This function will then calculate the performance score on top of a domain-specific agent evaluation matrix, which varies between different reputation Smart Contracts for different agents, and the returned result. For example, the weather agents are evaluated based on their comprehensiveness of data. In the agent composition case to create a composite agent to simulate the pollutant dispersion in an urban area, weather agents are expected to provide a series of weather data including the wind direction, wind speed, temperature, precipitation, *etc*. Since the more factors are taken into consideration the more precise the simulation is, whether a weather agent provides a comprehensive set of weather data is the most critical factor in its performance evaluation matrix. In this case, the ***evaluate_performance()*** function goes through the semantically structured weather data and counts the URIs of the data entries such as wo:hasPrecipitation[8]. The number of the data entries involved would be the performance score in this simple example. Such an automated and independent performance evaluation is only enabled when the agents that are parts of the KG. As mentioned above, the agents within the KG share the common ground for data exchange and are inter-operable. As a result, agents from different sources do not have the problem of heterogeneous I/O format. Therefore, the Smart Contracts to evaluate them could evaluate them based on their outputs by the same method as shown in the function ***evaluate_performance()***.

Subsequently, this function updates the reputation records of this agent and triggers the payment mechanism, which the next subsection will introduce in details.

[8]https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/WeatherOntology.owl#hasPrecipitation

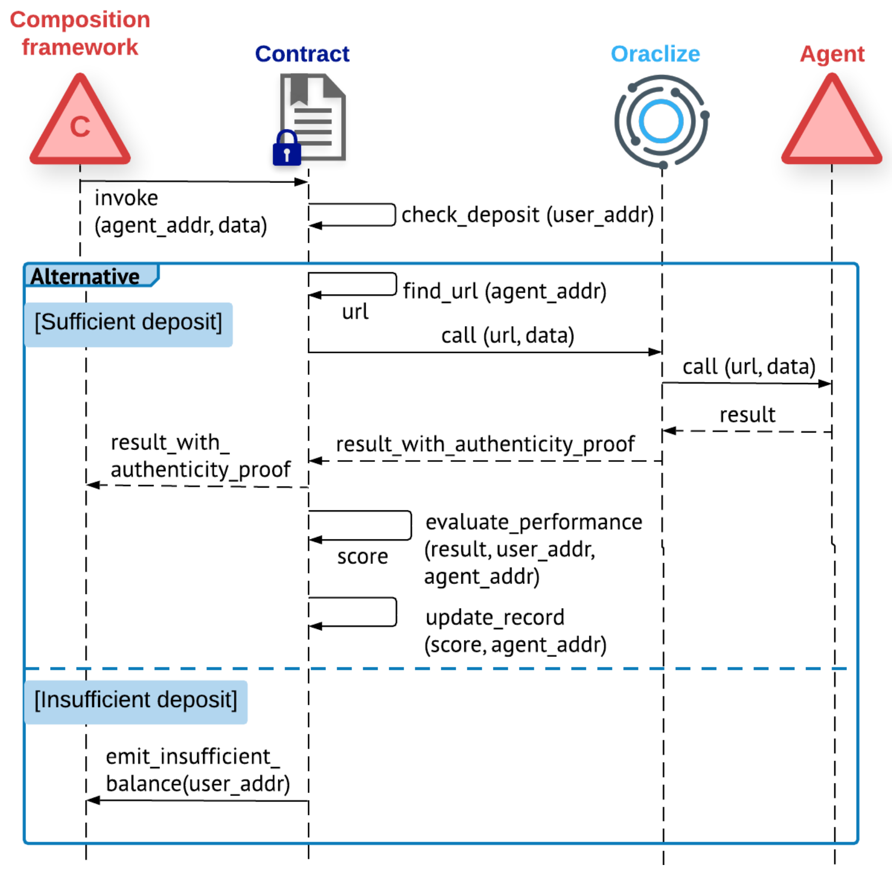

Composition framework
C
Contract
Oraclize
Agent
invoke (agent_addr, data)
check_deposit (user_addr)
Alternative
[Sufficient deposit]
find_url (agent_addr)
url
call (url, data)
call (url, data)
result
result_with_authenticity_proof
result_with_authenticity_proof
evaluate_performance (result, user_addr, agent_addr)
score
update_record (score, agent_addr)
[Insufficient deposit]
emit_insufficient_balance(user_addr)

Figure 4: *UML diagram for the agent reputation system: a) the process of performance evaluation start from the user making a request to invoke an agent, providing the hexadecimal address of the agent (invoke(agent_addr,data)). b) with sufficient deposit, the contract will delegate the invocation of agent to Oraclize service, which returns the result with authenticity proof. Based on the result, the Smart Contract will make an evaluation on the performance and then update the new cumulative score. From this point, the payment mechanism will be triggered, which is illustrated in Figure 5 c) with insufficient deposit, the contract will notify the user.*

3.2 Payment mechanism

The payment mechanism is made after the invocation which is proportional to the performance evaluated and is conducted automatically by the Smart Contracts. Such an implementation is enabled by the feature that the Ethereum Smart Contracts could receive and transfer funds from and to other Ethereum accounts. Meanwhile, such a design enable the payment mechanism to pay the users a compensation instead when the performance is lower than a certain threshold or the agent failed to provide a service. This feature further enhances the user's trust for the agents as they would automatically receive financial compensation for bad agent performance. Figure 5 demonstrates the working flow of the payment mechanism and the Appendix A.2 shows the Solidity source code of the implementation. The payment system is based on a deposit system. Both the agents and

