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

Xiaochi Zhou[a,b], Mei Qi Lim[a], Markus Kraft[a,b,c,*]

[a] *Cambridge Centre for Advanced Research and Education in Singapore (CARES), CREATE Tower, 1 Create Way, Singapore, 138602*
[b] *Department of Chemical Engineering and Biotechnology, University of Cambridge, Philippa Fawcett Drive, West Site, CB3 0AS Cambridge, UK*
[c] *Nanyang Technological University, School of Chemical and Biomedical Engineering,62 Nanyang Drive, Singapore, 637459*

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 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 agent marketplace is built on top of blockchain-based Smart Contracts.
- The agent marketplace is integrated into an agent composition framework,

*Corresponding author
Email address: mk306@cam.ac.uk (Markus Kraft)

to support automated agent selection and payment.

Keywords: `Blockchain, Smart Contract, Agent, Agent composition, Knowledge Graph`

1. Introduction

The progress in information technology is changing the field of chemical engineering at an increasing pace. For example, the number of agents that provide functions relevant to chemical engineering over the internet or intranet is increasing. Here we define agents as applications and web services that utilize semantic technologies and are accessible on World Wide Web. They are used for data acquisition, simulation, and optimization. For instance, the two thermodynamic databases NIST chemistry webBook [1] 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 [3, 4]. Knowledge Graphs are sets of interconnected classes, relations, and instances that are semantically described, *i.e.,* each distinct piece of information is denoted by a different 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 an ontology. Within the KGs, information such as the input/output (I/O) signatures and prices of agents are described on top of agent ontologies such as OntoAgent [4] 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.

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

The J-Park Simulator (JPS) [3, 4] 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, which carry out only one relatively simple function, from the KG and puts them into a sequence based on their I/O signature (*i.e.,* the data type of their inputs and outputs). For example, if one atomic agent takes a coordinate under the ontology class OntoCityGML:EnvelopeType as its output and returns a city name under the class dbo:City, it can be matched with an atomic agent that takes an instance of dbo:City and returns the population of this city. The agent composition framework will then compose this two agents to produce a composite agent that returns the city population given a selected coordinate. After creating a composite agent, the composition framework executes the composite agents. The details of the JPS agent composition framework has been described in this article. [4]

Any composition framework must face the problem of assessing the atomic agents' performance and reliability. In other words, the framework needs to know whether or not 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 affect the accuracy, predictive performance, and quality of the models that utilize this data [5]. In addition, agent performance and reliability are important selection criteria within the agent composition framework when there are multiple functionally identical agents available. Therefore, it is critical to provide credible information on the performance and reliability of the agents.

One of the most traditional ways 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 are 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 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 KGs. 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 assess performance and reliability, feedback-based reputation systems are considered as the most cost-effective and scalable. 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 from 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 purchase, and hence establishes users' confidence for this purchase [6, 7]. 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 [6, 7] and it can operate automatically. Such a solution is scalable and compatible with automated systems.

However, the current implementation suffers from persistent problems of rating fraud, from both users and administrators [8, 9, 10], which discredits the reputation system. Firstly, the higher ratings lead to more profit [11] 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. In this case, 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 for this purpose [12, 13, 14]. However, when the profit of creating a fraudulent rating surpasses its cost, the well-funded dishonest rating could be deliberately masked

as a regular one, 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 Dispersion agent is built on top of a proprietary Atmospheric Dispersion Modeling 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 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 who could also behave dishonestly. For example, an administrator may take bribery from an agent provider to tamper with the scores of their agent. Clearly, the countermeasures against fraud from agent providers are no longer applicable.

With the advent of 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 [15] proposed a blockchain-based P2P reputation system for file transaction. Carboni [16] designed an incentive-based feedback reputation 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 detail in Section 2.2, is proposed to provide decentralized control over the implementation of func-

tions [17]. Calvaresi et al.[18] proposed a reputation system in which the cumulative performance score of an agent is automatically calculated and managed by the Smart Contract and stored in the blockchain. This solution takes it 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. [19] 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 feedback from users. Consequently, although the Smart Contract-based solution prevents the risk of fraud from administrators, it still exposes the reputation system to the risk of rating frauds from users.

In conclusion, to the best of our knowledge, a mechanism to provide a 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 being resistant to 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 framework 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 fraud from both users and 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 2 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 5.2 provides plans for future improvement of the work, in relation to the limitations discussed in Section 5.1. Section 5 outlines the conclusions of this paper.

2. Background

In this section we provide some information on the J-Park Simulator (JPS) and Ethereum Smart Contracts. The JPS will provide the environment in which we built our use case and demonstrated the effectiveness of our solution. Ethereum 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 simulations and optimizations, 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 can be enabled.

For example, on top of JPS, Zhou et al. [20] proposed a methodology to use ontologies for the modeling and management of eco-industrial parks 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 more easily found. In addition, Devanand et al. [21] gave an example of using the JPS cross-domain KG to access financial and geographical information of potential sites for the optimal placement of a small modular reactor. Such an approach is enabled by the KG's capability to effectively incorporate cross-domain data and provide convenient access to them.

