Autonomous Economic Agents as a Second Layer Technology for Blockchains: Framework Introduction and Use-case Demonstration

David Minarsch*,Seyed Ali Hosseini*, Marco Favorito* and Jonathan Ward*
*Fetch.ai
Cambridge, UK
Email: {david.minarsch, ali.hosseini, marco.favorito, jonathan.ward}@fetch.ai

***Abstract*—The user experience of interacting with distributed ledger technologies (DLT) is fraught with excessive complexity, high risk and unintuitive processes. Moreover, smart contracts deployed in these systems are restricted to being reactive. These limitations have negative implications on user adoption and prevent DLTs from being general purpose. We introduce a framework for the development of *Autonomous Economic Agents* (AEAs), software agents that act autonomously and pursue an economic goal, and demonstrate how AEAs complement existing decentralised ledgers as a second layer technology. In particular, the framework enables a simplified user experience through automation, supports modularisation and reuse of complex decision making and machine learning capabilities, and allows for proactive behaviour facilitating autonomy. We demonstrate these gains in the context of a specific use-case, a multi-agent trading system modelling a *Walrasian Exchange Economy* populated by a number of agents trading a basket of tokens.**

I. Introduction

a) Motivation: The rise of Web 3.0 [1] and distributed ledger technologies (DLT) [2] has enabled trustless value exchange and public execution of smart contracts. However, one mass-adoption challenge these systems face today is that their usage often involves excessively many steps to reach the conclusion of an action or trade, thereby exacerbating the requirements on the user to provide input.

Furthermore, utilising and interacting with today's blockchains and DLT-based applications is difficult for humans. Users find themselves having to go through a complex, often unintuitive process and dealing with long alpha-numeric values (e.g. addresses, keys) just to interact with a simple smart contract.

Interactions with blockchains and smart contracts are typically of a high risk nature (e.g. financial) which if erroneous will result in severely undesirable outcomes (e.g. loss of assets). This makes the difficulty of interacting with these technologies even more consequential.

This difficulty is magnified when we go beyond simple transaction settlement scenarios and move towards environments involving frequent, fast, perhaps interdependent, transactions.[1] The dynamic environment in which blockchains reside implies that outcomes of individuals' actions are affected by the actions taken by other individuals. For instance, pricing for transactions is largely determined by congestion [3]. Individuals are bound to make imperfect decisions due to the speed at which the system operates and the information richness it contains.

[1]This is further exacerbated by the fact that the user has to deal with two concepts of time (i.e. block time and clock time), which can make it very challenging for humans to time their actions correctly.

One technical limitation of smart contracts is their purely reactive nature. Smart contracts enable the creation and execution of potentially complex business logic. However, for this logic to execute, the contract needs an external trigger. In other words, an event outside of the logic that resides in a contract must happen, in order for the contract to execute. This significantly restricts the functionality provided by smart contracts to reactive systems. As such, smart contract technology alone cannot be employed to create applications with proactive behaviour [4].

b) Contribution: In this paper, we propose a technology, Autonomous Economic Agents (AEAs) which addresses the above problems by creating a novel second layer of agency on top of the existing benefits and functionalities provided by blockchains and smart contracts. This technology is facilitated by a framework that enables the development of AEAs. It utilises the base layer and complements the base layer's shortcomings.

AEAs are defined as *intelligent agents operating on an owner's behalf, with limited or no interference from its owner, and whose goal is to generate economic value*. This implies, AEAs are not just any agents. They have an express purpose to generate economic value and would usually be deployed in adversarial and multi-stake holder scenarios with competing incentives between the different actors. AEAs are not to be equated with artificial general intelligence (AGI). They can have a very narrow goal-directed focus involving some economic gain and implemented via simple conditional logic.

The AEA framework allows for the creation of AEAs that automate almost all interactions with blockchains and smart contracts. A user (e.g. individual or company) can be represented by an AEA which handles the low level interactions with blockchains and smart contracts at run-time. This simplifies the user experience and such automation is especially suited for applications where well-defined continuous interactions with a blockchain over a long period of time is expected. In addition, the AEA framework allows for complex decision making and machine learning capabilities to be modularised and re-used off-chain.

The many abstraction layers offered by AEAs along with their automation of interactions with ledgers and smart contracts enable the creation of large scale, complex applications. An example of such an application is the trading agent competition (TAC) which we developed, and through which we demonstrate the main claims regarding the contributions of the AEA framework.

c) Related Work: One of the key value-driven aims of the AEA framework is the integration of two very relevant technologies with diverse backgrounds: multi-agent systems [5] and distributed ledger technology [6].

Multi-agent systems have been the subject of development in the research community for more than twenty years [7]. During this time, many frameworks have been proposed for the creation of agents [8] and multi-agent systems [9]–[14]. However, these systems have seen little usage outside of the research community and remained mostly as research projects.

The absence of a native digital financial system is a common characteristic amongst existing multi-agent frameworks, a crucial feature which, arguably, could be the reason for their lacklustre adoption in the industry, especially given the role of economic activities in multi-agent systems [5]. This sets the AEA framework apart and in fact is one of the main motivations driving its conception.

d) Structure: The remainder of the paper is structured as follows. In section II we provide a description of the framework architecture as well as the developer tools to build AEAs. In section III we follow with an introduction to a specific use case, i.e. the TAC, which demonstrates the relevance of the framework to real world problems and substantiates our claims regarding the framework's usefulness. We show the results of the AEAs' interaction in the specific use case and discuss it. We conclude in section IV.

