Autonomous Economic Agent Framework

David Minarsch[1], Marco Favorito[1,2], Seyed Ali Hosseini[1], Yuri Turchenkov[1], and Jonathan Ward[1]

[1] Fetch.ai, UK
[2] Sapienza University of Rome, Italy
{david.minarsch, marco.favorito, ali.hosseini, yuri.turchenkov, jonathan.ward}@fetch.ai

Abstract. The Internet and the services delivered via it are increasingly centralised on a few monopolistic platforms. Today's web frameworks are conceived to cater for increasing returns to scale and winner-takes-all business models with a built-in asymmetry between users and services. Existing multi-agent and agent architectures have seen no significant adoption outside niche applications. We propose a novel agent framework which is designed to allow for a decentralised digital economy to manifest where each individual and organisation is represented by an autonomous economic entity with its own agency. The framework bridges the old and new web and employs distributed ledger technologies as core parts of its construction. We introduce the framework, discuss the performance characteristics of its current implementation and demonstrate several application areas.

Keywords: Autonomous Economic Agents · Agent Framework · Multi-agent System Framework · Distributed Ledger Technology.

1 Introduction

1.1 Web 2.0 and its short-comings

Today's digital services are highly centralised. By some estimates [15], over 43% of web-traffic volume incorporates one of the *FAANMG*[3] platforms. In the process, they extract a significant amount of rent [41] indicating low competition.

The increasing monopolisation is partially caused by unsuited or outdated regulation [42]. However, it is argued that the design of the Web 2.0 [16] itself contributes to this outcome. In particular, the dominant client-server architecture favours a centralised ownership of servers (monolithic or micro-service implementation) [4], causes a lack of interoperability [6], and leads to centralisation of economic control and data [5].

[3] Facebook, Apple, Amazon, Netflix, Microsoft and Google.

1.2 Two new trends

With the recent rise of the decentralised Web 3.0[4] [46] and distributed ledger technologies (DLT)[5] [48] - in particular Bitcoin [31] and Ethereum [12] - it is evident that alternative, decentralised systems can be technologically and economically sustained.

We introduce an agent framework for autonomous economic agents (AEAs), which embraces this technological shift and which we demonstrate is capable of allowing multi-stakeholder multi-agent systems (MAS) to finally find wide-spread production deployment. Our novel approach is chiefly enabled by two drivers: the first is the trustless, non-intermediated exchange of wealth and public code execution in smart contracts[6] mediated by DLT, which thereby provides a financial and contracting layer for the MAS [13]; the second, is the readiness of businesses to cooperate on the design of custom on-chain (i.e. DLT enabled) and off-chain (e.g. MAS) stateful protocols that permit industry-wide competition without a winner-takes-all market dynamic [32,7].

1.3 Contribution

The core contribution of the framework is that it enables wide-spread and scalable real-world deployment of multi-stakeholder MAS utilising DLT as evidenced in section 5. This contribution is facilitated by the framework's innovation in five key areas: developer experience, software engineering, artificial intelligence, economics, and user experience.

The main benefit of the framework for the software developer is that it allows them to re-use existing code to a much larger degree than in other agent frameworks (e.g. [8,20]) and client-server oriented web frameworks (e.g. [38,17]). Re-use is not restricted to libraries but extends to application specific components encapsulating subsets of the agent's business logic.

An actor-like framework design leads to software components that are loosely coupled, allowing for concurrency without requiring shared state which enables additional complexity to be incorporated by adding modules to the agent. Interaction between components occurs mostly via asynchronous message passing. This provides a consistent and scalable approach to communication within and across agents. Hence, an AEA can itself be viewed as a small MAS.

From the AI perspective, a developer is offered the flexibility to combine different approaches, such as reinforcement learning [44], deep learning [19] and symbolic AI approaches [21] in one framework.

[4] As discussed in [25] not to be confused with the Semantic Web [24].

[5] A distributed ledger is a consensus of replicated, shared, and synchronised data where processing nodes are geographically and organisationally - no central control - spread across multiple entities. Bitcoin network is a permissionless DLT in the form of a blockchain, with proof of work as the consensus algorithm and Bitcoin as the cryptocurrency.

[6] Smart contracts are computer programmes which are executed by nodes of a DLT, usually a blockchain, and can, similar to objects, hold their own state. They can be used to automate enforcement of contract terms, reduce the need for trusted intermediaries and allow for reuse and encapsulation to create interoperable on-chain protocols like decentralised exchanges [2].