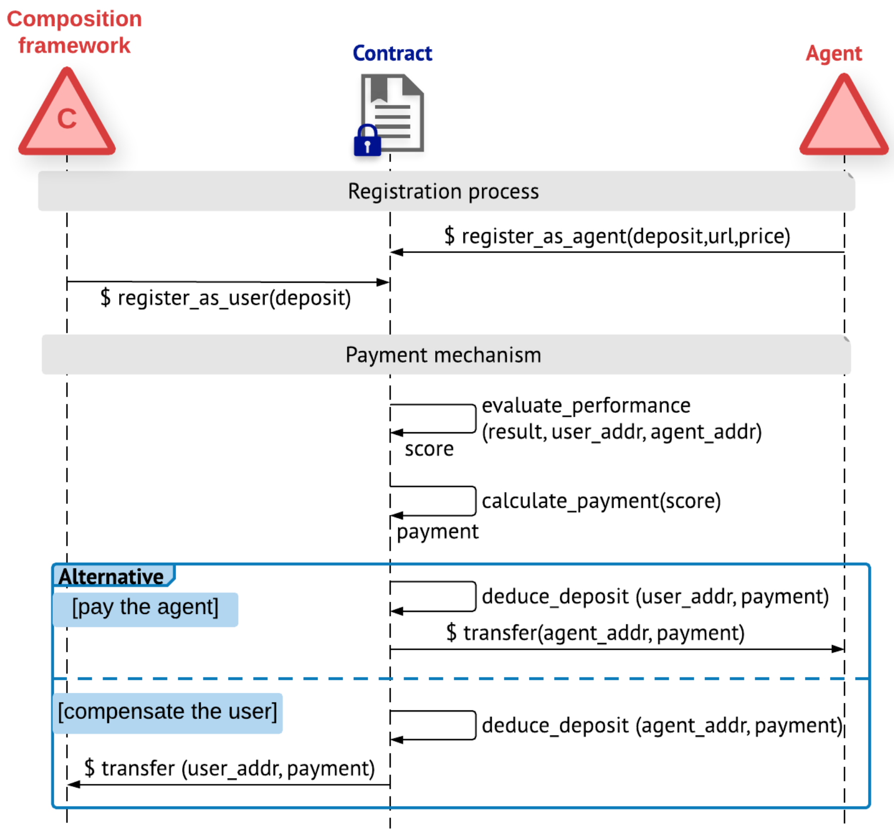

Composition framework
C
Contract
Agent
Registration process
$ register_as_agent(deposit,url,price)
$ register_as_user(deposit)
Payment mechanism
evaluate_performance (result, user_addr, agent_addr)
score
calculate_payment(score)
payment
Alternative
[pay the agent]
deduce_deposit (user_addr, payment)
$ transfer(agent_addr, payment)
[compensate the user]
deduce_deposit (agent_addr, payment)
$ transfer (user_addr, payment)

Figure 5: *UML diagram for registration and payment mechanism: a) to register, the user need to pay a deposit to the contract through the register_as_user(deposit) function. (the $ sign denotes a payable function). b) an agent register with its HTTP URL and its price for one invocation besides deposit. c) the transaction is triggered when the invocation process is finished and the agent performance is evaluated. Based on the score, the contract will calculate the amount of payment or compensation and then conduct the transaction.*

users are required to register and pay a deposit in order to join the agent marketplace. The registration processes are implemented in function ***register_as_agent()*** and function ***register_as_user***. An agent must provide its URL and the price for each invocation to register and pay an amount of deposit defined by the Smart Contract. The address of Ethereum node that registers this agent will be recorded as the Ethereum address of the agent, such an address will serve as the identifier of this agent within the agent marketplace. A user also needs to make a deposit. Subsequently, the Smart Contract will include them in the agents or user list and update the list on the blockchain. These lists are also accessible to other nodes on the blockchain through functions ***get_all_agents_address()*** and ***get_agent_record()*** so that other nodes could lookup the agents reputation records.

Since the Smart Contract can make an unbiased evaluation and can not be manipulated, it can be trusted to control the deposit of users and agents and make the payment by itself through the blockchain's secured financial transaction layer. When a call to the agent and the according evaluation is completed, the Smart Contract will calculate out the amount of payment with the function ***calculate_payment()*** and then make the transfer via the built-in ***transfer()*** function of the Smart Contract.

4 Use case

This use case demonstrates how we implemented the agent marketplace in the JPS to provide credible reference for performance-based agent selection.

As introduced in the background section , the JPS agent composition framework needs to select the optimal agent when there are multiple functionally-identical ones available in the JPS KG. However, before implementing the outcome of this paper, it faces the problem of the absence of credible sources of the agents' performance and reliability. Because of the vast number agents in the JPS KG, the distributed nature of their implementation, the dynamic nature of their performance, and the automated nature of the composition framework, it is impossible to evaluate their performance through an institute or to use contracts to guarantee their performance. Although the feedback-based reputation system offers a scalable solution for the problem, it is not suitable for the JPS agent composition framework as it is vulnerable to fraudulent ratings. A considerable number of JPS agents are high-value simulation/optimization agents, such as ADMS and SRM[9]. As a result, if a feedback-based reputation system is implemented for the JPS agent composition framework, there is a high risk of rating frauds against it.