II. AEA Framework

The AEA framework aims to enable rapid, scalable and modular agent development and allows individuals and businesses to easily and quickly compose software that can create economic value for them. The framework attempts to make agent development as straightforward as it is to develop web apps with popular web frameworks. To this end, we chose the Python programming language for the development of the first instantiation of the framework as it is arguably a popular and accessible technology, with a vivid community around it. All agent to agent communication is designed to be agnostic to the language in which the framework is implemented. The framework is open-source and published on GitHub [15] and PyPI [16].

A. Introduction to the Framework

The framework is divided into two parts: a core developed by the authors, and extensions (i.e. packages) developed by other developers.[2] This allows for a modular and scalable agent framework.

AEAs are aimed to inhabit environments which are adversarial and multi-stakeholder in nature. Therefore, AEAs' interactions are message-based and asynchronous. Currently there are four types of framework-specific packages which can be added to an AEA: *Connections*, *Protocols*, *Contracts*, and *Skills*. The framework places no restrictions on the usage of readily available PyPI[3] packages.

1) Connections and Protocols: Connections wrap APIs/SDKs to services, or entire services that are external to the agent. As such they provide an interface to the outside world. A connection in an AEA is responsible for providing the translation between the message-based communication language defined for use within the framework and an external language or application protocol. An example is the *HTTP client connection*. Once added to an AEA, the connection accepts messages from the AEA, translates them into HTTP requests and sends them as a HTTP client to a target URL. The received response is then translated back into a framework specific message and is handed over to the AEA. Another important example of a Connection are *LedgerApis* which allow AEAs to utilise different DTLs.

Every AEA is equipped with its own *Multiplexer* which calls and manages the AEA's Connections. A Multiplexer is capable of handling several Connections at once, allowing an AEA to connect to different environments. To be able to communicate, AEAs use messages wrapped in *Envelopes* which act as an "outer" Agent Communication Language (ACL) [17] encapsulating other ACLs. An envelope references a specific ACL, referred to as *Protocol*, and the message field contains the serialised message that conforms to the protocol. This setup guarantees that all AEAs can communicate with each other on the envelope level *via* a standardised format. However, they can only decode the content of a message if they have an implementation of the protocol. By adopting this two-layered approach to communication, we make the framework compatible with any message based language format which in turn increases interoperability. This enables any existing message-based agent architecture to be connected to the AEA framework via a simple translator - implemented in a Connection - that encodes and decodes message envelopes.

Envelopes are the core objects agents use to communicate. An AEA receives Envelopes through its Connections. Connections deposit Envelopes in the Multiplexer's *InBox* while an AEA can deposit messages to be sent by Connections in the Multiplexer's *OutBox*. InBox and OutBox are simple thread-safe queues managed by the Multiplexer. They allow for the separation of the Multiplexer loop from the loop running the main agent.

2) Skills: Skills deliver economic value to the AEA by allowing agents to encapsulate and execute any arbitrarily complex logic. As such, a skill could wrap simple conditional logic or an advanced deep learning model. To help with Skill development and make the framework scalable, the framework provides a stable API and a number of abstract base classes for developers to extend. The two most important of these are *Handler* and *Behaviour*.

A Handler is registered against a particular protocol and receives messages for this protocol. As such, handlers are a reactive component of the agent. An example of code that

[2]We will publish, in due course, a companion paper which introduces the core framework components and architecture in full detail.

[3]https://pypi.org

might go into a handler is: an action that the agent directly performs upon receiving a specific message (e.g. sending a proposal in response to a call for proposal), or update to a model that the agent is using (e.g. a price model) which is drawn upon by the agent's proactive component. A Behaviour then embodies an agent's proactiveness [5]. It is a scheduled action that is triggered by the agent's internal processes to further its goals rather than by an external trigger (e.g. search for a trading partner on a regular interval). The framework offers several types of Behaviour classes that encapsulate popular patterns of execution. Some of the examples include, *Cyclic* behaviours that execute repeatedly, *One-Shot* behaviours which are executed only once, *Sequential* behaviours composed of sequentially executed sub-behaviours, and *FSM* behaviours which execute their child behaviours according to a Finite State Machine.

Note, it is required that message handling, internal state updates, and behaviour executions are fast enough so they fit into the agent's execution cycle (more on this in Section II-A5). This requirement is imposed to prevent skill components from blocking each other.

For this reason, CPU bound long running tasks inside skills must be implemented as *Tasks* which are executed by a task manager in a separate thread. As such, Tasks differ from behaviours primarily in their execution times. For instance, machine learning and other long running CPU-bound tasks, excluding I/O bound logic, can be executed in the task manager to avoid the reactiveness and proactiveness of the agent to be compromised.

Lastly note that Skills act as black boxes to each other. They can exchange state via a thread-safe key-value store where required, otherwise no other interaction is intended between them.