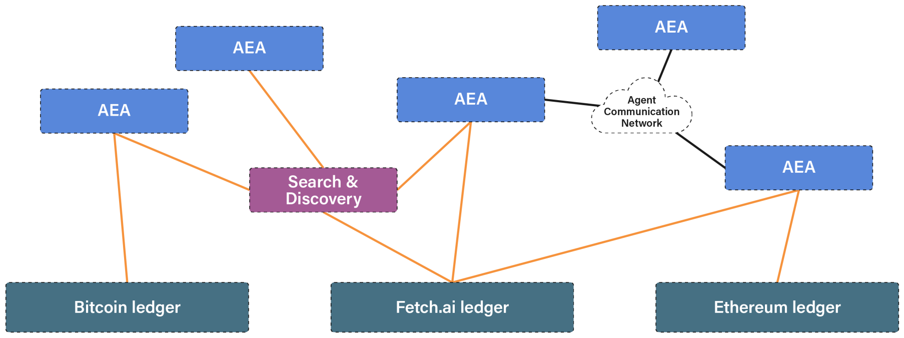

Fig. 1: Diagrammatic representation of an AEA-based MAS. AEAs run off-chain, on heterogeneous devices controlled by their stakeholders. They can use DLTs for settlement and commitments and various (agent-based) services for communication as well as search and discovery.

The native integration with DLT is novel relative to other agent and multi-agent frameworks (e.g. JADE [8], SPADE [20], Jason [10]). It provides a financial settlement and commitment layer, enabling the framework to support deployment of multi-stakeholder systems. In particular, it is possible for anyone to write software for deployment into decentralised and permissionless markets and therefore provides explicit economic benefit to its user and allows for new economic organising principles.

Finally, the framework enables developers to distribute agents as finished products to end-users, lowering barriers to wide-spread adoption of MAS. For instance, through encapsulating complicated interaction flows with DLT and delegation to AEAs, the framework improves the user-friendliness of DLT [28].

2 Definition and Environment Requirements

In reference to [39] we define an *autonomous economic agent (AEA)* as

> an intelligent agent operating on an owner's behalf, with limited or no interference of that ownership entity, and whose goal is to generate economic value for its owner;

where the *economic* aspect is realised through exchange and commitments facilitate primarily by DLT. The literature contains related definitions as *machines that obtain their own agency through being equipped with crypto-currency wallets* [35]. This definition is in so-far helpful as it puts the focus on the wallet which the agent maintains and which provides an explicit financial metric of the agent's economic value.

The preceding definitions imply a set of requirements for the environment an AEA operates in (stylised in figure 1). In particular, AEAs and MAS more generally require:

1. a means to interact with other AEAs, agents and services in a structured way,
2. a delivery mechanism of messages via the Internet,

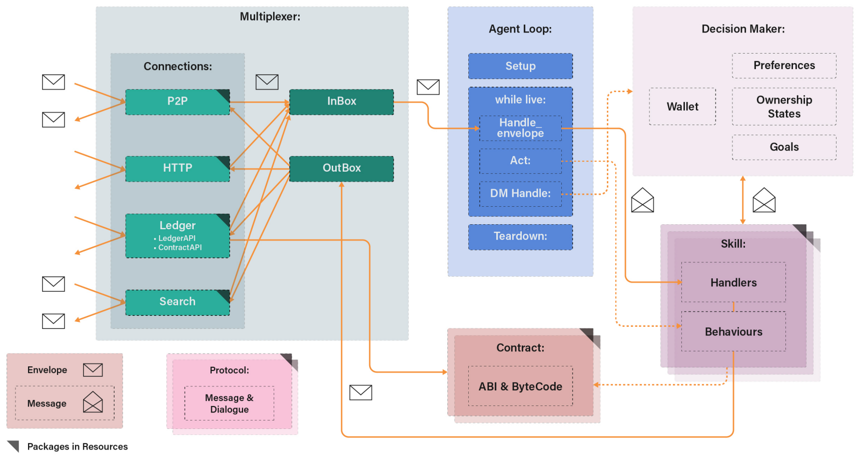

Fig. 2: Simplified illustration of the AEA framework. Connections executed in the Multiplexer receive messages in AEA and third-party protocols. The AgentLoop calls Handlers and Behaviours in Skills based on messages and configurable ticks, respectively. The DecisionMaker manages the (crypto) Wallet. Dark corners indicate (non-core, agent-specific) packages kept in Resources.

3. access to a financial settlement system, and
4. access to a search and discovery system.

The AEA framework makes use of a protocol framework for bilateral dialogue-based interactions to aid 1 [22]. For message delivery (2) AEAs utilise a peer-to-peer permissionless agent communication system [36], which supports arbitrary message-based interaction protocols (1). Currently, a custom centralised search and discovery system for agents is used to satisfy 4. Thanks to the modular nature of AEAs this can easily be replaced with a fully decentralised alternative in the future. Finally, AEAs—unlike other agent types—form part of the second layer to distributed ledgers [28]. They use ledgers and smart contracts [33] to perform financial transactions and make commitments (3). Crucially, AEAs are not run *on* a ledger (i.e. they are not a smart contract), they are executed on any host with the necessary resources and Internet access.

3 Framework Architecture

The architecture described in this paper is currently implemented as open-source in the Python programming language.[7] However, the framework could be implemented in any object-oriented programming language that supports asynchronous programming. An illustration is provided in figure 2.

[7] The AEA framework's repository can be found at https://github.com/fetchai/agents-aea.

3.1 Actor-like design and modularity

The AEA framework is designed as an actor-like [3] asynchronous message passing system [23]. As such, it allows for a high degree of modularity and components largely communicate via messages.