Therefore, the agent composition framework is connected to the the Smart Contract-based agent marketplace for the access of agent performance records. As shown in Figure 6, the Smart Contract-based agent marketplace will store and manage the performance record of the agents within the JPS KG, provide the agent composition framework the access to these records, and automatically evaluate the performance of agent after its execution under the agent composition framework.

As shown in the Figure 7, in this use case the composite agent required has the inputs reaction mechanism and region and it's output is the type air dispersion[10]. Such a composite agent can be used to evaluate the suitability of proposed locations of installing a new power plant or chemical plant or to assist evacuation planning in case of emergency such as tank leakage. Eight agents are put into the composition result but there is a redundancy when it comes to the agents that provide the weather data in a particular city. The three weather agents built on top of AccuWeather[11], Yahoo Weather[12], and OpenWeatherMap[13] accordingly. The agents wrapping around these web services take the URIs of cities and return the detailed and semantically restructured weather data including the wind velocity, the temperature, the precipitation, *etc.* However, despite of their identical I/O signature, there is a difference of the comprehensiveness of weather data, which is the most critical evaluable factor affecting the quality of weather data and hence the simulation. Therefore, we implemented a Smart Contract in the agent marketplace that

[9] https://cmclinnovations.com/products/srm/

[10] which is temporarily represented by Table as the air dispersion concept is under development.