3) Contracts: Contract modules provide containers for APIs to interact with smart contracts. For instance, an ERC20 contract bytecode and ABI (Application Binary Interface) can be wrapped in a contract module and complemented by an API to interact with the ABI. This API might implement common business logic for interaction with the ERC20 contract, such as deployment, creation and minting of tokens as well as transfer. The contract module can then be defined as a dependency for skills which require access to its high-level functionality.

4) Wallet and Decision Maker: The goals and preferences of an agent, specified by the user, are managed by the *Decision Maker*. It is the only object capable of updating the agent's ownership state by signing transactions, and hence provides a final selection on the proposed transactions by the skills. It is responsible for the crypto-economic safety and the only object with access to the agent's wallet. All AEAs must have a *Wallet* which contains the private-public key pairs that the agent holds.

The role of the decision maker is limited to the evaluation of internal messages from skills. Thus, the decision maker does not directly interact with other agents. This limitation in the decision maker's scope is intended to balance the need of a mediator between competing skills with the risk of the decision maker becoming a bottleneck.

5) Execution: Every AEA has a *main execution thread* which is broken down into three stages/footnoteThe framework implements multiple execution modes, here we present the synchronous agent loop.:

1) *Setup*: sets up all registered resources (i.e. connections, protocols, contracts and skills) by calling their associated `setup()` method.
2) The main loop then executes the following steps on each tick:
   - `react()`: calls the associated handler of every Envelope waiting in the 'InBox' queue.
   - `act()`: calls the `act()` function of every registered Behaviour.
   - `update()`: enqueues scheduled tasks for execution with the Task Manager and executes the decision maker.
3) *Teardown*: tears down all registered resources by calling their associated `teardown()` method.

An AEA's main loop and task loop run synchronously in independent threads. Note the execution time of `react()` and `act()` calls are limited to avoid individual skills blocking each other indefinitely.

In addition to a main loop, an AEA's Multiplexer has an event loop running in a separate thread which processes incoming and outgoing messages across several connections asynchronously.

B. Developer Tools

The framework is readily installed from PyPI. A command line interface is provided which can interact with the local file system and a remote registry to compose AEA projects. The user has the option to install the components of the AEA step-by-step. The framework also comes with a test framework allowing developers to write end-to-end tests for their AEAs.

III. AEA use-case: Trading Agent Competition (TAC)

In this section, we introduce a Trading Agent Competition (TAC), accompanied by its open source implementation as a package in the AEA framework. The competition acts as a testing ground for our claims regarding the contributions of the framework, as described in Section I. As such, it serves as a demonstration of the values that can be brought forward by the framework in a fairly general, simple, and relevant use-case.

After a brief introduction in Section III-A, we describe key aspects of the competition in Section III-B, starting with a formalisation of the competition's underlying economic model, followed up with explanations of the role of an ERC1155 smart contract in transaction settlement via atomic swaps, the various phases in a competition, the trustlessness property, and the competition's network environment.

Then in Section III-C, we introduce two specific types of AEAs which have been developed by the authors.[4] In

[4]Of course TAC being a framework allows for other agent strategies to be designed and entered in a competition.

particular, we describe their trading strategies and how they negotiate in a competition.

Next, in Section III-D, we present the result of the competitions we ran with default settings involving AEAs of the two types mentioned above. This is followed up in Section III-E by a discussion of how claims we made in Section I about the values of the framework as a second layer technology over blockchains are highlighted in the TAC use-case and corroborated in practice via the competitions ran.

A. Introduction

There is a rich history of trading agent competitions, often under the TAC name (see [18]), but this has no direct affiliation with any of those projects.

The scenario of the TAC in the AEA framework is focused on one of the most fundamental forms of economic interactions; bilateral trades. The competition involves a society of agents, each starting with a number of goods, in which agents could engage in one-to-one negotiations and trades using a numeraire token, i.e. *money*, as their medium of exchange.

Each agent, in the real world, would represent an individual or a group of people and is tasked with looking after their interest by maximising their utility. In order to be able to do that, the agents should be made aware of their owners' preferences [19] and values [20]. In the competition, this is simulated by explicitly giving each agent a representation of their owners' preferences over goods at the beginning of each round. During the competition, the goal of each agent is to maximise its owners' interests by engaging in as many profitable trades as possible, of course taking into account their preferences.

The trading agents competition package has applicability in multiple areas. It can be used as a multi-agent game, for instance to trade crypto-tokens on the Ethereum blockchain [21]. It can also serve as a tool to study the effects of different types of agent strategies on trading performance [22].

B. Model

1) The economy: We describe our agent economy, a Walrasian exchange economy [23].

(**Agents**) There exists a set A of agents. Section III-C describes the agents in more detail. There is also a unique agent c, called the *controller* agent, which initialises and manages the registration for the competition, and due to its special role, is not included in A.

(**Goods**) There is a tuple of n sets of goods $X = \langle X_1, \dots, X_n \rangle$ where each $i \in \{1, \dots, n\}$ represents one type of good (e.g. a fungible token) and each element of the set X_i is an instance of that good type (e.g. equivalent instances of the same fungible token). In general, we may have $|X_i| \neq |X_j|$ for any two $i, j \in \{1, \dots, n\}$ where $i \neq j$; that is, the number of available good instances could be different for two different goods. As a consequence, we may in general have differing aggregate supply across goods.

