INTERNATIONAL TELECOMMUNICATION UNION

TELECOMMUNICATION STANDARDIZATION SECTOR

STUDY PERIOD 2022-2024

FOCUS GROUP ON AUTONOMOUS NETWORKS (FG-AN)

AN-I-XXX

Original: English

Question(s): N/A

Virtual, 2022

Decentralized Controller Evolution Architecture Including an Integrated Marketplace

Source: Rakuten Mobile,
University of Glasgow

Title: Team "Digital Twins" Final Report, Build-a-thon 2022

Contact:	Jaime Fúster de la Fuente Rakuten Mobile	Email : jaimefusterf@gmail.com
Contact:	Álvaro Pendás Recondo Rakuten Mobile	Email : alvaropr97@gmail.com
Contact:	Paul Harvey (Mentor) University of Glasgow	Email : paul.harvey@glasgow.ac.uk

Keywords: Build-a-thon, proof of concept, evolution, blockchain, IPFS, Ethereum, close-loop

Abstract: This contribution is the Final Report of the team "Digital Twins" for the Focus Group on Autonomous Networks (FGAN) Build-a-thon (BT) 2022. We present a Proof of Concept (PoC) of the decentralized controller evolution architecture for Autonomous Networks (AN) using a distributed Marketplace. Such a marketplace serves as a decentralized mechanism for the secure and traceable evolution of controllers, which is achieved by a private Ethereum blockchain and a distributed and decentralized filesystem, the Interplanetary Filesystem (IPFS).

ACRONYMS

- Artificial Intelligence (AI)
- Autonomous Network (AN)
- Build-a-thon (BT)
- Deep Neural Network (DNN)
- Digital Twin (DT)
- Focus Group on Autonomous Networks (FGAN)
- Graph Neural Network (GNN)
- Hypertext Transfer Protocol (HTTP)
- InterPlanetary File System (IPFS)
- International Telecommunication Union (ITU)
- JavaScript Object Format (JSON)
- Machine Learning (ML)
- Proof of Concept (PoC)
- Representational State Transfer (REST)
- Content Identifier (CID)
- Distributed Ledger Technology (DLT)
- OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)
- Web Server Gateway Interface (WSGI)

INDEX

1. INTRODUCTION

The Focus Group on Autonomous Networks (FGAN) input document FGAN-I-239 [1] describes the 2022 FGAN Build-a-thon (BT) as a challenge open to the public where different teams work to develop a Proof of Concept (PoC) related to the FGAN list of use cases (FGAN-O-013 [2]). The BT is aligned with the International Telecommunication Union (ITU) Artificial Intelligence (AI)/Machine Learning (ML) in 5G Challenge.

This document is a report of the team "Digital Twins" contribution to the BT, which code can be found in our Github repository [3]. We have implemented a PoC for the distributed autonomous evolution of controllers based on the FGAN architecture (FGAN-I-198 [4]). We specifically address important issues, such as the security, auditability, explainability and scalability of the process. Additionally, the PoC also includes the subsequent automatic deployment of any controller obtained from the evolution. This last part contains the TOSCA parsing of a Representational State Transfer (REST) service for the chosen controller.

The remainder of this contribution is organized as follows. Section 2 describes the problem statement and how it is related to the use cases list. Section 3 explains the implementation of the PoC, including the used tools. Section 4 goes over the results of the PoC. Finally, conclusion and future work are both contained in Section 5. The contribution source code can be found in the GitHub repository [3], along with a detailed explanation for the deployment of the PoC demo. The repository also includes the Neo4j code for generating the graphs shown in this document.

2. PROBLEM STATEMENT

This section explains the problems and challenges that this contribution aims to solve, as well as their relevance for the development of an Autonomous Network (AN). Section 2.1 shows the architecture for the evolution process. Section 2.2 explains how the protocol InterPlanetary File System (IPFS) and blockchain are combined for building a Marketplace for controllers. Finally, Section 2.3 describes the list of FGAN use cases for which this contribution is valuable.

2.1.DISTRIBUTED EVOLUTION OF CONTROLLERS FOR AN

The document FGAN-I-198 [4] describes a High-Level Architecture Framework for AN, including evolutionary-driven networks. This architecture is shown in Figure 1. A module is "a building block consisting of executable code and a module specification from which controllers are assembled". A module's specification includes its input and output interfaces, and a metadata description of its functionality. Examples of possible modules include aggregation functions, DNS configuration interfaces, an entire deep neural network (DNN) model, a single layer of a DNN model, etc. On the other hand, a controller can be considered as a software closed-loop composed of modules. A controller instance is "an executable representation of a controller including modules, their configurations, and parameter values". Exploratory evolution tries to find a suitable controller for a required use case specification using evolutionary algorithms.

This contribution describes an end-to-end structural PoC based on the FGAN architecture and on the secure and traceable evolution of controllers. The evolutionary algorithms that would drive the process and the reactive evolution of controllers to operational conditions are beyond the scope of this work.

Figure 2 shows the high-level Neo4J graph representation for the PoC closed-loop, which uses all of the concepts presented in Figure 1, except for the Selection and Operation Controller. Note that for the Real Time Online Experimentation the use of a Digital Twin (DT) is considered. A DT, which concept is described in FGAN-I-058 [5], is "a mathematical representation of a physical and/or logical object". For example, a DT could be a Graph Neural Network (GNN) that reproduces the behaviour

of a network, producing outputs such as latency or packet loss rate, given inputs that describe its state and policy. This PoC uses the concept of a DT in the architecture, but does not focus on its implementation, which is a problem out of the scope of this project. Section 3 explains further on this topic.

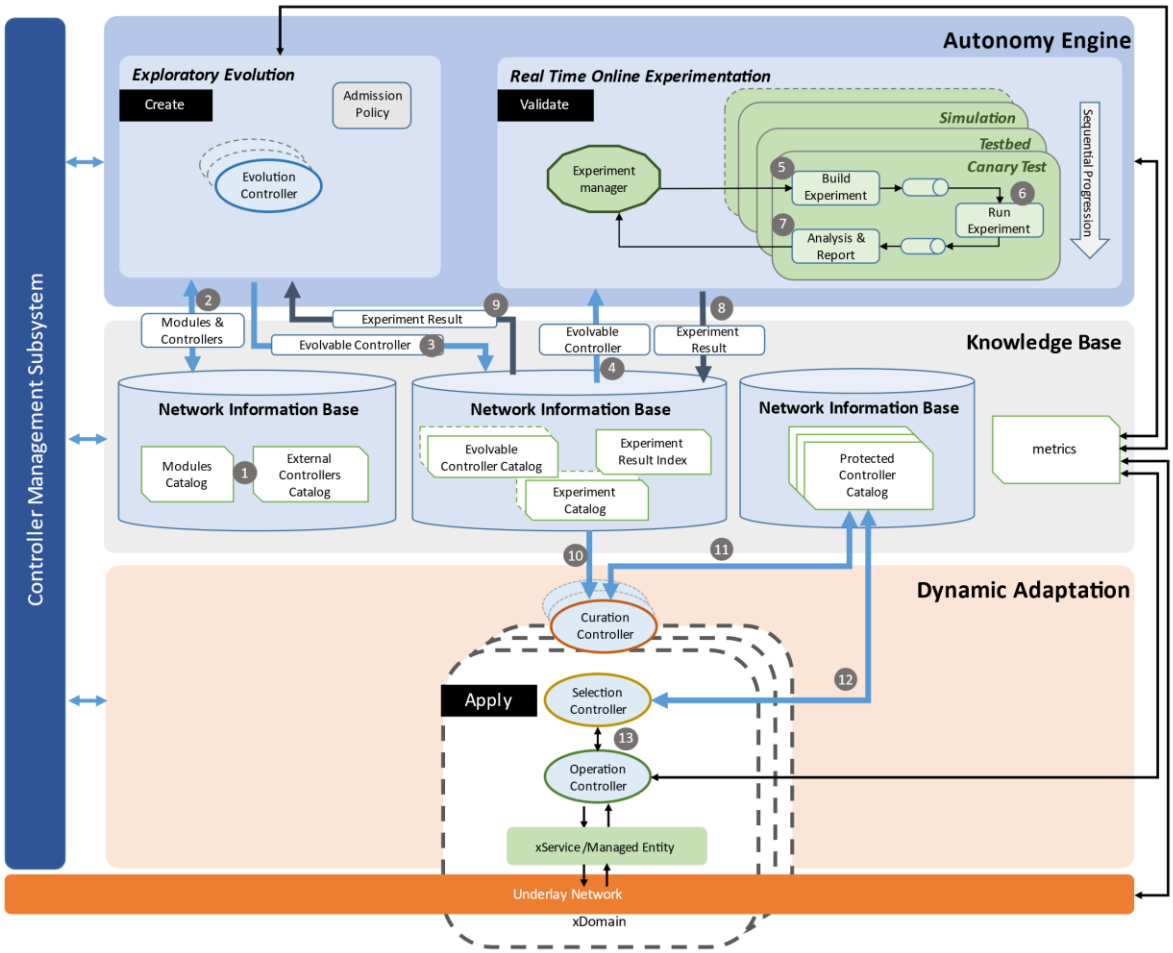

Figure 1: High-Level Framework for Evolutionary-Driven Autonomous Network. Source: FGAN-I-198 [4].