The framework can be divided into two parts: the core developed by the authors and external contributors, and packages implementing agent-specific business logic. These can be developed by anyone using the framework. There are four types of packages that can be readily added to the framework:

- Skills: primary business logic modules (CPU bound),
- Protocols: messages and dialogue rules,
- Connections: networking related (I/O bound) logic and translations between AEA and third-party protocols,
- Contracts: wrappers for smart contract logic.

Furthermore, the framework allows straightforward inclusion of additional APIs to third-party DLTs via framework plugins and supports the use of readily available software packages in the target programming language.

From the perspective of the framework, packages consist of code and configuration. The framework loads the specified packages and then places them with respect to each other and executes them where appropriate (cf. inversion of control). Before we explore the packages in turn, we discuss the core framework components.

3.2 Core components of an AEA

The central framework class is the *AEA* class which houses three core components: *Resources*, *Runtime* and *Wallet*.

Runtime and Resources Resources is a collection of packages available to the AEA. In particular, it contains the Skills, Protocols, Connections and Contracts the AEA uses in code form as well as their configurations. Resources acts as a registry for code to be executed by the Runtime. Packages in Resources are immutable. However, additional packages can be dynamically added at runtime. Dynamically added packages are removed from Resources once the AEA tears down.

The Runtime is responsible for executing the code in the packages. It consists of three abstractions:

- a *Multiplexer* executes the Connection packages,
- an *AgentLoop* executes Skill packages, as well as the *DecisionMaker*, a unique type of Skill discussed below, and
- an optionally enabled *TaskManager* executes long running and CPU bound tasks.

The Runtime deals with two types of primitive concepts: scheduled tasks and events or messages. All messages or events are processed atomically by the AEA components.

The Multiplexer and the Connections that it contains continuously listen for events on pipes, sockets and queues. External communication arriving at one of the AEA's

Connections is, where necessary translated to framework-specific messages. For instance, HTTP requests are translated to messages in a HTTP Protocol. The Multiplexer then passes these messages to an AEA-internal queue, the *InBox*, for processing by the AgentLoop. Similarly, the Multiplexer continuously monitors another AEA-internal queue, the *OutBox*, for outgoing messages and passes those to the relevant Connection for processing.

The AgentLoop is responsible for proactive execution of periodic tasks like Behaviours (discussed in detail below) in Skills, and processing removal and addition of new components, as well as reacting to new messages appearing in the InBox and handling them with a corresponding Handler in Skills. It is the responsibility of the AgentLoop to fetch the appropriate component responsible for processing a given message from the Resources and passing the message to the component.

A feature that arises from the framework's implementation in Python is that the Runtime can be configured in two modes: the *threaded* mode, where the AgentLoop and the Multiplexer are run in their own threads (i.e. the tasks scheduled by a given component are cooperatively scheduled in the same thread) , and the *async* mode, where all the tasks scheduled by the components are run asynchronously on a single event loop. This allows the developer to configure the trade-off between cooperative multitasking used in the event loop implementation (async mode) and pre-emptive scheduling used for thread scheduling (threaded mode) [43].

Wallet The Wallet is a simple data structure. It contains the private keys of the AEA, and therefore allows for the public key and address to be computed and for the agent to append digital signatures to transactions.

3.3 Packages

The four packages make up the core mechanism via which the AEA is extensible and composable by design.

Communication (Connections and Protocols) The Protocol and Connection packages enable AEAs to communicate with other AEAs as well as internally.

Connections wrap APIs or SDKs to in-process services or services external to the agent like a user interface, a peer for inter-agent communication or a DLT. They can be though of as both the sensors and the actuators of an AEA, as they provide an interface to the outside world. A Connection is responsible for providing the translation between framework communication languages (see Protocols below) and external languages, if needed. A Connection can be developed by anyone and the base Connection class defines a stable interface to the Multiplexer. The interface consists of four primary methods: ‘connect’ and ‘disconnect’, ‘send’ for sending via the Connection and ‘receive’ for receiving via it.

To communicate with each other and for communication between AEA components including Skills and Connections, AEAs use *Envelopes* which act as an outer agent communication language (ACL) [34] wrapping specific ACLs (cf. [18]) . An Envelope has five fields:

- *sender* and *to*: the Address[8] of the sender/receiver;
- *protocol_id*: the identifier of the Protocol used;
- *message*: a bytes field for the serialized message;
- *context*: an optional field for routing, containing a URI.

The ‘protocol_id‘ references a specific ACL or other language, and the ‘message‘ field contains the serialized *Message* in that *Protocol* (cf. interaction protocol [45]), for instance FIPA [14] ACL. This setup guarantees that all AEAs can communicate with each other on the Envelope level via a standard format. However, they can only decode the content of a message if they have an implementation of the Protocol. For the delivery of the Envelope various (third-party) protocols and services can be used (e.g. [36]). By adopting this layered approach to communication we avoid reinventing the wheel: any existing message-based agent architecture could be connected by simply writing a translator that encodes/decodes an Envelope and still ensures interoperability. We also make it possible to have a consistent language inside the framework whilst being compatible with external changes.