(**Money**) There is a special type of good (e.g. another fungible token), indexed by 0, the *numeraire good* or money. It serves as a unit of account and medium of exchange to agents.

(**Endowments**) Agents are provided with *endowments* in goods and money. That is, each agent $a \in A$ has a good endowment $e^a = \langle e^a_1, \dots, e^a_n \rangle$, a tuple of length n, where the set $e^a_i \subseteq X_i$ is the endowment of agent a in good i. Each agent is given at least some $base\_amount > 0$ of each good, thus for any $i \in \{1, \dots, n\}$, we have $base\_amount \leq |e^a_i| \leq |X_i|$. We assume that good endowments are allocated such that for each $i \in \{1, \dots, n\}$ we have $\bigcup_{a \in A} e^a_i = X_i$; that is, all good endowments for a good sum to the total instances of that good. Finally, every agent is endowed with the same $e^a_0 = money\_amount > 0$. Goods and money can only be traded in integer amounts, that is they are indivisible.

(**Current holdings**) Each agent a's *current good holdings* are denoted by $\mathbf{x}^a = \langle x^a_1, \dots, x^a_n \rangle$ where for any $i \in \{1, \dots, n\}$, $x^a_i \subseteq X_i$ is the set of instances of good i that agent a currently possesses. Therefore, $|x^a_i| \in \{0, \dots, |X_i|\}$; note how agents can have no instance of a good i at some point in time but trivially never a negative amount. That is, there is no borrowing of goods. We further assume that for each $i \in \{1, \dots, n\}$ we have $\bigcup_{a \in A} x^a_i = X_i$; that is, all agents' current good holdings of a given good sum to the total instances of that good available. At the beginning of a round in the competition, before any trades take place, the good holding of any agent is equivalent to its endowment. Furthermore, each agent a's *current money holding* is denoted by x^a_0 and it must always be the case that $\sum_{a \in A} x^a_0 = |A| \times money\_amount$ and for each agent a, $x^a_0 \geq 0$, that means there is also no borrowing of money.

(**Preferences**) Agents are assigned *preferences* for goods and money by the controller agent c. Specifically, each agent $a \in A$ has a preference relation $\preccurlyeq_a$ on goods which totally ranks any possible combination of good bundles. Preferences are assumed to be transitive, thus for any three arbitrary goods x, y and z, $x \preccurlyeq_a y$ and $y \preccurlyeq_a z$ means $x \preccurlyeq_a z$. In practical terms, each agent a has a utility function u^a which is quasi-linear in goods and money:

$$u^a(x^a_0, \mathbf{x}^a) = x^a_0 + g(\mathbf{x}^a) = x^a_0 + \sum_{i \in \{1, \dots, n\}} s^a_i \times f(|x^a_i|) \quad (1)$$

such that $s^a_i > 0$ and

$$f(|x^a_i|) = \begin{cases} \ln(|x^a_i|) & \text{if } |x^a_i| > 0 \\ -L & \text{otherwise} \end{cases} \quad (2)$$

Breaking down the utility function in Equation 1, x^a_0 is agent a's money holding, s^a_i is the utility parameter agent a assigns to good i, and $f(|x^a_i|)$ parameterizes the number of instances of good i that agent a has. The logarithmic nature of $f(\cdot)$ implies decreasing returns; i.e. the agent values acquiring each additional instance of a good less than acquiring the previous instance. $L > 0$ is a large constant.

An agent a's utility parameters for goods are represented by $s^a = \langle s^a_1, \dots, s^a_n \rangle$ where s^a_i is the utility parameter agent a assigns to good i. Equation 3 ensures the sum of the utility parameters for all goods are the same for every agent a:

$$\sum_{i \in \{1, \dots, n\}} s^a_i = 1 \quad (3)$$

With the specific design of the utility function in Equation 1, we are implementing the well studied Cobb-Douglas function [24] for $g(\mathbf{x}^a)$ which has the gross-substitutes property. This property means that an increase in the price of one good causes agents to demand more of the other goods.

In the context of the competition, the utility function serves as the metric which the agents have to maximise. The agent which best achieves this goal becomes the winner of the competition.

(**Trade cost**) Settling trades in DLT-based financial systems is costly, in particular transactions incur transaction fees. As transactions are settled against a blockchain in real time, they incur variable transaction costs $k(t)$ determined by the blockchain.

2) ERC1155 Smart Contract: Trading multiple goods (i.e. tokens) on the same ledger can be implemented in multiple ways. For efficiency and cost reasons, we run the TAC utilising a smart contract [25]. In particular, we use the ERC1155 Smart Contract standard which allows for multiple ERC20 (fungible) and ERC720 (non-fungible) tokens to be maintained in one smart contract. It furthermore makes atomic swaps between two agents, involving sets of tokens feasible in a single transaction. We represent goods as fungible tokens, as different instances of the same good are considered identical in the preferences of the agents. We represent money as a separate fungible token.[5]

3) Trading game phases: The trading game has three phases: a pre-trading phase, the trading phase and a post-trading phase.

In the pre-trading phase, agents wanting to participate in the competition register with the controller agent. Only if a min_agent number of agents register, then the competition moves to the trading phase. The controller agent uses the pre-trading phase to deploy the ERC1155 smart contract, create the fungible tokens (i.e. goods and money) for the competition and mint the good and money endowments in tokens for the registered trading agents.