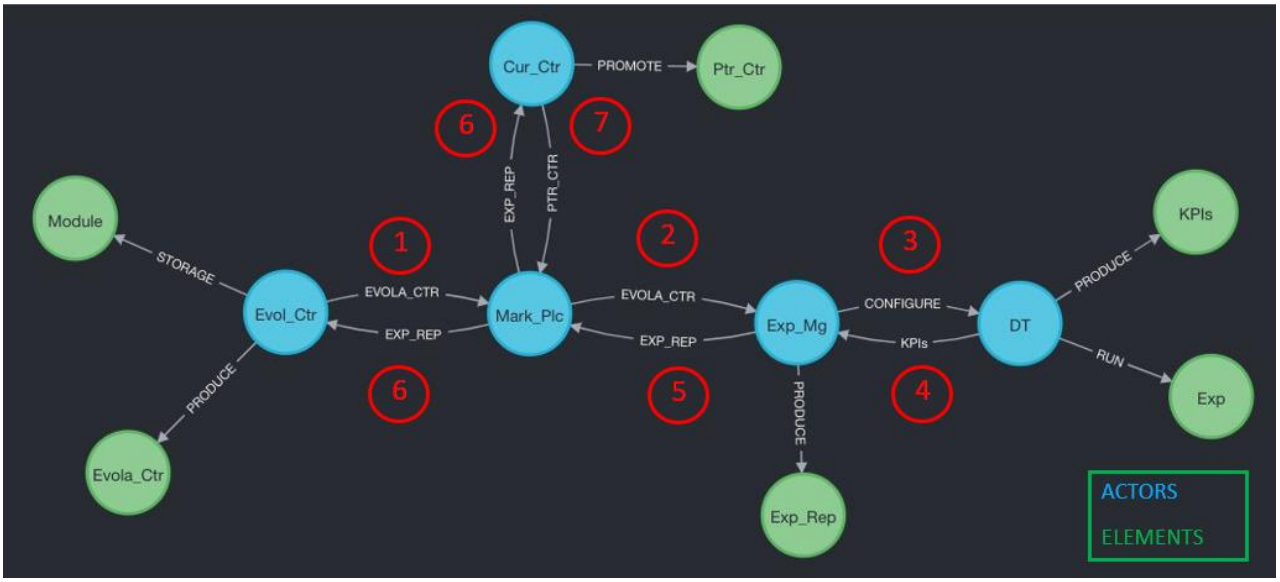

Figure 2: Neo4J graph representation of the high-level architecture of the PoC.

The blue nodes represented in Figure 2 are labelled as actors in the graph, while the green ones are denoted as elements. Actors are controllers that can evolve, although the evolution of actors is out of the scope of this PoC. Element include supporting artefacts, such as software modules, evolvable controllers, repositories, etc.. The Evolution Controller (Evol_Ctr) stores the list of available modules and uses them to compose an Evolvable Controller (Evola_Ctr) that is sent to the Marketplace (Mark_Plc). Although the Marketplace is represented as a single node, it is a distributed network of several nodes that are part of the rest of the actors. That means that the Marketplace is distributed across all the instantiations of the Evolution Controller, Experimentation Manager, Digital Twin and Curation Controller, ensuring the security and traceability of the evolution process. Section 2.2 and Section 3.1 explain further on this topic.

The Marketplace will share the Evolvable Controller with the Experimentation Manager. This actor is responsible for configuring the Digital Twin. That means setting the correct parameters for reproducing the desired environment and sending a list of experiments to be run. The Digital Twin runs the Experiments (Exp) and collects Key Performance Indicators (KPIs) for the controller under testing, which sends back to the Experimentation Manager, that composes an Experimentation Report (Exp_Rep) that includes all the run experiments with their results. The Experimentation Report is then shared in the Marketplace and sent to the Curation Controller (Cur_Ctr) that decides if the Evolvable controller is promoted to a Protected Controllers (Ptr_Ctr), information that is shared in the Marketplace too. The Experimentation Report is also sent to the Evolution Controller as feedback for the composition of the next Evolvable Controller, repeating the same process.

Some clarifications about the evolution process presented here are:

- Although the PoC only considers one instance for each actor (each actors runs only in one computer system except for the Marketplace), multiple evolution processes can run simultaneously with the Marketplace keeping track of every one of them.
- For simplicity, in this PoC the Marketplace directly forwards the messages that it receives to the other nodes. For example, the Experimentation Report is sent to the Curation Controller automatically after receiving it from the Experimentation Manager. Another strategy more aligned with the concept of a Marketplace would be that the other actors initiate the communication, periodically asking if there are any updates on the information they are interested in.

- The deployment of a controller after the evolution process is addressed independently as explained in Section 3.3.

Other assumptions and considerations related to the implementation details are explained in Section 3.

2.2. SECURE AND TRACEABLE EVOLUTION

In order to warrant traceability and security in the evolution process, every node in the system has to be able to trust and verify unequivocally the authenticity of each generated artifact. In this sense, and to achieve this level of trust between disparate nodes, a proper decentralization of the evolution process needs to be ensured. The Marketplace solves this with the implementation of two technologies: Ethereum and the InterPlanetary File System (IPFS). Trust, authentication and traceability are ensured thanks to Ethereum, a Distributed Ledger Technology (DLT) with Smart Contract (SC) functionality, while integrity, scalability and fault tolerance are achieved with IPFS, a distributed and decentralized filesystem that implements content addressing.

This sub-section attempts to justify the main technologies used in the implementation of the Marketplace and how these are implemented to satisfy the aforementioned requirements.

2.2.1. TRUST AND AUDITABILITY

In an AN, every node is capable of making decisions on its own, without depending on a central authority for validation. This means that ensuring autonomy in such a set of independent nodes implies ensuring trust, for which one crucial requirement is decentralization. This implies that, in order for evolution to take place in a fully decentralized system, the evolution process needs to be auditable and nodes have to be authenticated.

Figure 3: Ethereum logo.

Being a blockchain, Ethereum has the capability of recording and tracing transactions, which provides traceability and trust throughout the process. Thanks to the use of SCs, it also has the ability to implement decentralized applications (DApps) that enable for more complex processes to take place in the blockchain.

Given the sensitive nature of the evolution and experimentation process in an AN, security and privacy need to be ensured as well, for which a private and permissioned blockchain can be configured through Ethereum. A permissioned blockchain is a blockchain in which every node has to be given permission to join and operate. It restricts access to the network to certain nodes and may also restrict the rights of those nodes on that network. A private blockchain is a permissioned blockchain that is controlled by a central authority or organization, who determines who can be a node and what rights and functions a specific node may have. This ensures that only trusted actors join the network, warranting security and privacy, which is why we considered it for our use case.

2.2.2. DISTRIBUTED MEANS OF STORAGE

Even though the Ethereum blockchain provides a means to ensure decentralization and auditability throughout the evolution process, the nature of DLTs makes it unsuitable for storing large amounts

of data or executing complex applications. A distributed means of storage is therefore required to safeguard a decentralized way of storing artifacts, without sacrificing scalability or fault tolerance.

The InterPlanetary File System (IPFS) is a peer-to-peer hypermedia protocol for storing and sharing data in a distributed file system. Put in simple terms, IPFS allows for sharing and storing data of any kind (files, websites, applications...) in a distributed and decentralized fashion, without depending on a single endpoint or source's availability to access them.

Whereas other hypermedia sharing protocols, such as HTTP, may reference a specific resource by pointing to a specific location in a specific server (e.g., http://my-website/index.html), IPFS effectively achieves decentralization though **content addressing**. Every file in IPFS is linked to a Content Identifier (CID), represented by a **cryptographic hash** that uniquely identifies the content of the file, and makes it possible for the file to be accessed from any node in the network without having to worry about integrity or authenticity of the data.