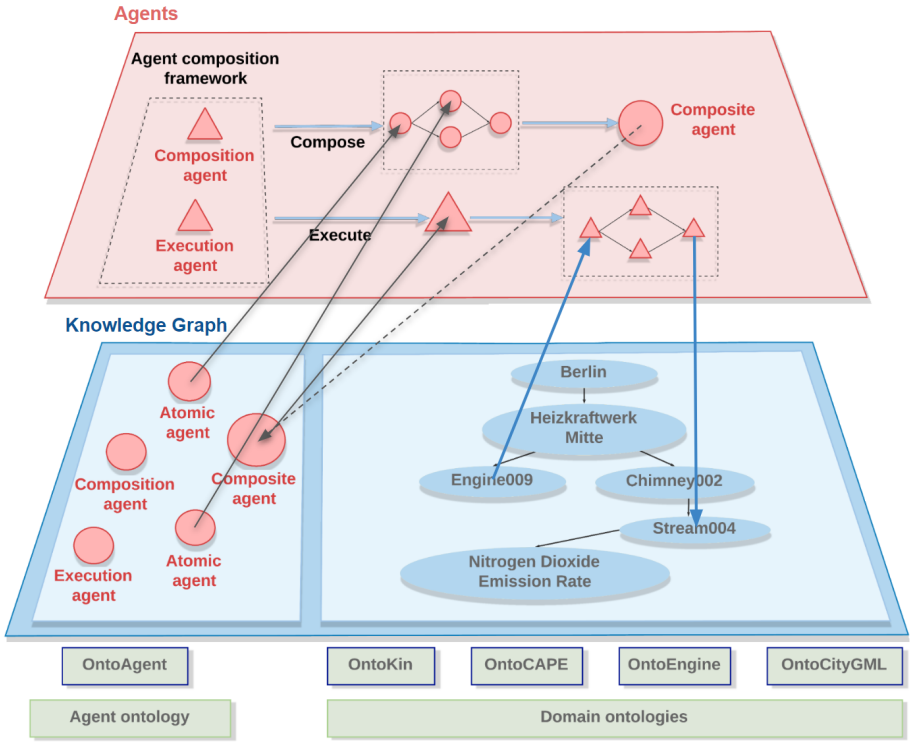

Figure 1: The JPS KG and agents: a) The JPS KG (blue layer) contains both the agent ontology (left) and the domain ontologies (right). b) Each ontology includes the terminology (green boxes) and the instances (pink nodes for agent instances and blue nodes for domain ontologies). c) The agent layer (pink 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.

To constitute the JPS cross-domain KG, a set of ontologies are developed or incorporated that contain structured and connected knowledge and data that are represented semantically, *i.e.* classes and individuals are denoted by Uniform Resource Identifiers (URIs). As each URI uniquely points to a distinct class or individual (*e.g. dbr:Cambridge*[2] and *dbr:Cambridge,_Massachusetts*[3] denote "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 classes and individuals are referred to as an ontology [22]. Furthermore, a collection of interconnected ontologies integrated for a certain purpose is considered as a KG according to our understanding.

As shown in 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 [23] is integrated to describe concepts and individuals that are related to chemical process engineering. Starting from OntoCAPE, OntoEIP [20] is developed to model eco-industrial parks. Moreover, OntoCityGML, OntoKin, and OntoEngine [24] are included to cover the field of city modeling, chemical kinetics, and internal combustion engines. These ontologies are used collectively to provide knowledge and data for cross-domain simulation and optimization.

However, these ontologies must be continuously managed and updated due to the dynamic 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 [4] ontology. For an agent instance, OntoAgent describes its I/O signature by assigning domain ontology classes to its inputs and outputs, so that machines can

[2] `http://dbpedia.org/resource/Cambridge`

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

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 demonstrates 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 selects out the optimal agent among those that are functionally identical by ranking them with regard to their performance scores. However, due to the lack of a scalable and secure approach to provide a 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 an 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, Ethereum 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 [25, 17] is a blockchain that is designated to support the deployment of Smart Contract-based decentralized applications (DAPPs). The main purpose of this paper is to demonstrate a proof-of-concept implementation. Therefore, since the Ethereum blockchain is a rather well-known public blockchain with sufficient documentations, we believe this public platform is particularly suitable for the implementation in this paper.

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 is in accord with the prior block, which only takes minimal computational power, a user on the blockchain could verify the integrity of all the records 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 this, Ethereum and many other blockchains implement proof-of-work systems to increase the difficulty of recalculating 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 recalculate the blockchain. As a result, the historical records in the blockchain are secured.

Furthermore, 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 can 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. [26] 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 are starting to support a blockchain-based Smart Contract. The major ones are Bitcoin [27] and Ethereum [28]. However, the Bitcoin blockchain only provides a limited set of functionalities for their Smart Contracts [29]. 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, the same as other data on the blockchain, the bytecodes are inherently tamper-proof. Currently, Solidity[4] is the main language used to develop Ethereum Smart Contracts. The Ethereum blockchain includes Ethereum Virtual Machines (EVMs). The EVMs, which are installed on each node locally, assure 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 with. 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,

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

the cryptocurrency on the Ethereum blockchain, and make transactions to other nodes too. This feature allows Ethereum Smart Contracts to carry out tasks that involve financial transactions.

Another feature of the Ethereum Smart Contracts 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 state changes on it can be precisely reproduced based on the transaction records. However, an HTTP request may return different results with the same input. As a result, the Smart Contracts cannot receive any data, including the result of an HTTP call, other than explicit transactions from other nodes on the blockchain.

Therefore, the Oraclize [30] 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 requests indirectly via the Oraclize service.