The trading phase consists of k game instances $g_1, \ldots, g_k$, with pauses in between, where at each game instance g:

- For every participating agent, the controller agent generates endowments in good $\mathbf{e}^a$ and money $\mathbf{e}_0^a$, and preferences $\mathbf{s}^a$, ensuring that they are somewhat orthogonal. This is implemented through a *generator* algorithm in the game controller which acts as the random seed to each game instance. The controller then sends these along with contract addresses to every agent in the competition.
- Agents trade with each other. When a pair of agents arrive at mutually beneficial terms of trade, they conduct an atomic swap [27].[6] This means, the first agent cryptographically signs the terms of trade and the second agent then submits both the terms of trade as well as its counter-party's signature in the form of a transaction to the smart contract. If the feasibility constraints listed in *current holdings* in Section III-B1 are met and the signatures are referencing the submitted terms, then the smart contract permits the atomic swap and the goods change ownership. Transactions with invalid signatures or incompatible terms of trade are rejected.
- After some set period of time, the game instance finishes and the controller agent constructs a league table for this game instance from the on-chain data.

In the post-trading phase, the controller agent reports the final league table by computing a (weighted) average across all game instance league tables.

4) Trustlessness: The trading environment in this competition is designed to be trustless. This system allows agents to trade with one another without being required to take trust into consideration, as trust is guaranteed by the design of the system itself. Specifically, the ERC1155 smart contract ensures that every transaction sent to it is signed by all parties involved in the transaction. Because the ERC1155 smart contract deployed on a permissionless blockchain is the only entity with the competition's global state, this makes it impossible for agents to submit invalid transactions according to current holdings, e.g. submitting transactions in which they spend/give away more money/good than what they actually have.

5) Network Environment: The agents participating in the competition communicate over an overlay network utilising the Internet. The overlay network consists of special *OEF node(s)*, a competition manager – implemented via a controller agent – and a set of participating agents. The OEF nodes come with specific functionalities. One such functionality is message relaying. This means that through appropriate APIs, agents send and receive messages to each other, and the OEF handles the low level delivery of the messages over the overlay network. Another functionality is service registration and search. Agents can register services (e.g. goods to sell/buy) and search for registered services.

The registration and search process of services works as follows. An agent first generates a description of the services it offers in terms of data models. It then registers the service on the *OEF search and discovery* node. An agent can search for the services of all agents who are registered on any *OEF search and discovery* node. The agent does so by formulating a query in a custom query language and sending this query to the *OEF search and discovery* node. Once the service search request has been submitted to the node, the node processes it and returns a list of agents which match the query to the requesting agent.

C. Agents

Agents in the competition focus on discovering how they can arrive at an optimal bundle of goods and money - as defined by their preferences - through successive trades. Emphasis for agents is placed on a) finding the right agents to trade with, and b) doing the right trade with them. This might involve identifying other agents' needs to arrive at the optimal

[5]The native cryptocurrency [26] of the target blockchain - here Ethereum - could be used as money. However, for the competition this implies the controller needs access to a large amount of ETH funds which makes this more restrictive in practice.

[6]Note the swaps between agents' assets happen on the same chain and have the atomicity property as defined in [28], [29].

trading sequence. However, a dummy agent which simply selects random bargaining partners and trades if it is beneficial to it, without any longer term plan is still able to improve its score in the competition.

1) Strategy: Recall each agent a has a utility function $u^a(\mathbf{x}^a, x_0^a)$, which given a good holding $\mathbf{x}^a$ and money holding x_0^a, gives a number representing "how good agent a's current situation is". The utility number is used by the agent to compare different states and helps in decision making. For instance, when deciding how much better off the agent would be if it acquired another copy of a good i. This is equivalent to comparing the utility values u^a of the two holdings $\langle x_1^a, \ldots, x_i^a, \ldots, x_n^a \rangle$ and $\langle x_1^a, \ldots, \tilde{x}_i^a, \ldots, x_n^a \rangle$ where $|\tilde{x}_i^a| = |x_i^a| + 1$.

In general, an agent a compares two states with good and money holdings of respectively $\mathbf{x}^a, x_0^a$ and $\tilde{\mathbf{x}}^a, \tilde{x_0}^a$, by calculating the *marginal utility* $u^a(\tilde{x}^a, \tilde{m}^a)$ - $u^a(\mathbf{x}^a, x_0^a) = g(\tilde{x}^a) - g(\mathbf{x}^a) + \tilde{x_0}^a - \mathbf{x}^a$. This is the quantity by which the agent assesses whether an exchange resulting in its good and money holdings to change from $\mathbf{x}^a$ to $\tilde{x}^a$, respectively x_0^a to $\tilde{m}^a$, pays off. In particular, for the exchange to pay off, the change in the marginal utility has to be positive.