Content addressing ensures that a new CID will be generated whenever a file is modified in IPFS. The integrity of artifacts can therefore be guaranteed through the use of unique CIDs, which in turn can be registered in the blockchain to provide auditability to file storage.

Figure 4: IPFS logo

2.3 RELATED USE CASES

The list of use cases described in FGAN-O-013 [2] for which this PoC is relevant and why is the following:

- **FG-AN-usecase-001, Import and export of knowledge for autonomous network**
  The Marketplace is the perfect example of managing the exchange of information between peer components. As previously mentioned in Section 2.2, integrity and security of data are achieved through the use of IPFS, while the Ethereum blockchain enables traceability and consistency throughout the evolution process thanks to the use of smart contracts.

- **FG-AN-usecase-002, Configuring and driving simulators from autonomous components in the network**
  The evolution process based on the experiments that are configured and run by the Experiment Manager in the Digital Twin is an example of this use case.

- **FG-AN-usecase-003, Peer-in-loop (including humans)**

By combining both IPFS and blockchain technologies, this PoC effectively implements a peer-to-peer network of autonomous nodes that exchange data that can be understood and modified by humans as well as analysed by different autonomous nodes to take actions (like updating the knowledge base). For instance, the Experimentation Manager *(*Exp_Mg*)* may upload an experimentation report to the Marketplace, which in turn will be shared with the Curation Controller (Cur_Ctr), who may take the decision to promote a specific controller to a Protected Controller.

- **FG-AN-usecase-026, Ev-as-a-service: Achieving zero touch evolution in a delegated autonomy case**

  The PoC is an end-to-end structure for autonomous evolution of controllers given some specific requirements. As such, this work can support generation of guidelines in future iterations of the FGAN architecture.

- **FG-AN-usecase-0027, Experimentation as a service: Digital twins as platforms for experimentation**

  Although the implementation of a DT is out of the scope of this project, the architecture presented here is an example of how to integrate a DT in a close-loop network for the evolution and testing of controllers.

- **FG-AN-usecase-0031, OpenCN: An open repository of intents for controllers and modules**

  The Marketplace developed for the PoC offers an open repository that not only stores the modules and controllers available, but also ensures security by keeping track of the transactions between all the nodes and has the potential to trace the evolution process when using an evolution algorithm.

- **FG-AN-usecase-0039, Towards openness in AN & FG-AN-usecase-0040, Awareness in AN**

  The Marketplace defined in this report is an open platform that allows for the creation, storage and access to data that is part of a common knowledge base of information. Such a knowledge base is heterogeneous – as it accepts different formats of data – and open –because that heterogeneous data can come from disparate actors.

  Being open, and thanks to the traceability and security provided by blockchain and IPFS, allows for the knowledge base provided by the Marketplace to be used and understood by different systems and is the building block that makes awareness possible within them.

3. PoC IMPLEMENTATION

This section focuses on the code implementation of the PoC, explaining the tools and strategies followed for the development and final deployment of the evolution close-loop. Section 3.1 explains the approach followed for the implementation and representation of both modules and controllers, as well as the evolution process. Section 3.2 describes the Marketplace development and the integration, final packaging and deployment of the PoC. Finally, Section 3.3 explains how after the evolution any controller can be fetched from the Marketplace to be deployed as a REST service.

3.1.MODULES, CONTROLLERS AND EXPERIMENTATION

Figure 2 shows the high-level architecture of the PoC, where modules and controllers are defined as green nodes or elements. For the controllers, the term Evolvable Controller (Evola_Ctr) remarks that it is the entity that goes through the evolution process. In this PoC, both modules and controllers

are implemented as Python classes. The source code for both definitions is available in [3], in the *Mod_Ctr.py* file, which is reproduced in all the nodes. The fact that all nodes work with the same definition is part of a list of important assumptions that are now explained.

3.1.1. IMPORTANT ASSUMPTIONS

- The list of available modules, as well as the definition of the classes *Module* and *Controller*, are contained in the *Mod_Ctr.py* file, having each node (blue circles in Figure 2) an identical copy of it. In the case of introducing new modules, something that is not considered in this PoC, a protocol for distributing the information and ensuring consistency is required. However, the Marketplace provides the perfect tool for implementing this feature.
- Since the PoC is focused on laying the groundwork for the evolution architecture, the modules and controller considered are rather simple, reproducing the behaviour of simple mathematical operations. However, we assume that this approach is sufficient for proving our architecture and that it can be improved in the future.
- As for the evolution, the algorithm uses random combination of modules, avoiding repetition of the combinations that have already being tried for which their result is testing is known. We assume that our use of random module combinations is structurally equivalent in this context to those controllers produced by evolution. Of course, they are not expected to be equivalent in their operation.

All these assumptions simplified the PoC implementation, but do not limit its effectiveness when laying the groundwork for the evolution of controllers in a distributed system. New features can be added with a few or no changes in the base work presented here. Section 5 comments further on this topic.

3.1.2. MODULE DEFINITION

The class *Module* have four attributes,

- The module id
- The function that it implements
- The parameter for that function
- The module representation

The list of possible functions is limited to the mathematical operations add, subtract, pow, and multiple. These functions are also defined as Python functions in the file *Mod_Ctr.py*.

Figure 5 shows the Python code for the multiple function, *f_mul*. Figure 6 shows an instantiation of a *Module* object with *f_mul* as function with the float 3 as a parameter. The id is a string explanation of the function, while the representation is the symbol that denotes the mathematical operator. The *Module* class has the method *execute* represented in Figure 7 that, given an input, returns the output of applying the function with the designed parameter. Once a *Module* object is instantiated is possible to modify its parameter with the method *set_method*.

```
def f_mul(x, param):
    return x * param
```

Figure 5: Python code for the definition of the function *f_mul*.

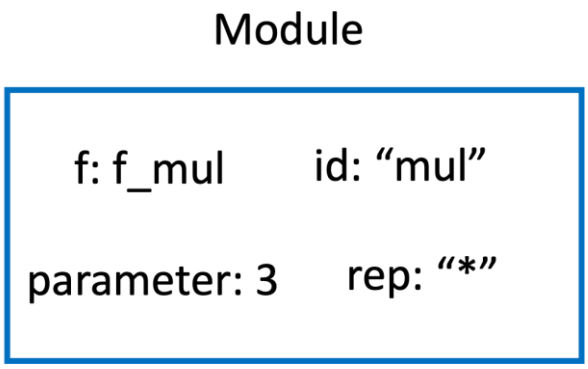

Figure 6: Representation of a *Module* object.

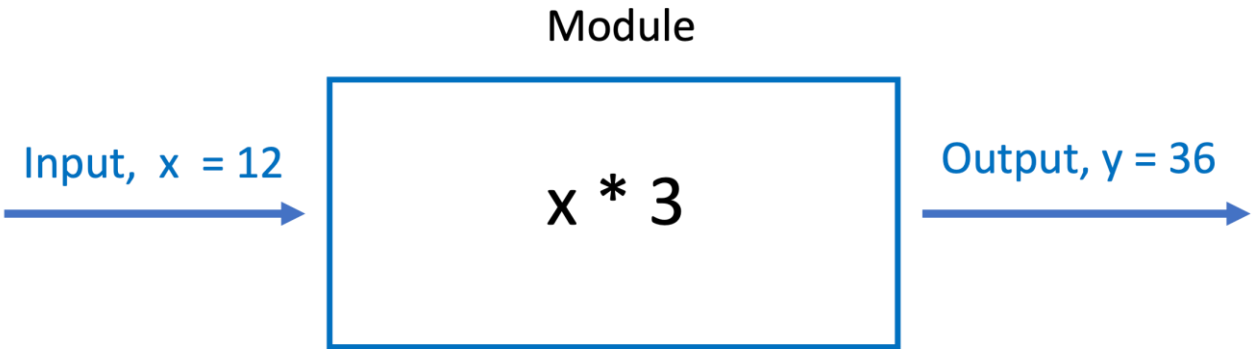

Figure 7: Representation of the method *execute* of a *Module* object.

3.1.3. CONTROLLER DEFINITION

The class *Controller* has three attributes:

- The controller id, which is unique
- A list of the modules that compose the controller
- A list of the parameters for each of its modules

Figure 8 represents the instantiation of a *Controller* object. In that case the two modules implement the functions substract and multiply tuned with the parameters 2 and 10 respectively. After the object is instantiated, it is possible to call the method *execute* as represented in Figure 9. The *executed* method for each *Module* objtect will be called in order, being the output of each phase the input for the next one. Additionally, the *Controller* has a method *get_dict* that returns a Python dictionary that describes its attributes.