In addition to the features of Smart Contracts themselves, some tools are implemented to test and inspect them. Besides the Ethereum main blockchain

network, Ethereum also provided a test network named Rinkeby[5]. On the main network, Ethers are acquired from mining and purchasing; therefore, the Ethers are of real-world market value. To avoid financial losses while testing Smart Contracts, we deployed and hence tested 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 accords with the bytecode published.

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

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

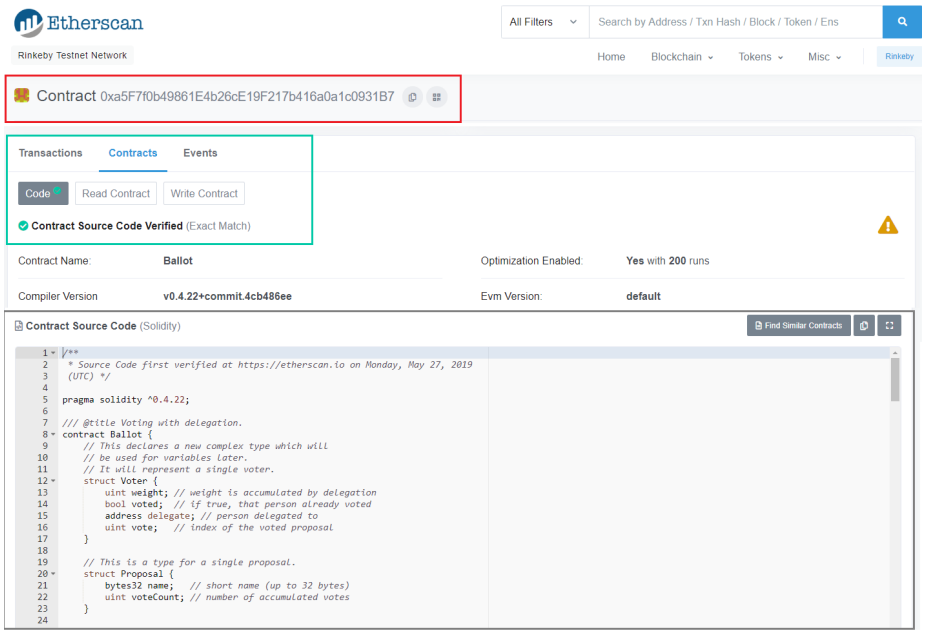

Figure 2: The screenshot of a Smart Contract information page: a) In the red box is the address of the contract, which is a hexadecimal number. Knowing this address, other nodes on the blockchain can make transactions to the contract or call its functions. b) Within the green box, it shows that this Smart Contract is verified, in the sense that its compiled bytecode published on the blockchain accords 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 of, firstly, evaluating the trustworthiness of the agent; secondly, selecting the optimal agent among functionally identical ones; and, thirdly, introducing a payment mechanism to facilitate a 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. One major difference between our design and the tradition centralized designs of reputation-systems is to leverage Smart Contracts as unbiased and autonomous third parties to provide ratings and avoid fraudulent ratings from human users. In addition, Blockchain-based Smart Contracts are published on Blockchains, as introduced in Section 2.2, one can investigate and verify whether the Smart Contracts implemented accord to their source code. As a result, any dishonest or biased code within the Smart Contracts can be spotted. Therefore, malicious behaviour from the team that implement or administrate the agent marketplace can be prevented.

The Smart Contracts making up this agent marketplace are developed in the Solidity[6] language and published on the Ethereum Rinkeby test network.

3.1. Reputation system

As mentioned above, building the reputation system based on users' feedback may lead to frauds. However, the features of the tamper-proof code and the decentralized execution of the blockchain-based Smart Contracts have 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.

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

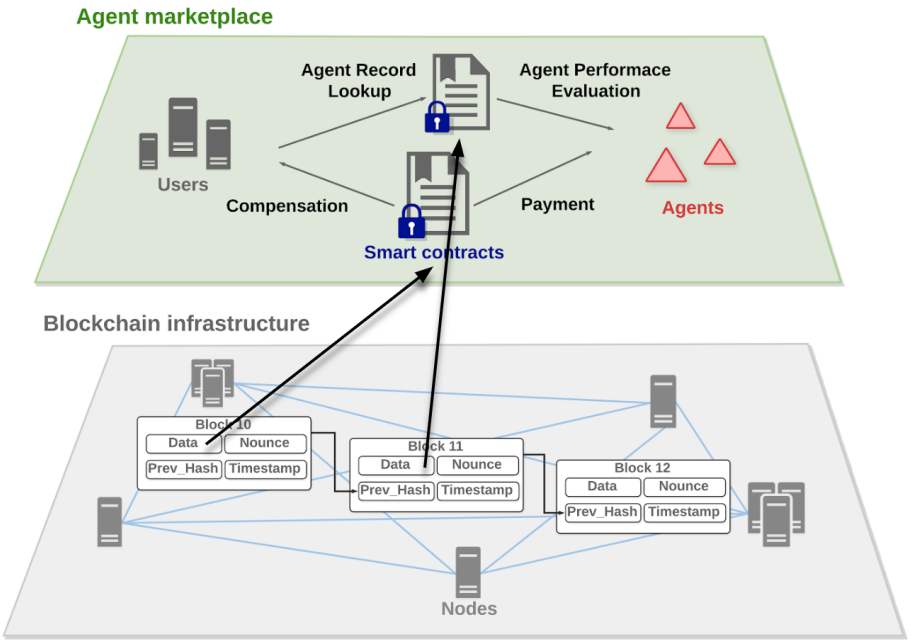

Figure 3: The Smart Contract-based agent marketplace: a) The agent marketplace (the green layer) is established with Smart Contracts on top of a decentralized blockchain infrastructure (the gray layer), which is updated and validated by a number of connected nodes collectively. b) The Smart Contracts 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.