Each agent a trades as follows. For each good i in her current holding $\langle x_1^a, \ldots, x_n^a \rangle$, the agent simultaneously offers to buy an instance, and sell an instance if she has at least two. The price p the agent is willing to sell the instance for is *equal or more than* the agent's marginal utility for i in the goods component only, i.e. $p \geq g(\mathbf{x}^a) - g(\tilde{\mathbf{x}}^a)$, where $\tilde{x}^a$ is the current good holding $\mathbf{x}^a$ minus the instance to be sold. On the other hand, the price p' the agent is willing to pay to acquire an instance is *equal or less* than the agent's marginal utility for i in the goods component only, i.e. $p \leq g(\tilde{x}^a) - g(\mathbf{x}^a)$, where $\tilde{\mathbf{x}}^a$ is the current good holding $\mathbf{x}^a$ plus the instance of the good to be bought.

With the above trading strategy, the agent will never make any profit in negotiations where it proposes a price. It can only make a profit in negotiations where is receives a proposal.

In a companion paper we compare the baseline strategy with a model-based trading strategy. A model-based agent uses information it gains from acceptances and declines of her proposals to create a price model for each good. It then uses the price model to offer the price it assumes has the highest likelihood to becoming a successful trade. The comparison of the results is beyond the scope of this paper.

2) Negotiation: The negotiation protocol agents use in the competition is inspired by the FIPA ACL [30]. This means, there is a multi-step dialogue during which agents send messages of the form $\mathrm{P}(c_1, \ldots, c_n)$ where P, called a *performative*, conveys the type of the message and $c_1, \ldots, c_n$ are the contents (e.g. Request(*resource*), Propose(*offer*, *price*)). Table I lists the allowed messages and for each message specifies its valid replies.

An agent A initiates a negotiation by sending a CFP (call-for-proposal) to their counter-party B. If B processes a CFP, it replies with either a Propose or a Decline. Decline sent at any point in the negotiation by anyone terminates the negotiation.

A Propose contains a list of offers, which partially or totally match the query delivered in the CFP. Once agent A receives the offers, it will either Accept or Decline. Upon acceptance, A will respond with an 'Accept' message to B which contains a cryptographic signature of the terms of trade. The negotiation is successfully terminated if B as the counter-party follows with a matching Accept. This involves, first submitting a signed transaction, containing the cryptographic signature of A as well as the terms of trade to the ERC1155 smart contract and second forwarding the transaction digest to agent A in a *Match-Accept* message. However, B might also Decline at this stage, which would be the case if its trading position has changed between when it made the proposal and when it receives A's Accept. Both A and B can use the transaction digest to verify if and when the transaction has been settled on the ledger.

TABLE I. NEGOTIATION MESSAGES

Message	Contents	Replies
$\mathit{cfp}(q)$	$q : \mathit{query}$	$\mathit{propose}(o,p)$ $\mathit{decline}()$
$\mathit{propose}(o,p)$	$o : \mathit{offer}$ $p : \mathit{price}$	$\mathit{propose}(o',p')$ $\mathit{accept}()$
$\mathit{accept}()$		$\mathit{match-accept}()$
$\mathit{decline}()$		
$\mathit{match\text{-}accept}()$		

The second Accept is used, as without it, B would have to sign the terms of each proposal it is making, which is very limiting. This is because at this point there is a high probability that the negotiation breaks down, because other simultaneous trades/negotiations impacting this one might be more beneficial. The reason for this is that any signed terms of trade and transaction submitted to the ledger must be considered in the *forward looking* state of the agent, i.e. the state the agent finds itself in once all the trades it committed to have been settled by the controller agent c.

D. Simulation Results

In this section we briefly discuss some simulation results. We have run a simulation with 10 agents. Half of the agents are baseline agents, the other half are model-based. We ran the simulation 103 times, each time with a different seed and otherwise identical configuration (details can be found in [31], [32]).

We first compare the initial and final scores of baseline agents. To avoid the initialisation from affecting the comparisons, we look at the difference of final and initial scores relative to the theoretical equilibrium score (if agents behaved as in the Walrasian Exchange economy). The null hypothesis we test for is that mean of the difference between the final scores less equilibrium scores (i.e. adjusted final score) is weakly smaller than the mean of the difference between initial scores less equilibrium scores (i.e. adjusted initial score). The alternate hypothesis is that the baseline agents have a strictly larger mean of the adjusted final scores than the mean of the adjusted initial scores less equilibrium scores. A one-sided t-test allows us to reject the null hypothesis at a significance level $\alpha = 0.0001$. Figure 1 shows the distribution of adjusted

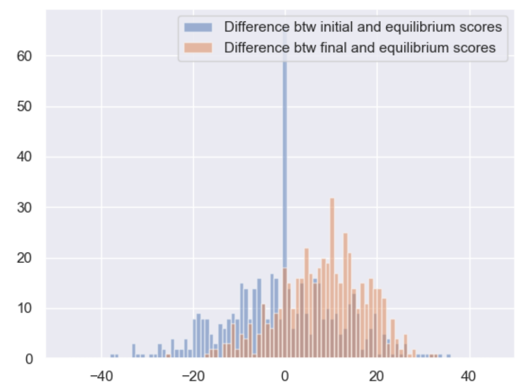

Fig. 1. The y-axis shows the count of observations in each bin, the x-axis shows mean the difference (split into bins).

scores for the two groups of agents. The mean adjusted final score is 8.251 whilst the mean adjusted initial score is -0.51.

The analysis shows that on average, agents significantly (both statistically and economically) improve their score irrespective of their starting point.