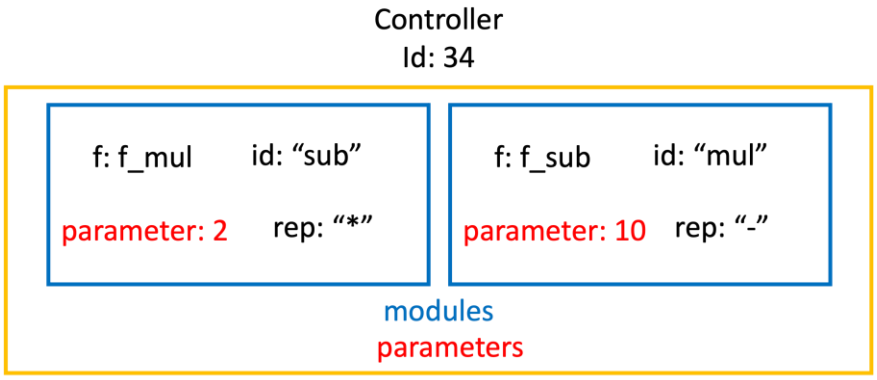

Figure 8: Representation of a *Controller* object.

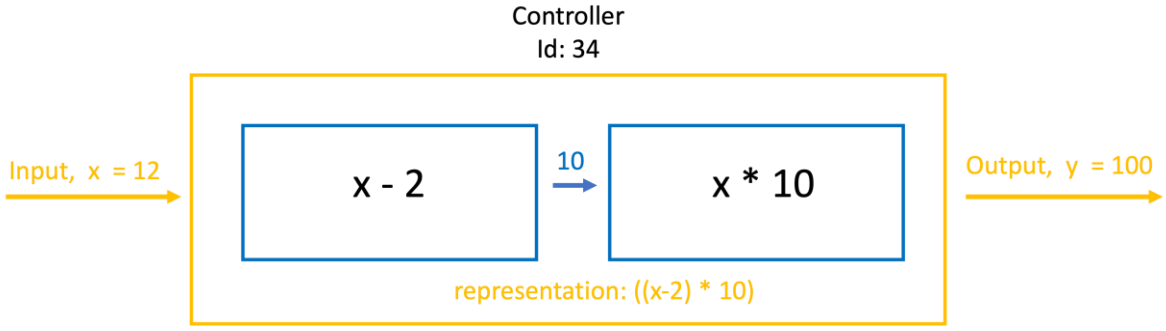

Figure 9: Representation of the method *execute* of the *Controller* class.

A Python dictionary can be easily converted into a JavaScript Object Notation (JSON) file which is the selected format for transmitting the controller information between nodes. The transmitter node sends the JSON description of a controller and since the receiver also has the list of all available modules, it is able to recreate a copy of the controller by calling the function *json_to_ctr (dictionary)*. We have chosen JSON since it is a lightweight data-interchange format that is readable by humans. Figure 10 shows the JSON representation of the *Controller* object introduced in Figure 8.

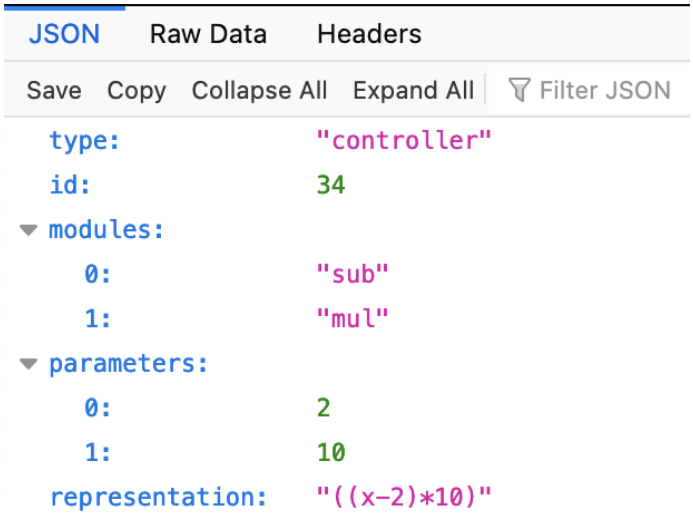

Figure 10: JSON representation of a *Controller* object extracted from the Marketplace.

3.1.4. EXPERIMENTATION AND EVOLUTION

All blue nodes in Figure 2 communicate by exchanging JSON files via the Hypertext Transfer Protocol (HTTP). On the Marketplace side, we have used the language JavaScript for handling the HTTP communication, while in the rest of the blue nodes we have chosen the combination of the Python framework Flask [6] and Gunicorn [7], a Python HTTP server for Web Server Gateway Interface (WSGI) applications.

Following the architecture presented in Figure 2, the Evolution Controller (Evol_Ctr) sends the JSON representation of a new Evolvable Controller (Evola_Ctr) as the one shown in Figure 10. After storing the JSON file, the Marketplace forwards it to the Experimentation Manager (Exp_Mg). This node configures the Digital Twin (DT) by sending both the JSON representation of the controller and a list of experiments to be run. This PoC presents a simple case where only two experiments implemented as Python functions are available in the DT:

- ***Average*:** Return the average of the controller output after 100000 iterations, being the input drawn from a uniform distribution between 1 and 10.
- ***Value*:** Given a fixed input, return the absolute value of the difference between the controller output and a desired value. By default, the input is 5 and the desired output is 35.

The DT insatiate a Controller object using the JSON representation and, after running the experiments, sends back to the Experimentation Manager a JSON containing the results or Key Performance Indicators (KPIs, Figure 2). The Experimentation Manager composes an Experiment Report (Exp_Rep) that is a JSON containing the id of the controller under testing and its results in each experiment. The Experiment Report is sent to the Marketplace and then forwarded to the Curation Controller (Cur_Ctr). An example of an Experiment Report (running the experiments with the default settings) is shown in Figure 11. The Experiment Report is also forward to the Evolution Controller, since the results are useful feedback if an evolutionary algorithm is implemented.

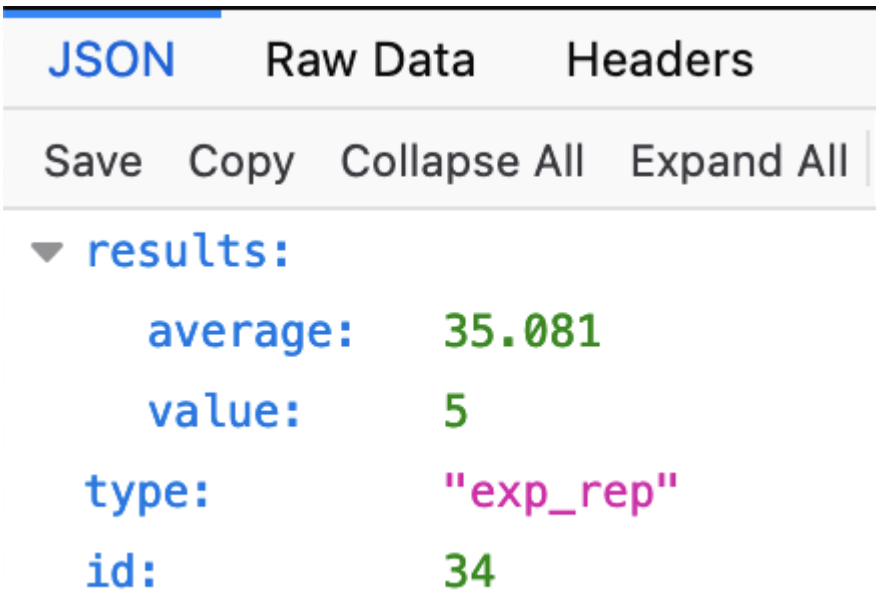

Figure 11: JSON representation of the Experiment Report of the controller described in Figure 10, extracted from the Marketplace.

Based on the Experiment Report, the Curation Controller (Cur_Ctr) will decide if the Evolvable Controller referenced by its id is promoted to Protected Controller (Ptr_Ctr) for one or more

experiments. Treating the protected category independently for each experiment is based on the idea that a controller suitable for a specific use case may perform poorly if the case changes. For example, Figure 12 shows the notification of the Curation Controller to the Marketplace informing that the controller with id 34 (Figure 10) is a Protected Controller for the experiment/use case *value*. Note that the same controller has not the category of protected for the experiment *average*. This decision depends on the criteria followed by the Curation Controller, that for this example is:

- ***Average*:** Protected Controller if the result is greater than 500.
- ***Value:*** Protected Controller if the result is smaller than 20.