The Protocol framework is adopted from [22]. Along with the Message class, which deals with representation, and the *Serializer* class which deals with decoding and encoding, a Protocol specifies a set of *Rules* over the message sequence.

Skills Skills implement the business logic of an agent. They allow encapsulation of (almost) any kind of code and are reusable across AEAs.

Skills are made up of three core abstractions:

- a *Handler* is responsible for handling messages in a registered Protocol, thereby implementing the AEA’s reactive behaviour. Each Handler is responsible for a single Protocol, but can send messages of any type of Protocol. The AgentLoop calls a ‘handle’ method and passes it the message as it appears in the InBox.
- a *Behaviour* encapsulates actions resulting from internal logic rather than as direct reactions to messages. They implement the proactiveness of the agent. Behaviours come in different types (e.g. cyclic/one-shot/finite-state-machine/etc. [28,8]) and are scheduled tasks from the perspective of the Runtime. The AgentLoop calls a defined ‘act’ method in the behaviour at the time specified in the Skill configuration.
- a *Model* is a data class. It is used to maintain shared state within a Skill.

With these abstractions[9], and the ability to call arbitrary code from them, Skills can implement logic ranging from very basic to extremely advanced. As such they might wrap simple conditional logic to complex deep learning models.

The framework does restrict the execution time of calls to both ‘act’ and ‘handle’. For CPU-bound and long-running logic (e.g. machine learning and other AI workloads), a *Task* can be created and submitted to a thread- or subprocess-based TaskManager.

[8] AEAs use *Addresses* for identification and for communication purposes. The Address is derived from the public key of a public-private key pair generated from the elliptic curve as specified by, for instance, the standard SECP256k1 [11].

[9] An analogy to the Model-View-Controller architecture prevalent in many web frameworks can be observed: Handlers have similarities to Controllers, and Messages can be considered the equivalent to Views.

A Skill shares state via the SkillContext which is accessible from any Handler, Behaviour and Model in the Skill. Additionally, SkillContext, modules within a Skill have read access to a limited number of objects exposed on the AgentContext, which contains agent-specific information, such as its public keys and addresses, and utilities, such as the OutBox for messages. Importantly, however, a Skill does not have access to the context of another Skill and it also does not have access to the agent's Wallet. To interact with other components, specifically Skills, Connections and the DecisionMaker, the Skill needs to use messages.

Contracts Contracts wrap smart contracts [47] for third-party distributed ledgers. In particular, they provide wrappers around the application binary interface (ABI) of a smart contract. They expose an API that is compatible with the framework to abstract away the implementation details of the ABI from the agent's Skills.

Contracts usually contain the logic to create smart contract transactions and make smart contract calls. As such they require network access to the underlying ledger. Contract packages are therefore executed in Connections. Skills communicate via messages with the Contract.

3.4 Economic control

DecisionMaker The DecisionMaker is a specialised type of Skill and the only component in the AEA with access to the Wallet.

The role of the DecisionMaker is limited to considering internal messages from Skills and making economically relevant and safe decisions. It does not directly interact with other AEAs. Instead, it mediates the competing Skills and restricts their capabilities.

The goals and preferences of an agent are managed by the DecisionMaker. It is the only object capable of updating the agent's ownership state (as represented on-chain in the form of tokens or off-chain) and preferences (cf. utility function) by signing transactions, and hence accepts or rejects the Skills' proposed transactions.

The framework provides a basic reference implementation of a DecisionMaker with a closed form representation of preferences, ownership and goals which is internally closed. The developer is free to extend it to their needs.

Crypto and Ledger APIs The DecisionMaker makes heavy use of *Crypto APIs* and—via Skills and Connections—of *Ledger APIs*. Framework-side, these consists of abstract classes: the former defines a set of abstract methods to create and handle DLT identities (i.e. public/private key pair for specific ledgers) as well as signing transactions, whilst the latter defines the abstract methods to interact with the ledger (e.g. get the current balance, send a transaction etc.). Crypto APIs are stored in the Wallet, and Ledger APIs can be called through the *Ledger Connection*, a default Connection that acts as access point to any ledger that the application supports. By default, the supported ledgers are Ethereum [12] and Cosmos [26], as well as any compatible ledger architectures. The framework allows for loading new types of Crypto and Ledger APIs types at runtime, through a global shared registry and plugin mechanism, hence achieving a high degree of extensibility and interoperability with other DLT systems.

3.5 Persistence

Certain parts of the agent's state, like for instance completed dialogues and their messages as well as other data accumulated at runtime, should be stored to avoid a continuous growing memory requirement at runtime and to be able to (automatically) recover the state from a crash. The current reference implementation provides an optionally configurable storage backend.

3.6 Dynamic adaption and security

AEAs are designed to dynamically add additional packages at runtime. To ensure integrity of the packages is maintained as they are shared and used in the AEA, the framework deploys a hashing strategy. All code is hashed using IPFS [9] multi-hashes. This ensures that an AEA can verify the integrity of a package at runtime.