We next look at the number of trades completed. The mean number of seller transactions (transactions where the agent is in the role of the seller) of the baseline agents is 10.72 whilst the mean number of buyer transactions of the baseline agents is 10.80. Since the competition ran for a maximum of 300 seconds in each instance, that implies that every agent performed slightly more than one transaction every 15 seconds.

E. Discussion

In this section we highlight the main contributions of the AEA technology in the context of the TAC. We discuss how AEAs solve the problems, outlined in Section I, that we currently face when using blockchain and smart contract technologies.

The first challenge for humans is the difficulty in using and interacting with these technologies, such as dealing with private keys and multi-step processes. From TAC, it is easy to see how automation of low level interactions with blockchains and smart contracts offered by the AEA framework eliminates these difficulties. The automation thus results in an increase in accessibility and a decrease in the complexity of interacting with DLTs.

Despite having very simple strategies, agents in the competition manage to perform an average of almost 20 transactions in the span of 5 minutes (see Section III-D), without making any of the mistakes a human might make under similar conditions. Furthermore, the results presented in Section III-D on the average score gains of agents participating in TAC is witness to the effectiveness of the transactions that agents execute in the competition.

The contrast in how humans and AEAs interact with DLT is magnified in the TAC for two reasons. First, the competition is larger in scale, both in time and number of transactions, than most typical human interactions with blockchains. Second, despite being trivial in construction, the competition is complex in participation if we take into account the environment's non-cooperative nature, the negotiation process, and the inter-dependency of transactions. In particular, humans would struggle to maintain multiple parallel negotiations and use information conveyed in the proposals they receive to build a model of aggregate demand and supply for individual goods.

The TAC scenario creates a high risk environment whereby making any mistakes either in interacting with the ledger or other higher level issues (e.g. pricing) would result in loss of assets and thus ranking by the agents. In theory, a mistake in only a single transaction could potentially result in the loss of all tokens, having an especially detrimental effect on an agent's score if it had a favourable initial position. AEAs navigate this risk well in the TAC and avoid erratic actions when interacting with DLTs.

AEAs add a proactive layer on top of a purely reactive environment offered by smart contracts. In particular, all of the operations related to search, discovery and advertisements that agents in a TAC perform cannot be achieve in a purely smart contract based setup.

Moreover, even though most of the logic that goes into an AEA's reactive Handler component, in particular establishment of the terms of trade, could in practice be implemented in a smart contract, this:

a) would get complicated rapidly because of all the layers of abstractions an AEA provides which would be missing in a pure smart contract solution. In particular, when the agent becomes more advanced employing, say machine learning techniques, this approach would reach its limits.
b) would be against the ethos of blockchains as we would be dumping a lot of transient information (e.g. states of whole negotiations) which are unnecessary to keep in an immutable, integration preserving data structure;
c) would be too expensive to maintain transient states on a blockchain.

AEAs coupled with DLTs allow for the optimal allocation of off-chain and on-chain tasks, thereby utilising the relative strengths of each system. The efficiency and private execution in AEAs can be coupled with public execution and trust-minimising properties of smart contracts. This also has privacy benefits [33] and naturally integrates well with state channel concepts [34].

IV. Conclusion

In this work, we propose a framework for the development of AEAs, a novel multi-agent based second layer technology, that complements common limitations of DLT base layer solutions. The main foci are on accessibility and complexity reduction through automation, modularisation of the agent software components, combining benefits of on-chain and off-chain environments, and support for proactive capabilities on top of smart contract functionality.

We have demonstrated the capabilities of the framework in a multi-agent trading scenario. The benefits include automated interactions with DLTs in a high-risk and complex environment that is larger in scale than typical human interactions with blockchains, reusable libraries to interact with blockchains and smart contracts, search and discovery functionalities enabled due to AEAs' proactiveness, and the ability to move computation and transient data off-chain.

The framework is under active development. We have developed AEAs targeting other interesting use cases, such as, a data marketplace for training machine learning models, automating supply chain interactions, as well as initial efforts in constructing AEA-based solutions for the mobility sector. As future work, we aim to develop more powerful preference representation and elicitation methods for the Decision Maker so that we get closer to our prescribed vision: AEAs representing humans and generating economic value on their behalf.

As for the TAC, we consider it both a suitable application in which to evaluate the effectiveness of AEAs' latest features and demonstrate their full capabilities. But we also see TAC as a standalone environment whereby the performance of trading strategies within different market configurations could be evaluated. There are many ways the competition could be further developed. Below, are two of the most immediate proposals we are considering:

- **Richer strategies**: we can develop strategies based on a variety of techniques, including more advanced reinforcement learning (RL) algorithms than the one currently employed by TAC's model-based agents and evolutionary/genetic algorithms.
- **Competing strategies**: agents can use multiple skills implementing different strategies simultaneously. Due to the heterogeneity of the TAC environment, it is likely that no single type of agent strategy is superior against all types of opponents.

ACKNOWLEDGEMENT

We thank Fetch.ai for supporting this research work and the release of the open-source implementation.

REFERENCES

[1] V. Kashyap, C. Bussler, and M. Moran, *The Semantic Web: Semantics for Data and Services on the Web*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[2] R. Wattenhofer, *Distributed Ledger Technology: The Science of the Blockchain*, 2nd ed. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2017.