[11] https://www.accuweather.com/

[12] https://www.yahoo.com/news/weather

[13] https://openweathermap.org/

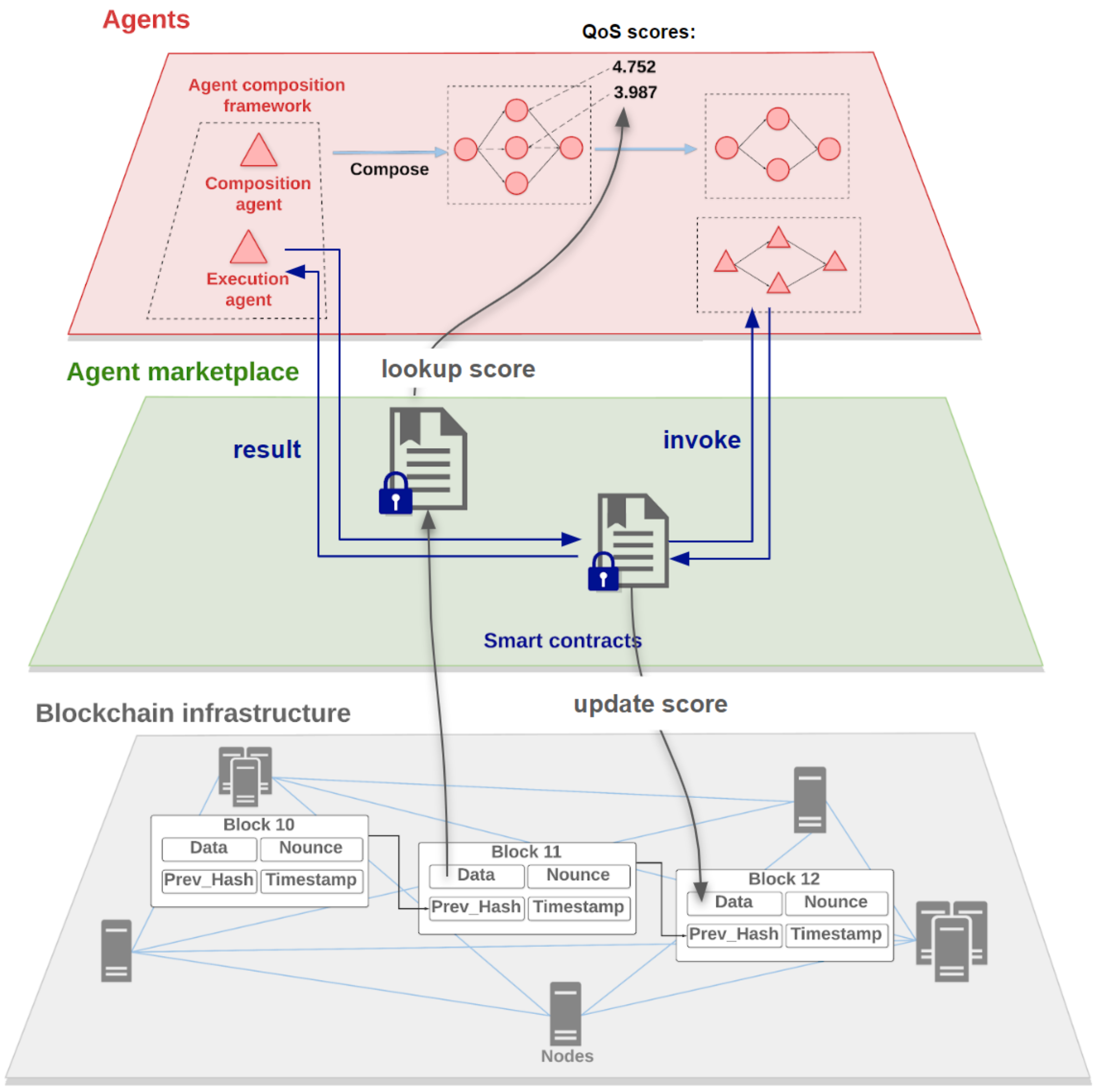
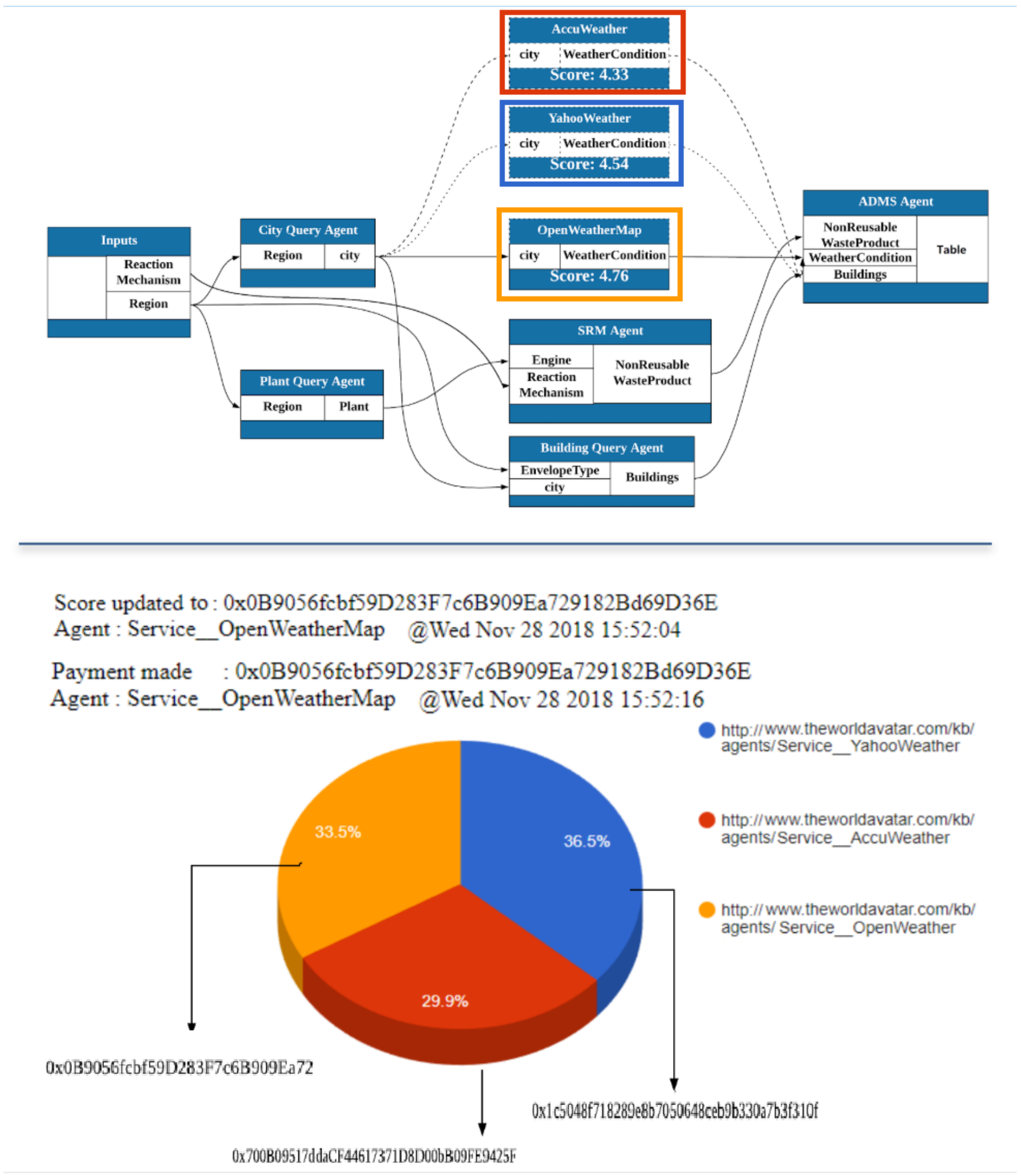


Figure 6: *The integration with the JPS agent composition framework: the agent marketplace is then applied to the JPS project to provide performance score lookup, performance evaluation, and transaction functions. a) the grey arrows represent the Smart Contracts reading QoS scores from or updating them to the blockchain. b) the blue arrows denote the delegated invocation of agents though the Smart Contracts.*

evaluates weather agents based on their data comprehensiveness. To make both the agent composition framework and the three weather agents members of the agent marketplace, the agent framework is assigned with an Ethereum Rinkeby test network account with an amount of mock fund. Three independent Rinkeby accounts are also set up for the weather agents. Then, we manually registered the agent framework as a user and the weathers agents as service providers and deposit a nominal amount of Ether for the user (the framework) and the agents. In addition, to connect the agent instance in the JPS and their records within the agent marketplace, we extended the OntoAgent onotlogy. A new