3.7 Relationship to other agent architectures

The AEA architecture attempts to combine deliberative and reactive components and can hence be seen as a hybrid agent: Handlers deal with reactive elements as representative of deductive reasoning agents. Behaviours and Tasks can deal with the deliberative elements of the belief-desire-intention (BDI) model and are more generally representative of features found in practical reasoning agents [49].

Furthermore, the AEA framework splits the deliberative and reactive elements into both vertical and horizontal layering. Skills are by default horizontally layered. Each Skill is connected to input (i.e. messages) so several Skills can act on the same input and produce suggestions (i.e. transactions) to the DecisionMaker. This means, Skills effectively compete as they consume the same messages but do not necessarily communicate. Within Skills, Handlers are vertically layered. Each type of input is dealt with at most one layer (one Handler).

The separation into Behaviours, Handlers and Tasks within Skills shows similarity to Turing Machines: Their planning layer is matched by our Behaviours. Their reactive layer is mirrored by our Handlers. Their modelling layer can be seen to relate to either our Behaviours or Tasks [27].

The Java-based JADE multi-agent framework [8] provides similar programming abstractions of ours regarding communication and execution model. The main differences specifically to JADE are: (*i*) each agent lives in its own Java thread, and must be associated to a JADE container (i.e. a Java process); our framework gives more flexibility by letting users run AEAs in different processes, in the same process but on different threads, or in the same process but using asynchronous programming; (*ii*) the proactive and reactive components are conflated into the single Behaviour abstraction, whereas we make a clear distinction by introducing the Handler abstraction; (*iii*) the scheduling of behaviours is cooperative, whereas ours is pre-emptive, either thread-based or asynchronous with configurable timeouts. Being a more mature framework, JADE provides features that the AEA framework does not currently support, e.g. ontology-based content, agent persistence, agent mobility services, and others.

SPADE [20] is a lightweight MAS library for agent development. Its major features are asynchronous execution based on Behaviours, a communication system built on the XMPP protocol [40], and a message dispatching mechanism based on message templates. The AEA framework almost completely covers the SPADE features in a modular and effective way: indeed, the AEA framework gives much more flexibility on the execution model to adopt and the transport layer to use (the XMPP protocol can be implemented as a custom Protocol-Connection pair of an AEA); and the same features of message templates can be achieved by relying on a Handler that based on certain attributes of the message spawns new Behaviours dynamically.

Jason [10] is a Java-based implementation of an extended version of the AgentSpeak programming language [37]. Jason is heavily based on the BDI agent architecture and logic programming, with reasoning cycles "sense-plan-act" that allows agents to evaluate which plans are triggered for execution each time an event occurs. Instead, the agent abstraction of our framework is closer to JADE's and SPADE's (agent loop that executes behaviours and handlers code), hence less declarative and succinct than Jason. This also reflects the different foci: where Jason has strong theoretical foundations and lends itself to implement BDI-type agents, our framework is general purpose (i.e. not tied to a specific agent-type), production-ready and with a focus on wide-spread adoption for DLT enabled consumer and industrial applications.

Unlike any other framework, the AEA framework provides a number of unique features: (*a*) native integration with DLTs for transacting and use of smart contracts, as well as economic control oriented design; (*b*) developers can package and re-use business-level functionality; (*c*) developers can distribute agents as finished products to end-users for deployment, for instance by using the AEA registry (https://aea-registry.fetch.ai) or IPFS [9]. Developers and researchers can leverage existing agent frameworks in the AEA framework by developing Connections and Protocols to bridge them.

4 Benchmark

We demonstrate a number of benchmark results of the Python implementation. These highlight that the framework is capable to serve a significant message load both in the single- and multi-agent per-process case.

All benchmarks use the same resource. A 2.2GHz Intel® Xeon® CPU with 15GB of RAM and 4 cores is used. The benchmarks are run on a freshly provisioned machine. The scripts are available in the project codebase.[10]

4.1 Single-agent: Reactiveness

We first measured the latency (milliseconds) and throughput (Envelopes processed per second) of an AEA, both in 'async' and 'threaded' runtime modes. The AEA has only one Connection and one Skill with a single Handler. The Connection continuously produces an Envelope containing a Message, deposits it in the InBox and waits for a response. The Handler simply echos received Messages back to the sender, the Connection itself. The experiment is run 100 times, and each run lasts for 10 seconds.

[10] Details on reproducability are provided in Appendix A.

Mode	Latency (ms)	Throughput (env/sec)
async	0.526 ± 0.521	1630.32 ± 16.211
threaded	0.800 ± 0.147	1106.762 ± 18.864

(a) Reactiveness

Mode	Throughput (env/sec)
async	6158.846 ± 516.349
threaded	4587.775 ± 820.525

(b) Proactiveness

Table 1: Benchmarks for both 'async' and 'threaded' runtime mode. The experiment is run 100 times, 10 seconds for each run. Latency and throughput are measured in milliseconds and envelopes per second, respectively.