As illustrated in Figure 4, to employ an agent, the user (the agent composition framework) needs to call the ***invoke()*** function, providing the Ethereum 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

through it for the agents. 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[7]. 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 are part of the KG. As mentioned above, the agents within the KG share common ground for data exchange and are interoperable. 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 do this 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 detail.

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

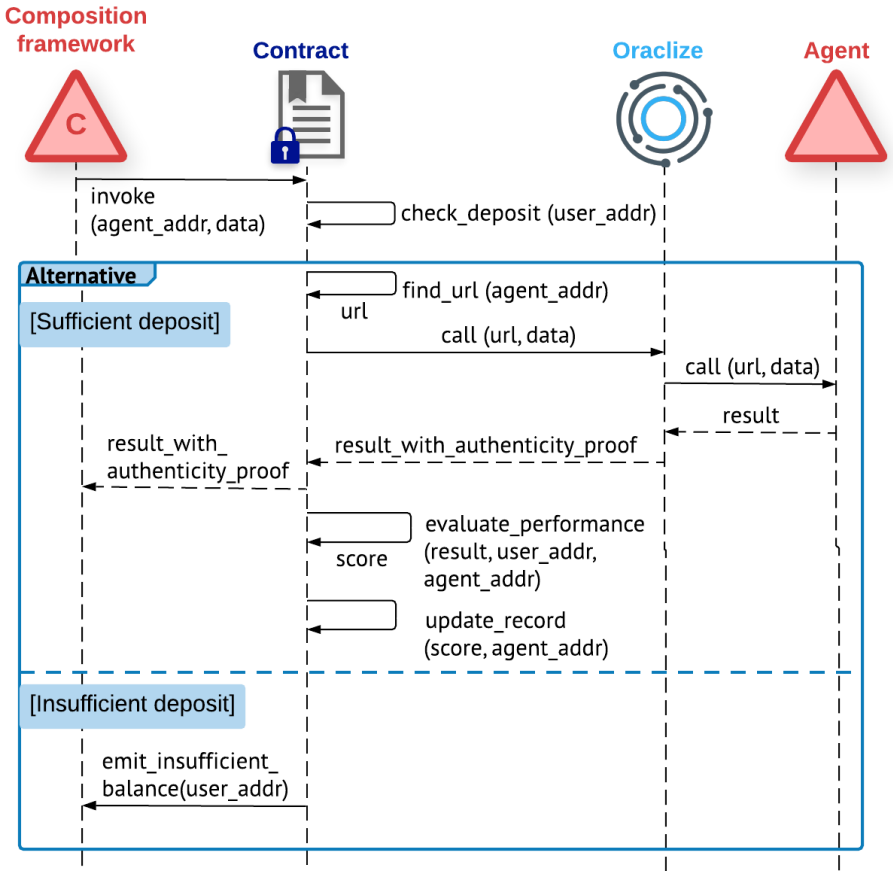

Figure 4: UML diagram for the agent reputation system: a) The process of performance evaluation starts from the user making a request to invoke an agent, providing the hexadecimal address of the agent (invoke(agent_addr,data)). b) With sufficient deposit, the contract will delegate the invocation of the agent to the Oraclize service, which returns the result with proof of authenticity. Based on the result, the Smart Contract will make an evaluation of 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) In the case of an 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 of Ethereum Smart

Contracts being able to receive and transfer funds from and to other Ethereum accounts. Meanwhile, such a design enables the payment mechanism to pay the users compensation when the performance is lower than a certain threshold or the agent fails to provide a service. This feature further enhances the user's trust in the agents as they would automatically receive financial compensation for poor agent performance. Figure 5 demonstrates the working flow of the payment mechanism and Appendix A.2 shows the Solidity source code of the implementation. In addition, considering the large-scale of the Knowledge Graph and the substantial number of agents operating on top of it, it is highly likely that many of them are proprietary and come with charges. Therefore, a payment system is a necessary component of the agent marketplace.

The payment system is based on a deposit system. Both the agents and 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 a deposit defined by the Smart Contract. The address of the 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 look up the agents' reputation records.

Since the Smart Contract can make an unbiased evaluation and cannot 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 the amount of payment with the function ***calculate_payment()*** and then make the transfer via the built-in ***transfer()*** function of the Smart Contract.