[3] G. Huberman, J. Leshno, and C. C. Moallemi. The economics of the bitcoin payment system. [Online]. Available: https://voxeu.org/article/economics-bitcoin-payment-system

[4] S. D. Levi, A. B. Lipton, S. Arps, Slate, and M. . F. LLP. An introduction to smart contracts and their potential and inherent limitations. [Online]. Available: https://corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations

[5] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Wiley, June 2009.

[6] R. Maull, P. Godsiff, C. Mulligan, A. Brown, and B. Kewell, "Distributed ledger technology: Applications and implications," *Strategic Change*, vol. 26, no. 5, pp. 481–489, 2017.

[7] V. Lesser, Ed., *First international Conference on Multiagent Systems*. The AAAI Press, 1995.

[8] K. Kravari and N. Bassiliades, "A survey of agent platforms," *Journal of Artificial Societies and Social Simulation*, vol. 18, 01 2015.

[9] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, *Jade — A Java Agent Development Framework*. Boston, MA: Springer US, 01 2005, pp. 125–147. [Online]. Available: https://doi.org/10.1007/0-387-26350-0_5

[10] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2007.

[11] M. Gregori, J. Palanca, and G. Aranda, "A jabber-based multi-agent system platform," in *Proceedings of the International Conference on Autonomous Agents*, vol. 2006, 01 2006, pp. 1282–1284.

[12] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with jacamo," *Science of Computer Programming*, vol. 78, no. 6, pp. 747 – 761, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016764231100181X

[13] R. Bordini, J. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 10 2007, vol. 8.

[14] A. Grignard, P. Taillandier, B. Gaudou, D. A. Vo, N. Q. Huynh, and A. Drogoul, "Gama 1.6: Advancing the art of complex agent-based modeling and simulation," in *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, G. Boella, E. Elkind, B. T. R. Savarimuthu, F. Dignum, and M. K. Purvis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 117–131.

[15] M. Favorito, D. Minarsch, A. Hosseini, A. Triantafyllidis, D. Campbell, O. Panasevych, K. Chen, Y. Turchenkov, and L. Rahmani, "Autonomous economic agent (aea) framework," 2019. [Online]. Available: https://github.com/fetchai/agents-aea/

[16] F. Limited, "Autonomous economic agent (aea) framework," 2019. [Online]. Available: https://pypi.org/project/aea/

[17] S. Poslad, "Specifying protocols for multi-agent systems interaction," *ACM Trans. Auton. Adapt. Syst.*, vol. 2, no. 4, p. 15–es, Nov. 2007. [Online]. Available: https://doi.org/10.1145/1293731.1293735

[18] M. P. Wellman, A. Greenwald, and P. Stone, *Autonomous Bidding Agents, Strategies and Lessons from the Trading Agent Competition*, ser. Intelligent Robotics and Autonomous Agents series. The MIT Press, 2007.

[19] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, ser. Princeton Classic Editions. Princeton University Press, 1944.

[20] K. Atkinson and T. Bench-Capon, "States, goals and values: Revisiting practical reasoning," *Argument & Computation*, vol. 7, no. 2 - 3, pp. 135 – 154, November 2016.

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

[22] G. Kendall and Y. Su, "The co-evolution of trading strategies in a multi-agent based simulated stock market through the integration of individual learning and social learning," *Proceedings of IEEE*, pp. 2298–2305, 2003.

[23] A. Marshall, *Principles of economics: unabridged eighth edition*. Cosimo, Inc., 2009.

[24] M. Brown, *The New Palgrave Dictionary of Economics*. Palgrave Macmillan UK, 2017. [Online]. Available: https://link.springer.com/referenceworkentry/10.1057/978-1-349-95121-5_480-2

[25] C. Clack, V. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *arxiv:1608.00771. 2016*, 08 2016.

[26] G. Hileman and M. Rauchs, "Global cryptocurrency benchmarking study," *Cambridge Centre for Alternative Finance*, vol. 33, 2017.

[27] R. van der Meyden, "On the specification and verification of atomic swap smart contracts," 2018.

[28] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, ser. PODC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 245–254. [Online]. Available: https://doi.org/10.1145/3212734.3212736

[29] W. Radomski, A. Cooke, J. Castonguay, Philippe andTherien, E. Binet, and R. Sandford, "Eip 1155: Erc-1155 multi token standard," Ethereum, Standard, June 2018. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1155

[30] I. F. S. Committee, “Communicative act library specification,” Foundation for Intelligent Physical Agents, Tech. Rep., August 2001.

[31] Fetch.ai, “Agents tac,” https://github.com/fetchai/agents-tac, 2019.

[32] D. Minarsch, M. Favorito, A. Hosseini, and J. Ward, “Trading agent competition with autonomous economic agents,” in *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS ’20. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2020, p. 2107–2110.

[33] G. Avarikioti, E. Kokoris-Kogias, and R. Wattenhofer, “Brick: Asynchronous state channels,” 2019.

[34] P. McCorry, C. Buckland, S. Bakshi, K. Wüst, and A. Miller, “You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies,” in *Financial Cryptography and Data Security*, A. Bracciali, J. Clark, F. Pintore, P. B. Rønne, and M. Sala, Eds. Cham: Springer International Publishing, 2020, pp. 35–49.