The results shown in Table 1a demonstrate that the AEA is capable of processing in excess of 1630 envelopes per second in the async mode and 1106 in the threaded mode. Moreover, on average, the latency in the async mode is lower than the one in the threaded mode.

4.2 Single-agent: Proactiveness

We next measured the latency and throughput of an AEA with a single Skill implementing a single Behaviour and a single Connection. The Behaviour continuously produces Messages. The Connection simply records receipt of a Message.

The results, reported in Table 1b, show that the AEA is capable to produce 34% more envelopes in async relative to threaded mode.

Given the architecture, it is to be expected that the async runtime mode dominates threaded in both reactive and proactive case. The tasks are well coordinated by the framework, hence cooperative multitasking should not pose a problem and the event loop implementation causes less context switching than threading.

4.3 Multi-agent, single process

We next measure the throughput when multiple agents are connected in a complete network and send each other envelopes. All agents are executed in the same process.[11] The individual agents are run in their own threads and each agent is run with the designated runtime mode. Each agent's OutBox is pre-populated with 100 envelopes which are then continuously circulated by the agents.

The round-trip times (RTT) are comparable for low numbers of AEAs in both runtime modes. For higher numbers of AEAs the async mode shows significantly lower RTT. A similar picture emerges for memory consumption. AEAs, irrespective of their runtime mode, are run in different threads and therefore multi-threading guarantees some form of non-starvation property and relatively equally distributed time slices across AEAs.

[11] The AEA framework is primarily targeting stand-alone AEA deployment matching its primary application as a multi-stakeholder MAS agent framework. This benchmark demonstrates that nevertheless multiple AEAs can be run in a single process.

Agents	RTT (ms)		Memory (mb)	
	async	threaded	async	threaded
2	0.340 ± 0.006	0.339 ± 0.012	54.928 ± 0.236	54.89 ± 0.152
4	2.108 ± 0.033	1.998 ± 0.046	56.715 ± 0.676	56.672 ± 0.617
8	7.935 ± 0.538	8.170 ± 0.371	62.062 ± 0.846	62.447 ± 1.038
16	16.757 ± 1.824	25.966 ± 3.45	76.881 ± 1.868	80.422 ± 2.014

Table 2: Multi-agent benchmark for both ‘async‘ and ‘threaded‘ runtime mode. The experiment is run 100 times, 10 seconds for each run. Initially 100 envelopes are deposited in each AEA’s OutBox.

5 Use Cases

We ourselves and third parties have developed AEAs targeting a number of different use cases with the framework. We provide a short overview to demonstrate the breath in scope:

– **Trading Agent Competition**: [29,30] demonstrate how a population of AEAs can replicate an exchange economy. Each AEA maintains a basket of digital assets and aims to increase its utility by executing bilateral trades autonomously. Agents negotiate using a FIPA-like [14] Protocol and settle trades atomically on a blockchain.
– **Autonomous Supply Chain**: [50] demonstrate the feasibility of an autonomous supply chain using MAS—powered by the AEA framework—and Internet of Things. The scenario showcases a perishable food products supply chain mechanism. Five types of agents were implemented for this demonstration: retailer, wholesaler, supplier, logistics agent, and third-party logistics agent.
– **Agent Worlds 1 to 4**: We have demonstrated the production readiness of the framework and AEAs developed with it in the context of an online and public agent competition which took place in four stages from October 2020 until February 2021. Any participant could download a finished AEA from the registry (https://aea-registry.fetch.ai), connect it via configuration to a public weather or mobility API for various cities across the world, and then run it as a seller of public weather or mobility data. The AEA would register itself for the competition, provided the participant staked a small amount of crypto-currency on a smart contract deployed on the Ethereum blockchain.[12] An AEA created and run by the authors acted as a buyer of this data. At multiple time points throughout the day, it would search for one of the data types in one of the specified cities and then purchase the data from all sellers offering it which were registered for the competition. In excess of 2’000 agents competed over two months and performed a total of more than 110’000 transactions. Each transaction was settled on a Fetch.ai test-net blockchain and resulted in a micro-reward for the participants.
– **Autonomous Option Traders**: [1] use an AEA to maintain a portfolio of put and call options deployed on the Ethereum ledger. The AEA allows the user to manually

[12] This acts as a spam protection and as an incentivisation mechanism.

submit option order requests via a graphical user interface. The AEA translates the HTTP requests into a sequence of actions which ultimately result in the order being executed on the ledger and being stored in local persistent storage. The AEA then monitors the option holdings of the user and exercises them when they are in-the-money (ITM) and due to expire within 5 minutes. This removes the possibility of ITM options expiring worthless and ensures users take profitable positions.

6 Discussion

6.1 Architecture Choices and Limitations

The reference implementation of the AEA framework in the Python programming language imposes an overhead at runtime relative to compiled programming languages. This was a conscious choice at the start of the development cycle to enable rapid iteration and prototyping. It means that in practice running an AEA on devices with less resources than a Raspberry Pi is a challenge. However, as the benchmark demonstrates, environments with a Python interpreter and moderate networking, CPU and storage requirements can make full use of the framework. The listed use cases the framework supports are more than satisfiable with these minimal requirements. Furthermore, with increased adoption, a lightweight AEA library can be implemented in a compiled programming language like Golang.