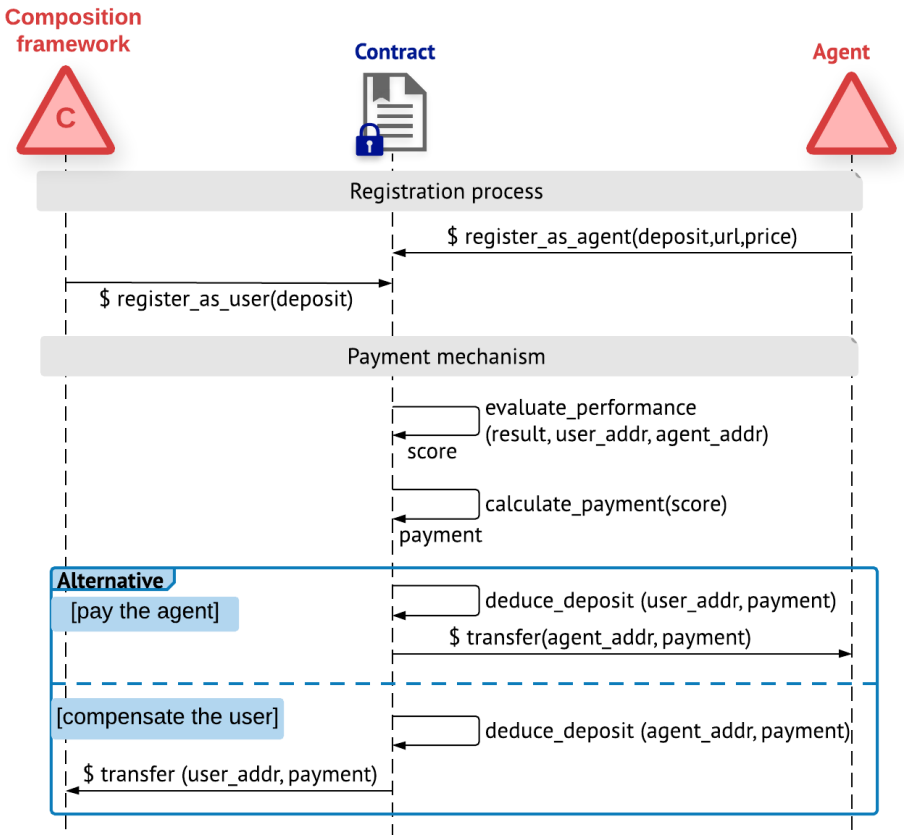

Figure 5: UML diagram for registration and payment mechanism: a) To register, the user must pay a deposit to the contract through the register_as_user(deposit) function. (The $ sign denotes a payable function). b) An agent register with its HTTP URL and its price for one invocation besides the deposit. 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.

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 Section 2, Background, 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 out-

come of this paper, it faces the problem of the absence of credible sources of the agents' performance and reliability. Because of the vast number of 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 "Dispersion agent" and "Engine Emission agent"

As a result, if a feedback-based reputation system is implemented for the JPS agent composition framework, there is a high risk of rating fraud 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 an agent after its execution under the agent composition framework.

As shown in Figure 7, in this use case the composite agent required has the inputs "Reaction Mechanism" and "Region" and its output is the type "Air Dispersion"[8]. Such a composite agent can be used to evaluate the suitability of proposed installation locations of a new power plant or chemical plant, or to assist evacuation planning in case of an 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 three different weather services accordingly. The

[8]which is temporarily represented by the class "Table" as the air dispersion class is under development.

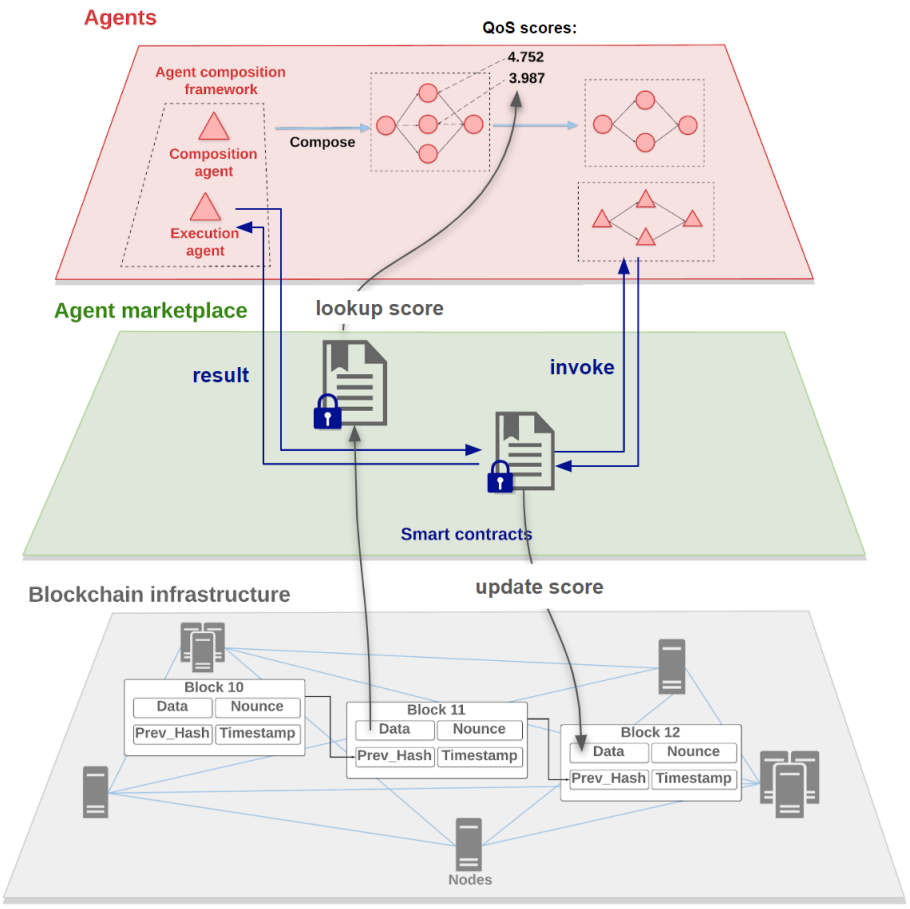

Figure 6: The integration with the JPS agent composition framework: the agent marketplace is then applied to the JPS project to provide performance score lookup, performance evaluation, and transaction functions. a) The grey arrows represent the Smart Contracts reading QoS scores from or updating them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