Therefore, these values serve as configurable thresholds to decide if an evolvable controller may be promoted to a protected controller.

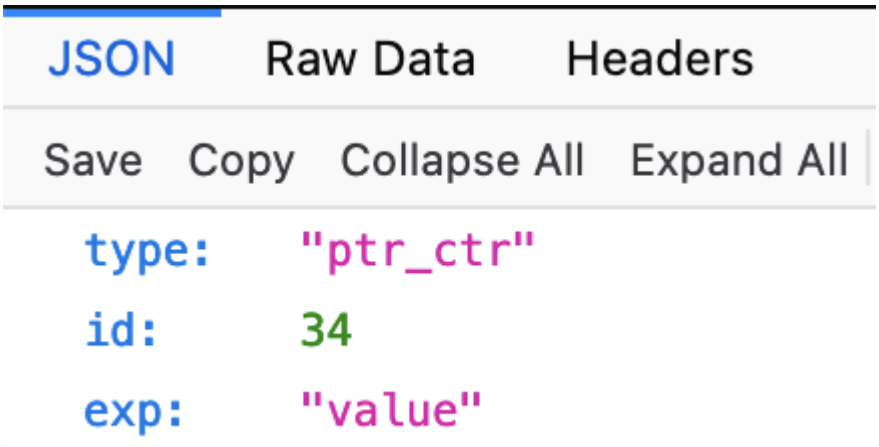

Figure 12: JSON file registration of Protected Controller, extracted from the Marketplace.

3.2. FUNCTIONAL IMPLEMENTATION

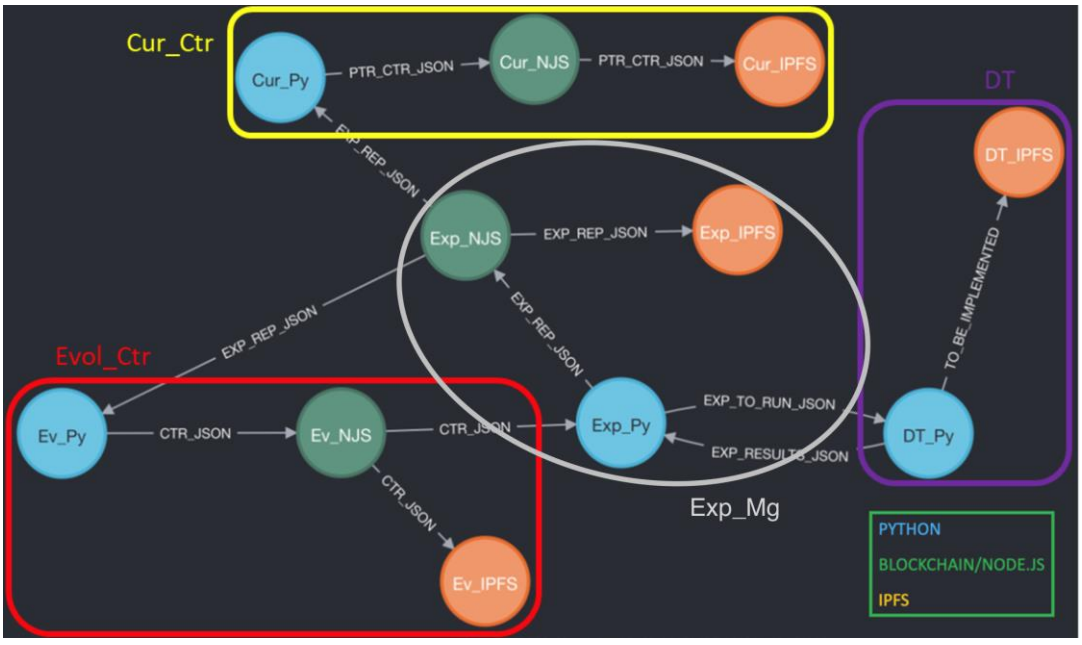

Figure 13: Functional level architecture of the PoC.

Figure 13 shows the functional level architecture of the PoC, where each node represents a specific technology implemented. As can be seen from the picture, each one of the actors shown in Figure 2

is composed by a subset of services that represent a specific function within the PoC. Different technologies are used depending on the type of functionality implemented by each service. A description of each type of service and the technology it uses is provided below:

- **Experimentation and Evolution**: responsible for the evolution and experimentation logic of each actor. As described in previous sections of this report, different actors have different functionalities, so implementation varies from actor to actor (the Cur_Ctr promotes controllers based on experimentation reports, the Evol_Ctr creates new controllers and sends them to the Exp_Mg etc.).
    - Programming language/Frameworks: Python.
    - Naming convention: *actor name abbreviation* + *_Py* (e.g., *Cur_Py* for Curation Controller)
- **Marketplace:** services that make up the functionalities of the Marketplace.
    - Ethereum service: responsible for receiving and executing transactions through smart contracts in Ethereum. These transactions include uploading a file to IPFS, retrieving an uploaded file by its CID, or by its id.
        - Programming languages/Frameworks: Node.js, Hardhat, Solidity.
        - Naming convention: *actor name abbreviation* + *_NJS* (e.g., *Cur_NJS* for Curation Controller)
    - IPFS service: node connecting each actor to a private IPFS distributed filesystem network.
        - Naming convention: *actor name abbreviation* + *_IPFS* (e.g., *Cur_IPFS* for Curation Controller)

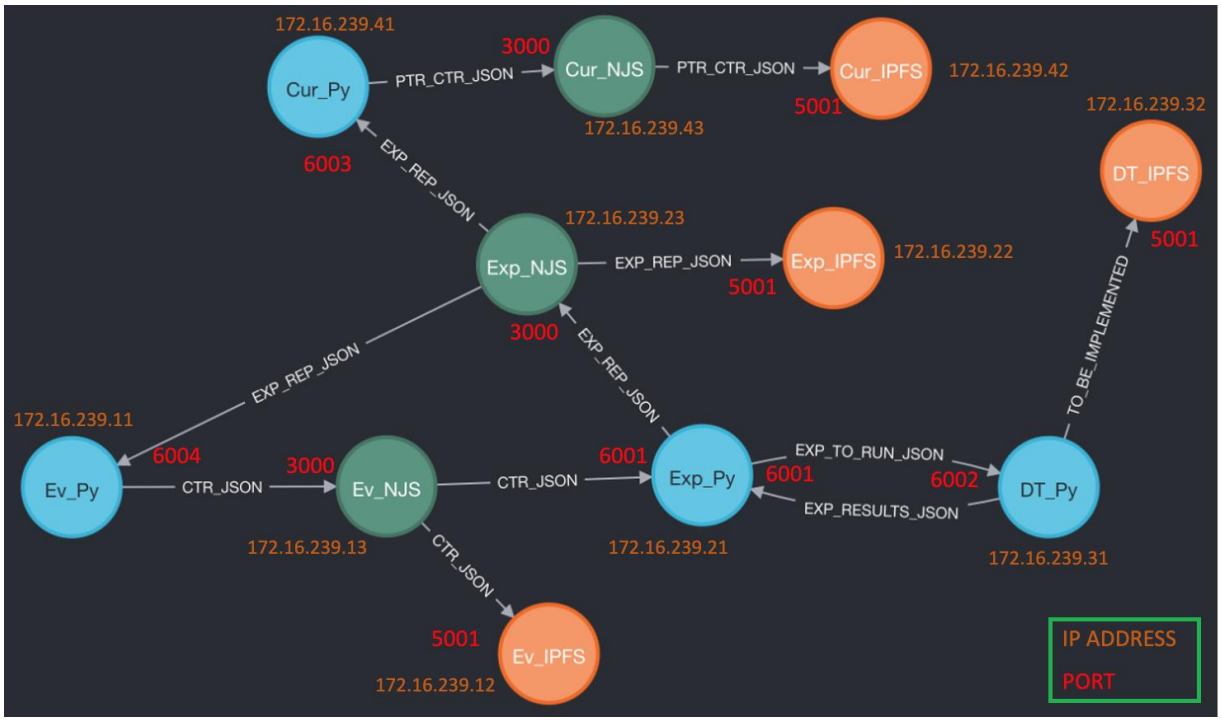

Figure 14: Network architecture of the PoC.