Over time, we hope to provide, in the core of the framework, generic implementation(s) of the DecisionMaker component which permit increasingly powerful configuration or ‘training’ by the user. For simple use cases, where the utility of an agent can be easily represented and evaluated, this is already possible as we demonstrate in our use cases.

Further limitations are discussed in the framework’s documentation.

6.2 Value add

We identify five innovations which underpin our main contribution discussed in section 1.

From a developer perspective the key benefit the framework provides are its modularity and composability. Unlike for other web and agent frameworks the componentisation and reusability extends to arbitrary application-specific business logic through the encapsulation in Skills. This flexibility does not necessarily lower the integrity of the system as the package hashing strategies deployed ensure that components are uniquely identified and tamper-proof.

From a software engineering perspective, all the software components are loosely coupled and an actor-like design approach maintained. Only a minimal amount of shared state exists and interactions between components is almost entirely via asynchronous message passing.

The generality of Skills presents another core contribution of the framework design. In particular, it means the agent framework does not prescribe the type of AI tools used, it is agnostic to whether the developer uses a deep learning model, reinforcement learning or traditional AI approaches.

The openness in the design approach is also maintained for Protocols. Unlike other agent framework we do not prescribe usage of a particular application or agent protocol (e.g. FIPA), instead developers have access to a generic protocol framework which can be adjusted to the relevant use case.

One of the biggest differentiators relative to existing agent and multi-agent frameworks is the native integration with distributed ledgers and the associated crypto-economic security concepts. This allows AEAs to be fully autonomous economic entities with an ability to transact and make commitments. It also allows arbitrary coordination mechanisms to be implemented.

The other big differentiator is offered on the user experience: agents can be distributed as finished products to end-users. This enables developers to share arbitrary agent-based solutions directly with their user base.

7 Conclusion

The AEA framework presented in this paper is the only production-ready framework we are aware of that unifies MAS and DLT. It is designed and built for production and in parallel to the development of real-world applications by ourselves and third parties. Arguably, several aspects of the framework, in particular the DecisionMaker, are underdeveloped. However, in the spirit of open-source development and to grow a community of AEA developers and researchers we believe it is crucial to start with an initial, useful implementation and iterate from there. We welcome contributions to the framework design and its implementation.

A Experiments

In this section, we provide instructions to reproduce the experiments.

A.1 Requirements

The framework can be used on any major platform (GNU/Linux, macOS, Windows). However, to run the benchmark, we suggest using UNIX-like systems (e.g. GNU/Linux or macOS).

Make sure your platform has the following software installed and the associated binaries accessible from the system path of your operating system:

– Python 3.8. This can be downloaded from here: https://www.python.org/downloads/release/python-380/.
– Make sure you have Pip installed: https://pip.pypa.io/en/stable/installing/. Also, the script requires that the CLI tool `pip` should point to `pip3`. Note that on some platforms this is not the default configuration.
– Git. This can be downloaded from here: https://git-scm.com/downloads.

A.2 Steps to reproduce the experiments

- Download the following script in your working directory to reproduce results: https://raw.githubusercontent.com/fetchai/agents-aea/v0.10.1/benchmark/run_from_branch.sh. (Alternatively, for the latest version use: https://raw.githubusercontent.com/fetchai/agents-aea/main/benchmark/run_from_branch.sh.)
- Assign execution permissions to the script. For example, on UNIX systems:
```
> chmod u+x run_from_benchmark.sh
```
- Run the script:
```
> ./run_from_benchmark.sh
```

References

1. 8ball030: Autonomous hegician (2020), https://github.com/8ball030/AutonomousHegician
2. Adams, H., Zinsmeister, N., Salem, M., Keefer, R., Robinson, D.: Uniswap v3 core. Tech. rep., Uniswap (2021)
3. Agha, G.A.: Actors: A model of concurrent computation in dist. systems. Tech. rep., MIT (1985)
4. Arkko, J.: The influence of internet architecture on centralised versus distributed internet services. Journal of Cyber Policy (2020)
5. Baker, D.: The internet is broken (2017), https://www.wired.co.uk/article/is-the-internet-broken-how-to-fix-it
6. Basaure, A., Vesselkov, A., Töyli, J.: Internet of things (IoT) platform competition: Consumer switching versus provider multihoming. Technovation (2020)
7. Beck, R., Müller-Bloch, C.: Blockchain as radical innovation: a framework for engaging with distributed ledgers as incumbent organization. In: HICSS (2017)
8. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing multi-agent systems with JADE. John Wiley & Sons (2007)
9. Benet, J.: IPFS - Content Addressed, Versioned, P2P File System. arXiv:1407.3561 (2014)
10. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley (2007)
11. Brown, D.R.: Sec 2: Recommended elliptic curve domain parameters. SEC (2010)
12. Buterin, V.: Ethereum Whitepaper (2013), https://ethereum.org
13. Calvaresi, D., et al.: Multi-agent systems and blockchain: Results from a systematic literature review. In: PAAMS (2018)
14. Committee, I.F.S.: Communicative act library specification. Tech. rep., Foundation for Intelligent Physical Agents (2001)
15. Cullen, C.: Over 43% of the internet is consumed by Netflix, Google, Amazon, Facebook, Microsoft, and Apple: Global Internet Phenomena Spotlight (2019), https://www.sandvine.com/blog/netflix-vs.-google-vs.-amazon-vs.-facebook-vs.-microsoft-vs.-apple-traffic-share-of-internet-brands-global-internet-phenomena-spotlight
16. DiNucci, D.: Fragmented future. Print **53** (1999)
17. Django Software Foundation: Django (2005), https://djangoproject.com
18. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an Agent Communication Language. In: CIKM (1994)
19. Goodfellow, I., Bengio, Y., Courville, A., Bengio, Y.: Deep learning. MIT press Cambridge (2016)
20. Gregori, M., Palanca, J., Aranda, G.: A jabber-based multi-agent system platform. In: ICAA (2006)