three weather agents are named as Weather agent A,B and C. The agents wrapping around these web services take the URIs of cities and return detailed and semantically restructured weather data including wind velocity, temperature, precipitation, *etc.* However, despite their identical I/O signature, there is a difference in 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 evaluates weather agents based on their data comprehensiveness. This paper aims to demonstrate a proof-of-concept implementation of agent marketplace on top of Smart Contracts, which is a rather new technology. Therefore, we decided to keep the use case as simple as possible. As a result, the evaluation of weather agents, which are relatively simple, based on single criterion. As the focus of this paper is to use Smart Contracts to prevent fraudulent ratings, we suppose that the specific evaluation matrix is less important here. Additionally, although the evaluation of weather agents is based on one criterion, evaluation matrices with multiple criteria can also be implemented. Moreover, the agent marketplaces can implement multiple alternative matrices focusing on different aspects of the agent performance to cope with various needs from agent users.

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 a sum of mock funds. Three independent Rinkeby accounts are also set up for the weather agents. Then, we manually registered the agent framework as a user and the weather agents as service providers and deposited 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 ontology. A new property, ontoagent:hasBlockchainAddr[9], is added to OntoAgent, which stores the Rinkeby network address of the agent.

After such a setup, the framework can now look up the performance records of three 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 retrieves 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

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

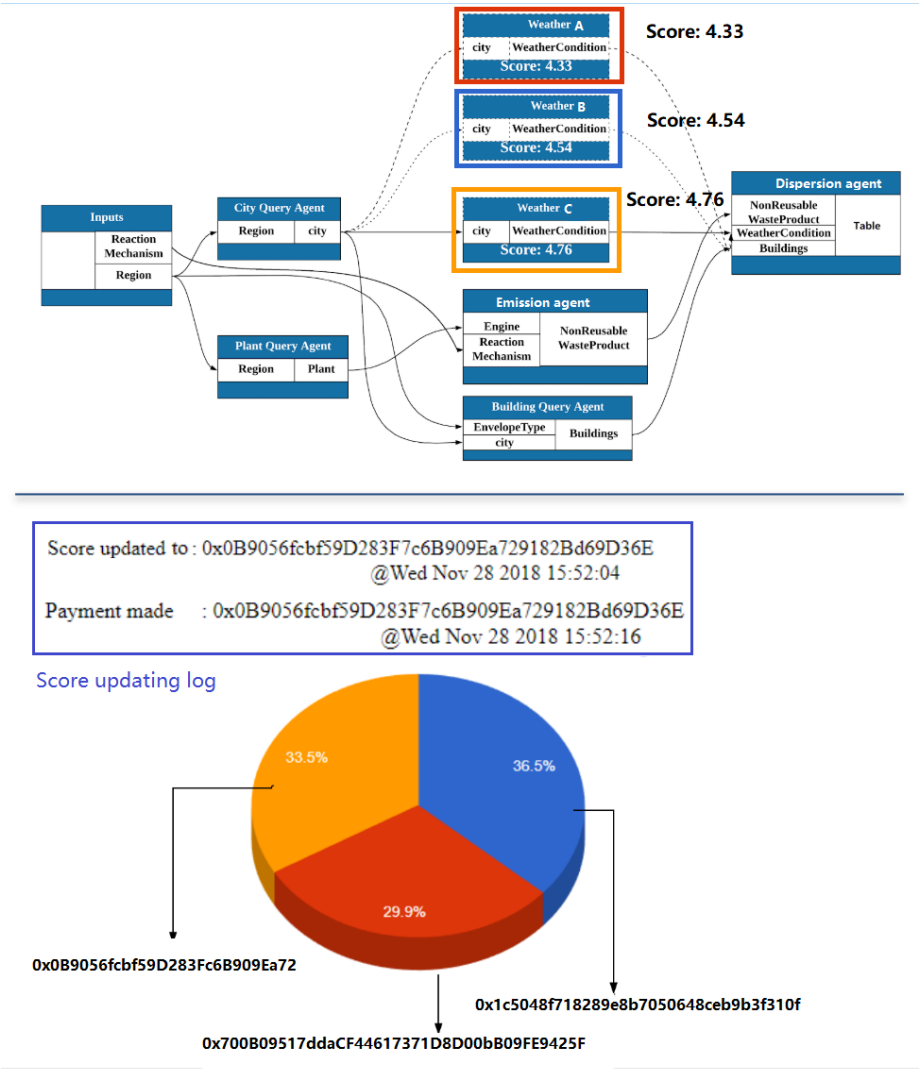

Figure 7: A screenshot of the agent composition framework integrated with the agent marketplace: a) The agent marketplace provides QoS scores for all three weather agents for the optimization of the composition result (top). b) The new performance evaluation result will be updated to the contract (bottom).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

based on previous evaluations, Weather agent C 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 the Weather agent C 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 a 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 weather service, it will then make a performance evaluation based on the completeness of the data (*e.g.,* some weather agents include cloud coverage data while others do not). By going through the weather data and comparing 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 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 six properties out of seven. 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 demonstrates the output produced by the composite agent.