Score updated to : 0x0B9056fcbf59D283F7c6B909Ea729182Bd69D36E
Agent : Service__OpenWeatherMap   @Wed Nov 28 2018 15:52:04

Payment made   : 0x0B9056fcbf59D283F7c6B909Ea729182Bd69D36E
Agent : Service__OpenWeatherMap   @Wed Nov 28 2018 15:52:16

● http://www.theworldavatar.com/kb/agents/Service__YahooWeather
● http://www.theworldavatar.com/kb/agents/Service__AccuWeather
● http://www.theworldavatar.com/kb/agents/ Service__OpenWeather
33.5%
36.5%
29.9%
0x0B9056fcbf59D283F7c6B909Ea72
0x1c5048f718289e8b7050648ceb9b330a7b3f310f
0x700B09517ddaCF44617371D8D00bB09FE9425F

Figure 7: *A screenshot of the agent composition framework integrated with the agent marketplace: a) the agent marketplace provides QoS scores for all three weather agents for the optimization for the composition result (the upper part) b) the new performance evaluation result will be updated to the contract (the lower part). The pie chart demonstrates the market share of the three weather agents.*

property ontoagent:hasBlockchainAddr[14] is added to OntoAgent, which stores the address of the Rinkeby network address of the agent.

After such a setup, the framework can now lookup the performance records of three

[14] http://www.theworldavatar.com/ontology/ontoagent/#hasBlockchainAddr

weather agents (as shown in Figure 7) and execute them through the Smart Contract. During the optimization process, the framework first queries the KG with SPARQL and retrieve the agents' Rinkeby addresses. With the addresses, the framework then calls the Smart Contract to look up the scores of the agents and ranks the agents according to the scores. Finally, the agent with the highest score will be kept. According to the cumulative score generated based on previous evaluations, OpenWeatherMap agent is selected.

After the optimization of the composite agent, the framework proceeds to the execution phase. To execute an agent, the agent composition framework sends the HTTP request to invoke OpenWeatherMap agent to the Smart Contract. Through Oraclize, the Smart Contract will make an HTTP request to the agent. In this use case, the Smart Contract will pay the Oraclize an amount of service fee for this request. Therefore, the agent invoked will be charged an extra fee by the Smart Contract.

When the Smart Contract receives the weather data provided by the OpenWeatherMap, it will then make a performance evaluation based on the completeness of the data (*e.g.,* some weather agent include the cloud coverage data but some do not). By going through the weather data and compare the attributes included in the result with a predefined list, the Smart Contract then comes up with a score reflecting the comprehensiveness of the weather data. As shown from Line 36 in the Appendix A.1, the Smart Contract searches for the URIs of the data properties such as wo:hasHumidity and counts the number of data properties contained in the result returned. Since the maximum number of data properties returned is seven, the number of data properties will be divided by seven for normalization. (In fact, Solidity currently only supports the storage of integers, the performance scores are therefore in the form of large integers in the actual implementation. However, to better demonstrate the design, we simplified the scores to float numbers in this paper. Subsequently, the Smart Contract will calculate and update the new cumulative performance score of the agent as shown in Line 38 in Appendix A.1. The performance score of this invocation is 4.28 out of 5 because the weather agent only returned 6 properties out of 7. Therefore 85% of the price, which is 0.085 Ether, will be transferred to the weather agent by the Smart Contract, as the Ethereum Smart Contracts are able to transfer Ether to other nodes. Figure 7 contains a message showing that a payment is made to the agent's hexadecimal Ethereum account address by the Smart Contract. Moreover, Figure 8 demonstrate the output produced by the composite agent.

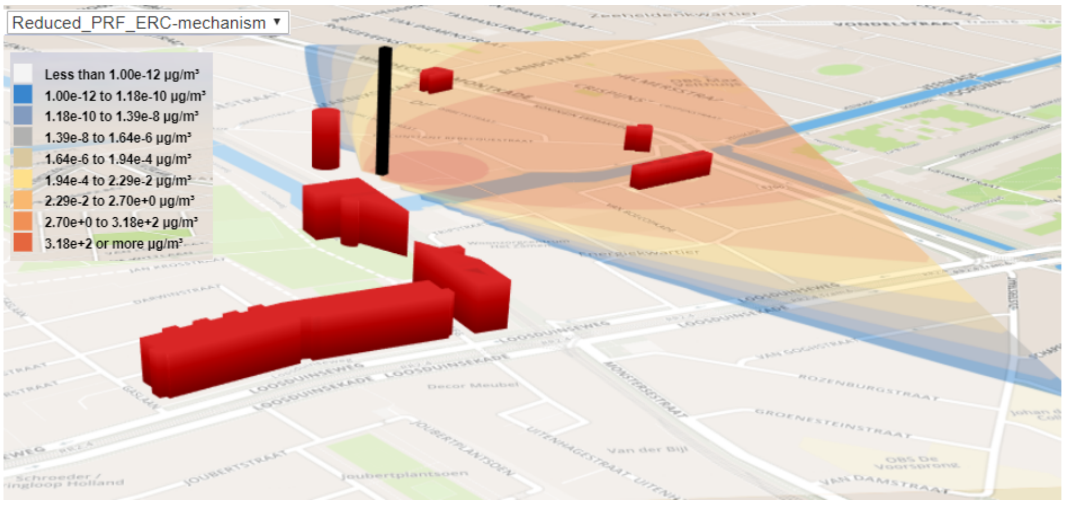


Figure 8: *A screenshot of the visualization of the air dispersion simulated by the composite agent.*