21. Haugeland, J.: Artificial intelligence: The very idea. MIT press (1989)
22. Hosseini, S.A., Minarsch, D., Favorito, M.: A practical framework for general dialogue-based bilateral interactions. In: Engineering Multi-Agent Systems. (to publish) (2021)
23. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: a comparative analysis. In: PPPJ (2009)
24. Kashyap, V., Bussler, C., Moran, M.: The Semantic Web: Semantics for Data and Services on the Web. Springer (2008)
25. Khoshafian, S.: Can the real web 3.0 please stand up? (2021), https://www.rtinsights.com/can-the-real-web-3-0-please-stand-up/
26. Kwon, J., Buchman, E.: Cosmos: A network of distributed ledgers (2016)
27. Lizán, F., Maestre, C.: Intelligent buildings: Foundation for intelligent physical agents. IJERA (2017)
28. Minarsch, D., Hosseini, S.A., Favorito, M., Ward, J.: Autonomous economic agents as a second layer technology for blockchains: Framework introduction and use-case demonstration. In: 2020 Crypto Valley Conference on Blockchain Technology (CVCBT). pp. 27–35 (2020)
29. Minarsch, D., Favorito, M., Hosseini, A., Ward, J.: Trading agent competition with autonomous economic agents. In: AAMAS. IFAAMAS, Auckland, New Zealand (2020)
30. Minarsch, D., Hosseini, S.A., Favorito, M., Ward, J.: Trading agent competition with autonomous economic agents. In: ICAART. vol. 13, pp. 574–582 (2021)
31. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep., Manubot (2019)
32. Poddey, A., Scharmann, N.: On the importance of system-view centric validation for the design and operation of a crypto-based digital economy. arXiv:1908.08675 (2019)
33. Poncibò, C., Di Matteo, L., Cannarsa, M., et al.: The Cambridge HB of smart contracts, blockchain tech. and digital platforms. Cambridge Univ. Press (2019)
34. Poslad, S.: Specifying Protocols for MAS Interaction. ACM Trans. Auton. Adapt. Syst. (2007)
35. Pschetz, L., Speed, C.: Autonomous economic agents. In: Living in the Internet of Things (IoT 2019) (2019)
36. Rahmani, L., Minarsch, D., Ward, J.: Peer-to-peer autonomous agent communication network. In: AAMAS. p. 1037–1045. AAMAS '21, IFAAMAS, Richland, SC (2021)
37. Rao, A.S.: Agentspeak (l): Bdi agents speak out in a logical computable language. In: European workshop on modelling autonomous agents in a multi-agent world. pp. 42–55 (1996)
38. Ruby on Rails: Ruby on rails (2004), https://rubyonrails.org
39. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson (2020)
40. Saint-Andre, P., Smith, K., Tronçon, R., Troncon, R.: XMPP: the definitive guide. " O'Reilly Media, Inc." (2009)
41. Sandbu, M.: The economics of big tech (2018), https://www.ft.com/economics-of-big-tech
42. Sandbu, M.: Fixing the internet's broken markets (2018), https://www.ft.com/content/5fbc2848-17c2-11e8-9376-4a6390addb44
43. Silberschatz, A., Galvin, P., Gagne, G.: Operating system concepts. Wiley (2018)
44. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
45. Torroni, P., et al.: Modelling interactions via commitments and expectations. In: Handbook of research on multi-agent systems: Semantics and dynamics of organizational models. IGI Global (2009)
46. Voshmgir, S.: Token Economy: How the Web3 reinvents the Internet. BlockchainHub (2020)
47. Wang, S., et al.: Blockchain-enabled smart contracts: architecture, applications, and future trends. IEEE Transactions on Systems, Man, and Cybernetics: Systems (2019)
48. Wattenhofer, R.: Distributed Ledger Technology: The Science of the Blockchain. CreateSpace Independent Publishing Platform (2017)
49. Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley (2009)
50. Xu, L., Brintrup, A., Minaricova, M.: Autonomous supply chains (2020), https://vimeo.com/438586015/bf50c7bc9