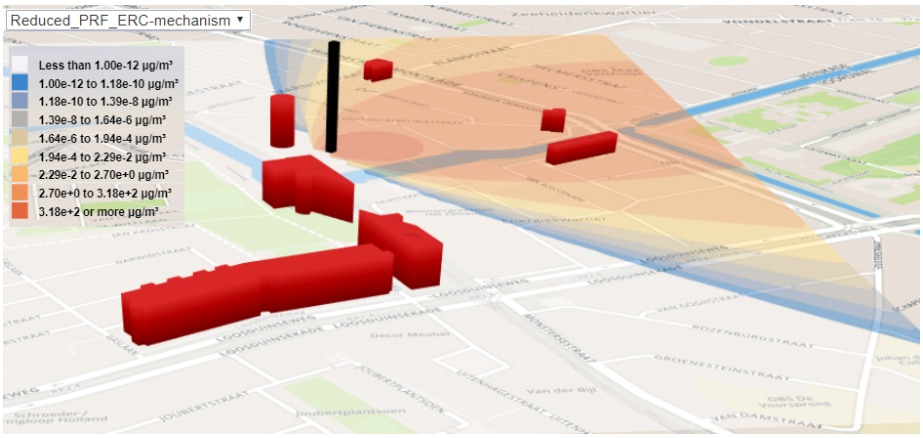

Figure 8: A screenshot of the visualization of the air dispersion simulated by the composite agent.

5. 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. Thanks to its design, the agent marketplace is clearly compatible with the highly automated nature of the agent composition framework, invulnerable from the fraudulent ratings from both other users and administrators, and scalable enough to fit a 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. In conclusion, the main contribution of this paper is to illustrate that Blockchain-based Smart Contracts can serve as absolutely trustworthy and unbiased third parties for evaluating agents operating on top of a Knowledge Graph, with a proof-of-concept implementation.

5.1. 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 have been crit-

icized for a series of known bugs [31]. 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 the possibility of being hacked, hence Smart Contracts may not be reliable. However, we trust that this technology will be largely improved in the future as it is so promising.

Secondly, although Smart Contracts' transparency enhanced users' trust of the agent marketplace, it also exposes any loopholes in the marketplace. Nevertheless, with the advent of more testing and validation tools, 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 stock market prices, the accuracy of the forecast is the most critical parameter. 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 further consideration, *e.g.* looking at past performance.

Lastly, the proof-of-work mechanism of the blockchain inevitably causes a time delay for transactions, because the update 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.

5.2. 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 types of agents 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 the marketplace containing explicit indications about the type of agents it registers, 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 the 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 the 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.

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] 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.

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

[3] A. Eibeck, M. Q. Lim, M. Kraft, J-Park Simulator: An ontology-based platform for cross-domain scenarios in process industry, 2019. URL: https://como.ceb.cam.ac.uk/preprints/222/, submitted for publication.

[4] X. Zhou, A. Eibeck, M. Q. Lim, N. Krdzavac, M. Kraft, An agent composition framework for the J-Park Simulator - a knowledge graph for the process industry, 2019. URL: https://como.ceb.cam.ac.uk/preprints/227/, submitted for publication.

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

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

[7] A. Jøsang, Trust and Reputation Systems, in: Foundations of Security Analysis and Design IV, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 209–245. doi:doi:10.1007/978-3-540-74810-6{_}8.

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

[9] D. Fraga, Z. Bankovic, 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, IEEE, 2012, pp. 41–50. doi:doi:10.1109/trustcom.2012.58.

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

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

[12] A. Whitby, A. Jøsang, J. Indulska, Filtering out unfair ratings in bayesian reputation systems, 2004. `https://www.csee.umbc.edu/~msmith27/readings/public/whitby-2004a.pdf` last accessed 2019-02-15.

[13] Y. Yang, Y. L. Sun, S. Kay, 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, ACM, 2009, pp. 1308–1315. doi:doi:10.1145/1529282.1529575.

[14] S. Liu, H. Yu, C. Miao, 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, International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 821–828. `http://dl.acm.org/citation.cfm?id=2484920.2485051` last accessed 2019-05-23.

[15] R. Dennis, 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:doi:10.1109/icitst.2015.7412073.

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

[17] V. Buterin, Ethereum: A next-generation smart contract and decentralized application platform, 2013. http://ethereum.org/ethereum.html last accessed: April 29, 2020.

[18] D. Calvaresi, A. Dubovitskaya, D. Retaggi, A. F. Dragoni, 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:doi:10.1109/wi.2018.0-107.

[19] M. Klems, J. Eberhardt, S. Tai, S. Härtlein, S. Buchholz, A. Tidjani, Trustless Intermediation in Blockchain-Based Decentralized Service Marketplaces, in: Service-Oriented Computing, Springer International Publishing, 2017, pp. 731–739. doi:doi:10.1007/978-3-319-69035-3{_}53.

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

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

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

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

[24] F. Farazi, J. Akroyd, S. Mosbach, P. Buerger, D. Nurkowski, M. Kraft, OntoKin: An ontology for chemical kinetic reaction mechanisms, 2019. URL: https://como.ceb.cam.ac.uk/preprints/218/, submitted for publication.