5 Outlook

In the future, the Knowledge Graph will include semantic instances of the agent marketplaces for different types of agents. As mentioned before, we envision that for a certain category of agents there will be a designated agent marketplace implemented, since the standard for evaluating different type of agent will vary. Therefore, when there are multiple types of agents in the KG, the composition framework needs a way to automatically locate the agent marketplace for a certain category of agents. We believe that by creating semantic instances of marketplace containing explicit indications about what type of agents does the marketplace register, the agent composition framework will be able to discover a suitable agent marketplace using the agent ontology.

Moreover, the agents within the marketplace are currently restricted to those representing cyber resources such as capacity for simulation or optimization. This is because of the lack of measures to collect untampered data to evaluate agents representing physical resources. For example, in order to evaluate the performance of a transportation agent sensors for location and status of the cargo are needed; however, it is difficult to prevent fraudulent behaviours such as manipulation of sensor signal. We propose that by embedding Smart Contracts into the firmware of physical devices such as sensors and integrating them into the blockchain network, the sensors could serve as unbiased estimators of the performance of a physical activity (*e.g.,* whether a storage tank has been filled) in the future. Consequently, the scope of the agent marketplace could be further extended.

6 Limitations

Firstly, although the blockchain-based Smart Contract has been widely applied in many fields, it is still a rather immature technology. Take the Ethereum framework and the Solidity language as examples, both of them are criticized for a series of known bugs [1].

The immaturity of the technology raises risks for the agent marketplace. For example, the Solidity language is still under development and this may lead to possibility of being hacked, hence Smart Contracts may not be reliable. However, we trust that this technology will be largely improved in the future since it is promising.

Secondly, although the transparency of the Smart Contract enhanced the users' trust for the agent marketplace, it also exposes the marketplace's loop hole if there is any. Nevertheless, with the advent of more testing and validation tools, the Smart Contracts could be further improved in the aspect of resistance against attacks.

Thirdly, this work is built on the assumption that the agents representing cyber resources can be evaluated by computer programs given the data returned by the agents. However, such an assumption is not applicable for all scenarios. For example, to evaluate an agent that forecasts the prices on the stock market, the accuracy of the forecast is the most critical parameter for its evaluation. However, it is not possible to calculate the accuracy based on the data returned by the agents. As a result, some agents can not be included in the agent marketplace without some further consideration, *e.g.* considering past performance.

Lastly, the proof-of-work mechanism of the blockchain inevitably causes time delay for transaction because the updating to the blockchain must be validated. Consequently, when the agent marketplace is implemented on top of a proof-of-work blockchain, it is not suitable for agents that are response-time sensitive.

7 Conclusion

This paper presents the Smart Contract-based agent marketplace, which is able to provide access for the agent composition framework to estimate the reliability of agents within a KG. By its design, the agent marketplace is clearly compatible with the highly automated nature of agent composition framework, invulnerable from the fraudulent ratings from both other users and administrators, and scalable enough to fit the vast number of agents within a KG. Also, the paper demonstrates the application of the agent marketplace within the JPS in order to support the JPS agent composition framework for agent selection and payment.

Acknowledgements

This project is supported by CMCL Innovations and the National Research Foundation (NRF), Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise(CREATE) programme. Markus Kraft acknowledges the support of the Alexander von Humboldt foundation.

List of abbreviations

HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
URL	**U**niform **R**esource **L**ocator
URI	**U**niform **R**esource **I**dentifier
UML	**U**nified **M**odeling **L**anguage
JPS	**J**-**P**ark **S**imulator
KG	**K**nowledge **G**raph
SPARQL	**S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage

References

[1] Solidity v0.4.24 – list of known bugs, 2018. URL https://solidity.readthedocs.io/en/v0.4.24/bugs.html. Accessed March 11th, 2019.

[2] Mol-Instincts, 2019. URL http://cccbdb.nist.gov/. Accessed September 13th, 2019.

[3] Oraclize, 2019. URL http://www.oraclize.it/. Accessed September 19, 2019.

[4] P. Buerger, J. Akroyd, S. Mosbach, and M. Kraft. A systematic method to estimate and validate enthalpies of formation using error-cancelling balanced reactions. *Combustion and Flame*, 187:105 – 121, 2018. doi:10.1016/j.combustflame.2017.08.013.

[5] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2013. http://ethereum.org/ethereum.html. Accessed September 19, 2019.

[6] D. Calvaresi, A. Dubovitskaya, D. Retaggi, A. F. Dragoni, and M. Schumacher. Trusted Registration, Negotiation, and Service Evaluation in Multi-Agent Systems throughout the Blockchain Technology. In *2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. IEEE, 2018. doi:10.1109/wi.2018.0-107.