All independent services have been implemented through Docker containers, orchestrated and configured via a Docker Compose *(docker-compose.yml)* file. Each container is interconnected creating a network that is shown in Figure 14. From it, it can be seen that each container/service of the system has an IP address and a specific set of ports exposed for communication with other nodes within the network. The distribution of these ports and addresses is detailed in Table 1.

Node	Container name	IP address	Ports	Function
Evol_Ctr	evol-ctr-py	172.16.239.11	6004	Builds controllers combining modules Sends controllers for experimentation
	evol-ctr-ipfs	172.16.239.12	4001 8090:8080 5001	Libp2p swarm connection HTTP gateway and read-only API IPFS API
	evol-ctr-mkp	172.16.239.13	8545 3000	JSON RPC connection to Ethereum Marketplace HTTP API
Exp_Mg	exp-mg-py	172.16.239.21	6001	Receives controllers Passes controllers and experiment list and parameters to the Digital Twin
	exp-mg-ipfs	172.16.239.22	4001 8091:8080 5001	Libp2p swarm connection HTTP gateway and read-only API IPFS API
	exp-mg-mkp	172.16.239.23	8545 3000	JSON RPC connection to Ethereum Marketplace HTTP API
DT	dt-py	172.16.239.31	6002	Runs experiments
	dt-ipfs	172.16.239.32	4001 8092:8080 5001	Libp2p swarm connection HTTP gateway and read-only API IPFS API
Cur_Ctr	cur-ctr-py	172.16.239.41	6003	Decides if a controller is protected
	cur-ctr-ipfs	172.16.239.42	4001 8093:8080 5001	Libp2p swarm connection HTTP gateway and read-only API IPFS API
	cur-ctr-mkp	172.16.239.43	8545 3000	JSON RPC connection to Ethereum Marketplace HTTP API

Table 1: Containers and ports

3.2.1. THE MARKETPLACE

As described in Section 2 of this report, rather than a single node of the network that makes up the PoC, the designed Marketplace is in itself a distributed network of several interconnected nodes. As shown in the previous sub-section, these nodes are composed of Docker containers incorporating two different services (Ethereum and IPFS) that make up the Marketplace and are included into every actor that is part of the PoC and shown in Figure 2.

This means that the Marketplace is distributed across all the instantiations of the Evolution Controller, Experimentation Manager, Digital Twin and Curation Controller, connecting every actor to the traceable ledger of evolution artifacts created throughout the process and stored in IPFS and accessible through the private Ethereum blockchain.

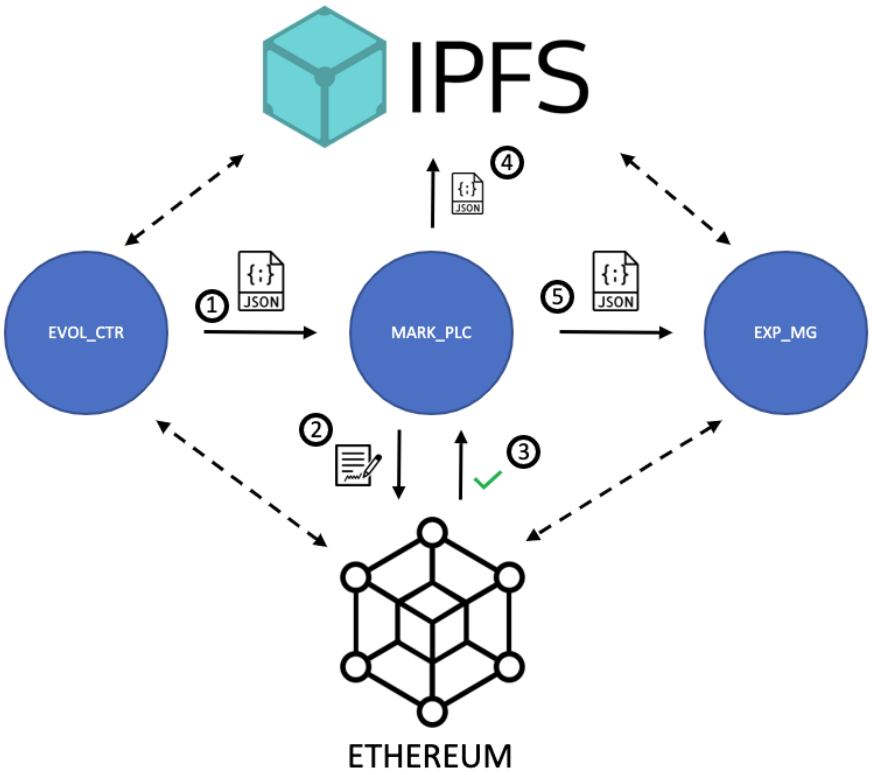

Figure 15: Example of uploading process in the Marketplace.

Figure 15 presents an example of how the Marketplace works when uploading an artifact. The process is described below:

1. The Evolution Controller sends an Evolvable Controller in the form of a JSON file to the Marketplace. This JSON file has the structure described in Section 3.1, containing information that identifies the controller according to its definition, and is sent via a POST HTTP request to port 3000 of the *evol-ctr-mkp* container, which handles the HTTP API of the Marketplace (see Table 1).
2. The Marketplace receive the JSON artifact via port 3000 and registers the file into Ethereum via the *Marketplace.sol* SC (see sub-section 3.2.2 for further details). The operation is run by the *marketplace.mjs* script, that is contained in every *_NJS* container and handles the operation of the Marketplace, interconnecting Ethereum with IPFS, and handling transactions on behalf of each node.
3. Ethereum validates the transaction, confirming in turn the registration of the JSON artifact into the chain.
4. Once the transaction is confirmed, the *marketplace.mjs* script uploads the JSON file to IPFS via the IPFS API (port 5001).
5. After uploading to IPFS, the JSON artifact is sent to the next corresponding actor/actors (in this case, the Experimentation Manager).

3.2.2. THE MARKETPLACE SMART CONTRACT

Just as has been stated in earlier sections of this report, the Marketplace SC is responsible of registering artifacts generated throughout the evolution and experimentation process into the blockchain, ensuring traceability, security, accessibility and scalability. While the private IPFS network stores artifacts and provides scalability while ensuring availability, the Ethereum blockchain keeps track of all transactions throughout the process and constitutes a historic record of file versions, upload times and other metadata.

In order to achieve this successfully, file registration is managed by *Marketplace.sol*, a SC that has been laid out to capture meaningful information in order to optimally ease identification of files and browsing the register. To this end, a structure named *File* has been created in the *Marketplace.sol* SC, whose elements are listed below:

- File number *(uint256)*: number of the file uploaded to the Marketplace chronologically.
- CID *(string)*: the file's Content ID, computed before being uploaded to IPFS.
- Name *(string)*: name of the file.
- Object Type (*string*): type of the object uploaded: controller, experimentation report etc.
- Uploader *(address payable)*: Ethereum address of the node that is registering the file.

A mapping is then assigned to the File structure and the total number of files uploaded to the Marketplace, which is used to keep track of files in the system. This number is in turn saved in the chain as the File number attribute.

It is worth noting that, while the *id* of a controller as described in section 3.1 represents a specific instance of a controller, the CID is a unique cryptographic hash that represents a file, and does not change as long as the contents of the JSON stay the same, i.e., when the evolution is run multiple times.

3.3.CONTROLLER DEPLOYMENT USING TOSCA

This PoC also includes the automatic deployment of a controller as a REST service. The code for this part can be found in the folder *ctr_deployment* [3]. For this purpose, a YAML file is written according to the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [8], as it is recommended in the Build-a-Thon proposal [1]. The TOSCA file is then parsed and executed by using the orchestrator xOpera [9]. Given the CID of a controller as input, the following task are performed by the orchestrator (Figure 16):

- Fetch the JSON representation of the selected controller from the Marketplace.
- Create an instance of the class Controller based on the JSON representation.
- Deploy a REST service in a Docker container.