[25] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, 2014. http://gavwood.com/paper.pdf last accessed 2019-06-01.

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

[27] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008). https://bitcoin.org/bitcoin.pdf last accessed 2019-05-23.

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

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

[30] Oraclize, 2018. URL: http://www.oraclize.it/, accessed April 29, 2020.

[31] 2019. URL: https://solidity.readthedocs.io/en/v0.4.24/bugs.html, last accessed: April 29, 2020.

Appendix A. Appendices

Appendix 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;
```

```
28      }
29
30      function evaluate_performance(string memory _result, address
            _user_address, address _agent_address) private {
31          /*
32              Domain specific algorithm for evaluation
33          */
34
35          uint score = 0;
36          if (_result.find("wo:hasHumidity")){
37              score = score + 1;
38          }
39
40          if (_result.find("wo:hasWindDirection")){
41              score = score + 1;
42          }
43
44          if (_result.find("wo:hasWindSpeed")){
45              score = score + 1;
46          }
47
48          if (_result.find("wo:hasCloudCoverage")){
49              score = score + 1;
50          }
51
52          if (_result.find("wo:hasTemperature")){
53              score = score + 1;
54          }
55
56          if (_result.find("wo:hasPrecipitation")){
57              score = score + 1;
58          }
59
60          if (_result.find("wo:hasAtmosphericPressure")){
61              score = score + 1;
62          }
63
64          score = score / 7 * 50000;
```

```
65
66        agent_map[_agent_address].invocation_counter =
67        agent_map[_agent_address].invocation_counter + 1;
68        calculate_payment();
69
70        agent_map[_agent_address].total_score =
71        agent_map[_agent_address].total_score + score;
72
73        user_map[_user_address].invocation_counter =
74        user_map[_user_address].invocation_counter + 1;
75    }
76
77    function join_URL(string memory _URL, string memory _data)
          private pure returns (string memory result ){
78        return string(abi.encodePacked(_URL,_data));
79    }
80    ...
```

Appendix A.2. Solidity registration and transaction

```
1
2     ...
3     uint public minimum_deposit_for_agent = 2 ether;
4     uint public minimum_deposit_for_user =0.2 ether;
5     uint public default_score = 45000;
6     address[] agent_address;
7
8
9     mapping(address=>agent) agent_map;
10    mapping(address=>uint)  user_deposit_map;
11
12
13    struct agent {
14        uint score;
15        uint deposit_balance;
16        uint invocation_count;
17        string URL;
18        bool validity;
19        uint price;
```

```
20      }
21
22
23      function register_as_agent (string memory _URL, uint _price)
24      public payable returns (bool _succeed){
25          if(msg.value >= minimum_deposit_for_agent){
26              if(!agent_map[msg.sender].validity)
27              {   // register the new vendor
28                  agent_map[msg.sender].deposit_balance = msg.value;
29                  agent_map[msg.sender].validity = true;
30                  agent_map[msg.sender].URL = _URL;
31                  agent_map[msg.sender].invocation_count = 0;
32                  agent_map[msg.sender].score = default_score;
33                  agent_map[msg.sender].price = _price;
34                  agent_address.push(msg.sender);
35                  return true; }
36          }
37          return false;
38      }
39
40
41      function register_as_user() public payable returns (bool
            _succeed)
42      {
43          return user_deposit_map[msg.sender] = msg.value;
44      }
45
46      ...
47      uint public commission_charge = 0.01 ether;
48      uint public compensation_charge = 0.1 ether;
49      uint public default_score = 45000;
50      uint public full_payment_score = 40000;
51      uint public half_payment_score = 30000;
52      uint public compensation_score = 20000;
53      ...
54
55      function calculate_payment(uint score, address payable
            _user_address, address payable _agent_address) private{
```

```
56          uint price = agent_map[_agent_address].price;
57          if (score >= full_payment_score) // make full
                make_payment_or_compensation
58          {
59              _payment =  price + commission_charge;
60              deduce_deposit(user_map, _user_address, _payment);
61              _agent_address.transfer(price);
62
63          }
64          else if (score >= half_payment_score){
65              _payment =  price/2 - commission_charge;
66              deduce_deposit(user_map, _user_address, _payment);
67              _agent_address.transfer(price/2);
68
69          }else{
70              _payment =  compensation_charge - commission_charge;
71              deduce_deposit(agent_map, _user_address, _payment);
72              _user_address.transfer(compensation_charge);
73          }
74      }
75
76      function deduce_deposit(mapping _map, address _payer_address,
            uint _payment) private {
77          _map[_payer_address].deposit_balance =  _map[_payer_address
                ].deposit_balance - _payment;
78      }
79
80      function get_all_agents_address() public view returns(address[]
             memory _vendors_address)
81      {  return agent_address; }
82
83      function get_agent_record(address _agent_address)
84      public view returns(uint _score, uint _invocation, bool
            _validity, uint _price){
85          return (agent_map[_agent_address].score,
86          agent_map[_agent_address].invocation_count,
87          agent_map[_agent_address].validity, agent_map[
                _agent_address].price);
```

```
88      }
89
90      function top_up_deposit() public payable{
91          agent_map[msg.sender].deposit_balance =
92          agent_map[msg.sender].deposit_balance + msg.value;
93      }
```