[7] D. Carboni. Feedback based Reputation on top of the Bitcoin Blockchain. *CoRR*, abs/1502.01504, 2015. URL http://arxiv.org/abs/1502.01504.

[8] P. Cuccuru. Beyond bitcoin: an early overview on smart contracts. *International Journal of Law and Information Technology*, 25(3):179–195, 2017. doi:10.1093/ijlit/eax003.

[9] R. Dennis and G. Owen. Rep on the block: A next generation reputation system based on the blockchain. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2015. doi:10.1109/icitst.2015.7412073.

[10] A. Devanand, M. Kraft, and I. A. Karimi. Optimal site selection for modular nuclear power plants. *Computers & Chemical Engineering*, 125:339 – 350, 2019. doi:10.1016/j.compchemeng.2019.03.024.

[11] A. Eibeck, M. Q. Lim, and M. Kraft. J-Park Simulator: An ontology-based platform for cross-domain scenarios in process industry, 2019. Submitted for publication.

[12] F. Farazi, J. Akroyd, S. Mosbach, P. Buerger, D. Nurkowski, and M. Kraft. OntoKin: An ontology for chemical kinetic reaction mechanisms, 2019. Submitted for publication.

[13] D. Fraga, Z. Bankovic, and J. M. Moya. A taxonomy of trust and reputation system attacks. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 41–50. IEEE, 2012. doi:10.1109/trustcom.2012.58.

[14] A. Jøsang. Trust and Reputation Systems. In *Foundations of Security Analysis and Design IV*, pages 209–245, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74810-6_8.

[15] M. Klems, J. Eberhardt, S. Tai, S. Härtlein, S. Buchholz, and A. Tidjani. Trustless Intermediation in Blockchain-Based Decentralized Service Marketplaces. In *Service-Oriented Computing*, pages 731–739. Springer International Publishing, 2017. doi:10.1007/978-3-319-69035-3_53.

[16] E. Koutrouli and A. Tsalgatidou. Taxonomy of attacks and defense mechanisms in P2P reputation systems - Lessons for reputation system designers. *Computer Science Review*, 6(2-3):47–70, 2012. doi:10.1016/j.cosrev.2012.01.002.

[17] S. Liu, H. Yu, C. Miao, and A. C. Kot. A fuzzy logic based reputation model against unfair ratings. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 821–828. International Foundation for Autonomous Agents and Multiagent Systems, 2013. `http://dl.acm.org/citation.cfm?id=2484920.2485051`. Accessed May 23rd, 2019.

[18] D. Macrinici, C. Cartofeanu, and S. Gao. Smart contract applications within blockchain technology: A systematic mapping study. *Telematics and Informatics*, 35(8):2337 – 2354, 2018. doi:10.1016/j.tele.2018.10.004.

[19] J. Morbach, A. Wiesner, and W. Marquardt. OntoCAPE: A (re) usable ontology for computer-aided process engineering. *Computers & Chemical Engineering*, 33(10): 1546–1556, 2009. doi:10.1016/j.compchemeng.2009.01.019.

[20] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `https://bitcoin.org/bitcoin.pdf`. Accessed May 23rd, 2019.

[21] R. D. Johnson III (ed.). NIST Computational Chemistry Comparison and Benchmark Database, NIST Standard Reference Database Number 101, Release 19, 2018. URL `http://cccbdb.nist.gov/`. Accessed September 13th, 2019.

[22] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000. doi:10.1145/355112.355122.

[23] J. J. Sikorski, J. Haughton, and M. Kraft. Blockchain technology in the chemical industry: Machine-to-machine electricity market. *Applied Energy*, 195:234–246, 2017. doi:10.1016/j.apenergy.2017.03.039.

[24] Y. Sun and Y. Liu. Security of Online Reputation Systems: The evolution of attacks and defenses. *IEEE Signal Processing Magazine*, 29(2):87–97, 2012. doi:10.1109/msp.2011.942344.

[25] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. doi:10.1006/knac.1993.1008.

[26] A. Whitby, A. Jøsang, and J. Indulska. Filtering out unfair ratings in Bayesian reputation systems, 2004. `https://www.csee.umbc.edu/~msmith27/readings/public/whitby-2004a.pdf`. Accessed February 15th, 2019.

[27] G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. `http://gavwood.com/paper.pdf`. Accessed June 1st, 2019.

[28] Y. Yang, Y. L. Sun, S. Kay, and Q. Yang. Defending online reputation systems against collaborative unfair raters through signal modeling and trust. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1308–1315. ACM, 2009. doi:10.1145/1529282.1529575.

[29] Y. Zhang, J. Bian, and W. Zhu. Trust fraud: A crucial challenge for China's e-commerce market. *Electronic Commerce Research and Applications*, 12(5):299–308, 2013. doi:10.1016/j.elerap.2012.11.005.

[30] L. Zhou, C. Zhang, I. A. Karimi, and M. Kraft. An ontology framework towards decentralized information management for eco-industrial parks. *Computers & Chemical Engineering*, 118:49–63, 2018. doi:10.1016/j.compchemeng.2018.07.010.

[31] X. Zhou, A. Eibeck, M. Q. Lim, N. Krdzavac, and M. Kraft. An agent composition framework for the J-Park Simulator - a knowledge graph for the process industry, 2019. Submitted for publication.