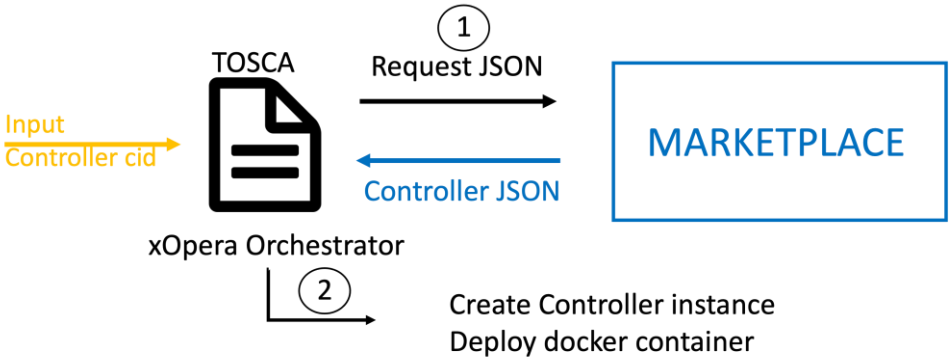

Figure 16: Representation of controler deployment.

For the development of the REST service, we have used the Python tools Flask and Gunicorn (Section 3.1.4). Once the service is deployed it will reply to HTTP GET requests (Figure 17). The HTTP response is a JSON that contains the information about the controller and the output based on the request's input (Figure 18).

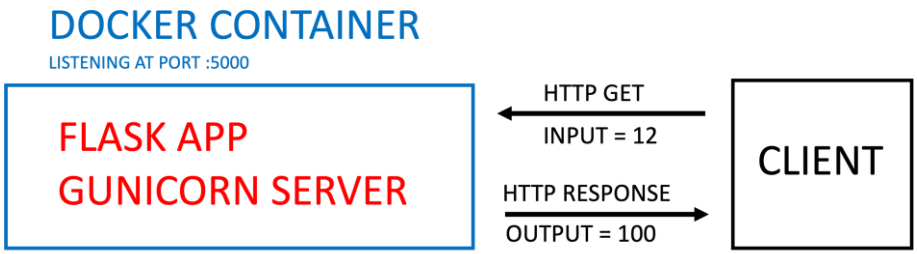

Figure 17: Controller REST service representation.

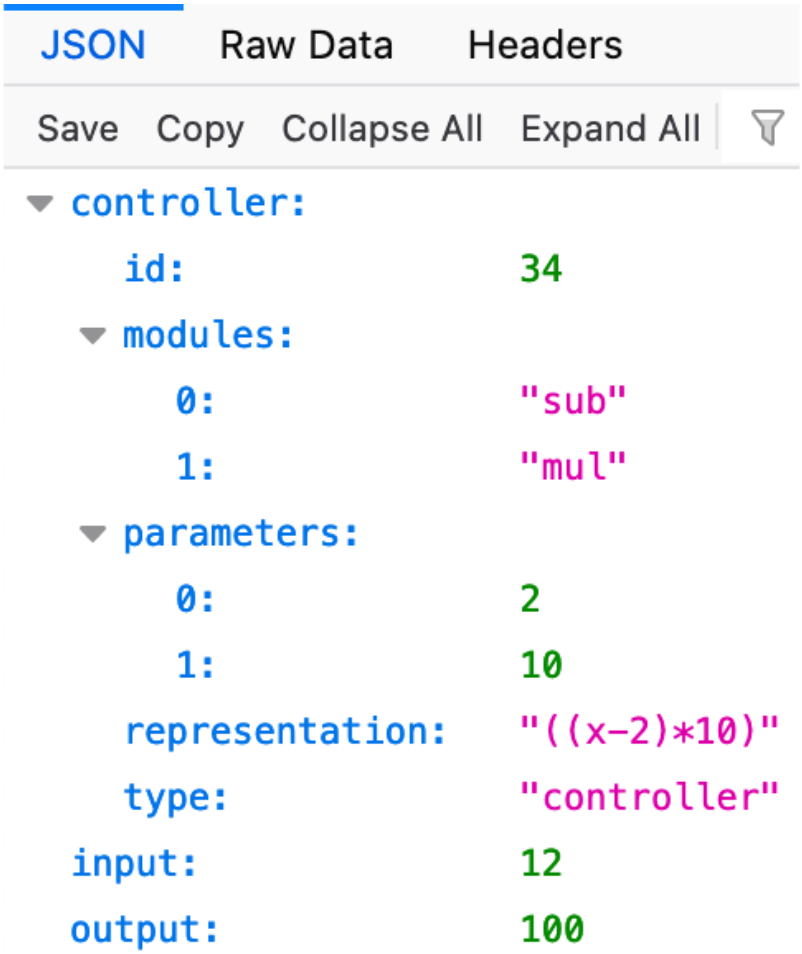

Figure 18: JSON response by the controller REST service.

The controller implemented is rather simple, and the REST service is not actually a close-loop (we already implemented a close-loop for this contribution in the evolution process). However, the purpose of this part is to prove that is feasible to deploy a controller with just its CID, getting its representation from the Marketplace. The CID of the controller with id 34 is the default input in the file *inputs.yaml* inside the *ctr_deployment* folder [3], although it can be replaced by any other controller's CID. As Section 3.2 explains, this CID will remain the same as long as the JSON representation of the controller is the same, no matter how many times the evolution is run. This feature ensures a "check-point" to which the service can always return.

4. PoC RESULTS

This section presents the results of the PoC which code is included in [3]. Along with this report, the BT submission includes a demo video of the PoC execution. The configuration parameters for the presented PoC are:

- Each controller is composed of 2 modules.
- The parameter of each module can only adopt two possible values: 2 and 10
- Evolution is done by randomly combining pairs of modules, avoiding repetition of the same modules in the same order with the same parameters (that would yield an identical controller).
- The PoC stops after trying out all possible combinations
- As for the Curation Controller criteria, the default parameters are used. As explained in Section 3.1.4 that means:
    - ***Average*:** Protected Controller if the result is greater than 500.
    - ***Value*:** Protected Controller if the result is smaller than 20.
- All docker containers run on a local machine and in a private network as explained in Section 3.2.

The purpose of this configuration is to illustrate the mechanics of the proposed architecture by a clear and quick demo that can be reproduced easily in a local machine by anyone who clones the repository [3]. The close-loop chronological order was explained in Section 2.1 and it is shown in Figure 2. Figure 13 shows the representation of the actual docker container nodes. As shown in the demo video, one must look at log messages of the docker containers to analyse the process.

Figure 19 shows the log messages of the Evolution Controller (its Python part), denoted as Ev_Py in Figure 13. The output shows the JSON representation of each controller that the Ev_Py node sends to Ev_NJS. The JSON is forwarded to Ev_IPFS and its response, the CID assigned to each controller, is sent back to Ev_Py by the Ev_NJS. With the printed CID it is possible to access the controller JSON representation in the marketplace, as shown in Figure 20 for the controller with id 34. It is interesting to change the port in the address, to access a different node (Figure 21). The port 8091 corresponds to the Exp_IPFS node and the 8092 to the DT_IPFS (see Figure 14 and Table 1) This proves that the information about each controller is reproduced in all the IPFS nodes.

Figure 19: Log messages of the Ev_Py docker container.

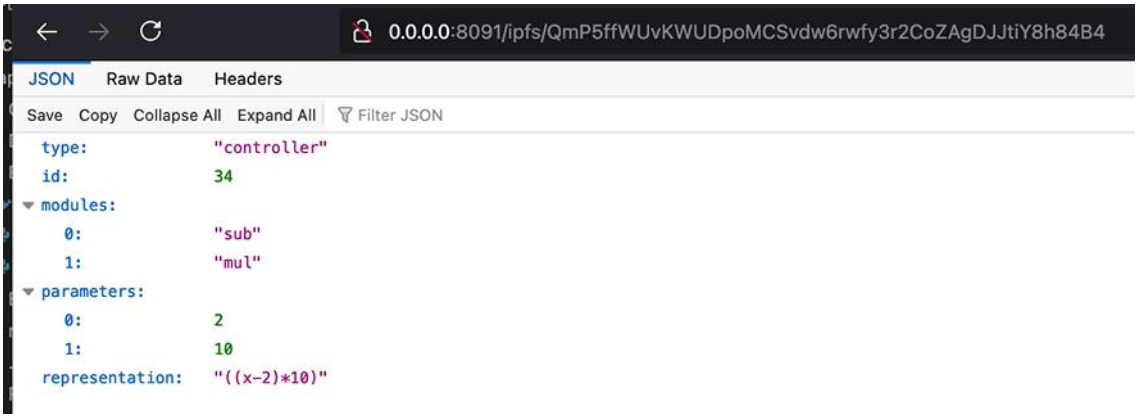

Figure 20: Access to the JSON representation of a controller in the Marketplace via its CID. The Marketplace node that is being consulted is Exp_IPFS.