A Appendices

A.1 Solidity code for the reputation system

```
...
event Insufficient_deposit(address user_address, uint256 balance);
event Callback_Received(address requester_address, bytes32 query_id, string
     result);

function _call(address _agent_address, string memory _data) public{
    string memory _URL = agent_map[_agent_address].URL;
    bytes32 _query_id = oraclize_query("URL", join_URL(_URL, _data));
}


function __callback(bytes32 _myid, string memory _result, address
    _sender_address) public {
    require (msg.sender == oraclize_cbAddress());
    emit Callback_Received(_sender_address, _myid, _result);
}

function invoke(address agent_address, data) {
    if (check_deposit){
        call(agent_address, data);
    }
    else{
        emit Insufficient_deposit(msg.sender, user_map[msg.sender].
            deposit_balance);
    }
}


function check_deposit(address _user_address, address _agent_address)
    private returns (boolean sufficient){
    return user_map[_user_address].deposit >=  agent_map[_agent_address].
        price;
}

function evaluate_performance(string memory _result, address _user_address,
     address _agent_address) private {
   /*
     Domain specific algorithm for evaluation
   */

   uint score = 0;
   if (_result.find("wo:hasHumidity")){
        score = score + 1;
   }

   if (_result.find("wo:hasWindDirection")){
        score = score + 1;
   }

```

```
        if (_result.find("wo:hasWindSpeed")){
            score = score + 1;
        }

        if (_result.find("wo:hasCloudCoverage")){
            score = score + 1;
        }

        if (_result.find("wo:hasTemperature")){
            score = score + 1;
        }

        if (_result.find("wo:hasPrecipitation")){
            score = score + 1;
        }

        if (_result.find("wo:hasAtmosphericPressure")){
            score = score + 1;
        }

        score = score / 7 * 50000;

        agent_map[_agent_address].invocation_counter =
        agent_map[_agent_address].invocation_counter + 1;
        calculate_payment();

        agent_map[_agent_address].total_score =
        agent_map[_agent_address].total_score + score;

        user_map[_user_address].invocation_counter =
        user_map[_user_address].invocation_counter + 1;
    }

    function join_URL(string memory _URL, string memory _data) private pure
        returns (string memory result ){
        return string(abi.encodePacked(_URL,_data));
    }
    ...
```

A.2 Solidity registration and transaction

```

    ...
    uint public minimum_deposit_for_agent = 2 ether;
    uint public minimum_deposit_for_user =0.2 ether;
    uint public default_score = 45000;
    address[] agent_address;


    mapping(address=>agent) agent_map;
    mapping(address=>uint)  user_deposit_map;


    struct agent {
        uint score;
        uint deposit_balance;
        uint invocation_count;
        string URL;
        bool validity;
        uint price;
    }


    function register_as_agent (string memory _URL, uint _price)
    public payable returns (bool _succeed){
        if(msg.value >= minimum_deposit_for_agent){
            if(!agent_map[msg.sender].validity)
            {   // register the new vendor
                agent_map[msg.sender].deposit_balance = msg.value;
                agent_map[msg.sender].validity = true;
                agent_map[msg.sender].URL = _URL;
                agent_map[msg.sender].invocation_count = 0;
                agent_map[msg.sender].score = default_score;
                agent_map[msg.sender].price = _price;
                agent_address.push(msg.sender);
                return true; }
        }
        return false;
    }


    function register_as_user() public payable returns (bool _succeed)
    {
        return user_deposit_map[msg.sender] = msg.value;
    }

    ...
    uint public commission_charge = 0.01 ether;
    uint public compensation_charge = 0.1 ether;
    uint public default_score = 45000;
    uint public full_payment_score = 40000;
    uint public half_payment_score = 30000;
    uint public compensation_score = 20000;
    ...

    function calculate_payment(uint score, address payable  _user_address, address payable _agent_address) private{
        uint price = agent_map[_agent_address].price;
        if (score >= full_payment_score) // make full make_payment_or_compensation
        {
            _payment =  price + commission_charge;
            deduce_deposit(user_map, _user_address, _payment);
            _agent_address.transfer(price);

        }
        else if (score >= half_payment_score){
            _payment =  price/2 - commission_charge;
            deduce_deposit(user_map, _user_address, _payment);
            _agent_address.transfer(price/2);

        }else{
            _payment =  compensation_charge - commission_charge;
            deduce_deposit(agent_map, _user_address, _payment);
            _user_address.transfer(compensation_charge);
        }
    }

    function deduce_deposit(mapping _map, address _payer_address, uint _payment) private {
        _map[_payer_address].deposit_balance =  _map[_payer_address].deposit_balance - _payment;
    }

    function get_all_agents_address() public view returns(address[] memory _vendors_address)
    {  return agent_address; }

    function get_agent_record(address _agent_address)
    public view returns(uint _score, uint _invocation, bool _validity, uint _price){
        return  (agent_map[_agent_address].score,
        agent_map[_agent_address].invocation_count,
        agent_map[_agent_address].validity, agent_map[_agent_address].price);
    }

    function top_up_deposit() public payable{
        agent_map[msg.sender].deposit_balance =
        agent_map[msg.sender].deposit_balance + msg.value;
    }
```