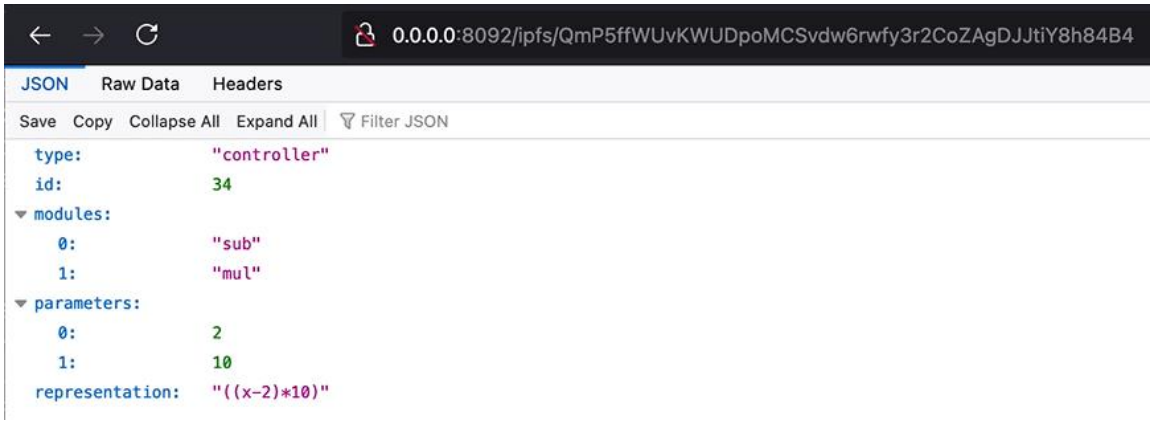

Figure 21: Access to a controller JSON representation in the Marketplace via its CID. The Marketplace node that is being consulted is DT_IPFS.

Moving forward, the next logical step is to check the Exp_Mg output (Figure 22) which includes the information sent to configure DT_Py, as well as its response. Experiment results are also uploaded to the Marketplace. Figure 23 shows the Exp_NJS output that contains the CID for each experiment and information related to the Ethereum smart contract. The output of the *marketplace.mjs* script indicates that a new artifact was received, registered to the chain and uploaded successfully to IPFS through the *Marketplace.sol* SC. This is indicated by printing the result of uploading the file to IPFS (the path and CID generated for the file), as well as the HTTP POST request sent to ports 6004 and 6003 of the Evol_Ctr and Cur_Ctr, respectively. The JSON representation for each experiment has a CID that allows to access it in the Marketplace (Figure 24).

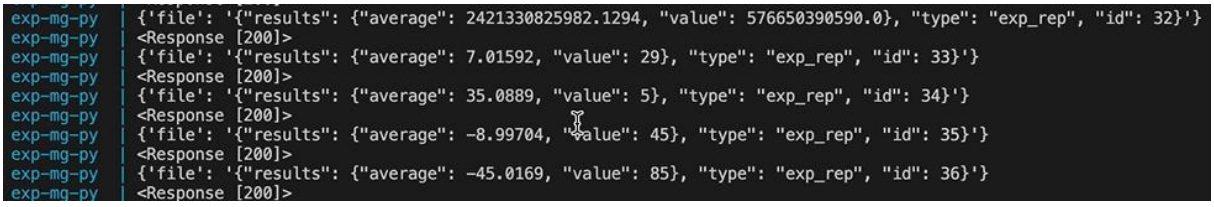

Figure 22: Log messages of the Exp_Mg docker container.

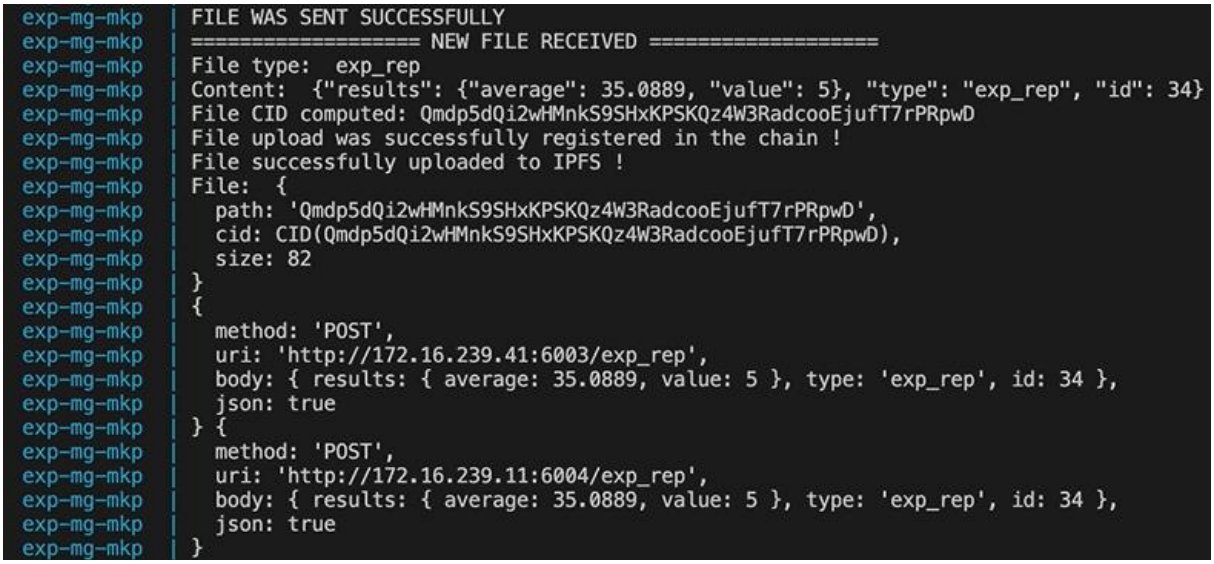

Figure 23: Log messages of the Exp_NJS docker container.

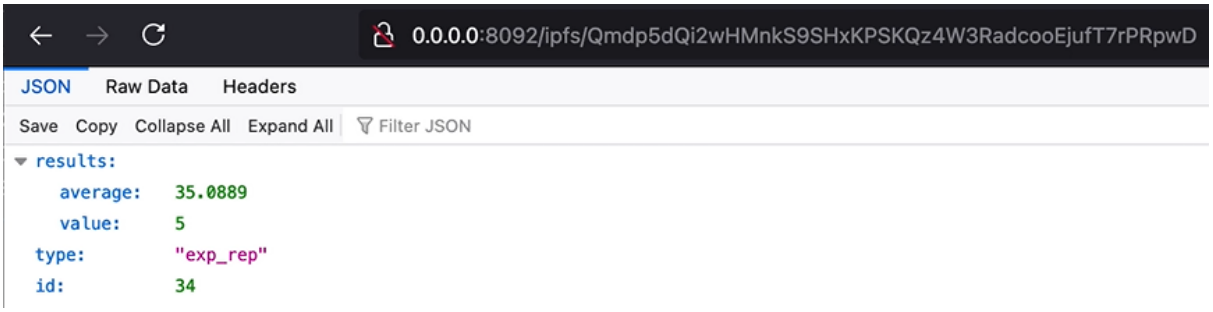

Figure 24: Access to the JSON representation of an experiment in the Marketplace via its CID.

In order to check if the controller has been promoted to the protected category, one must look at the the Cur_Py log messages (Figure 25). Focusing once again in the controller with id 34, it is possible to access to decision of the Curation Controller in the marketplace using the corresponding CID, as shown in Figure 26. This controller ends up with the category of protected for the *value* experiment, but not for *average*.

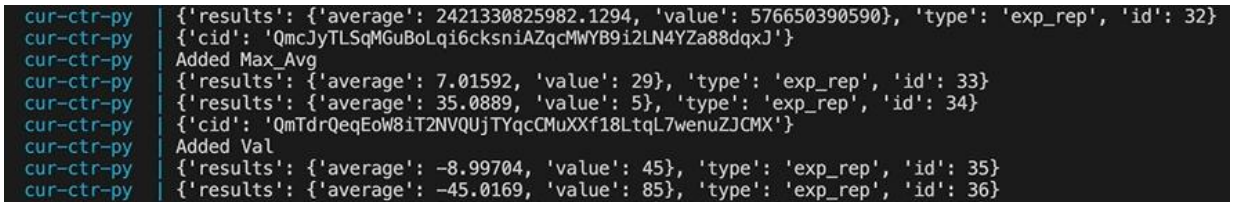

Figure 25: Log messages of the Cur_Py docker container.

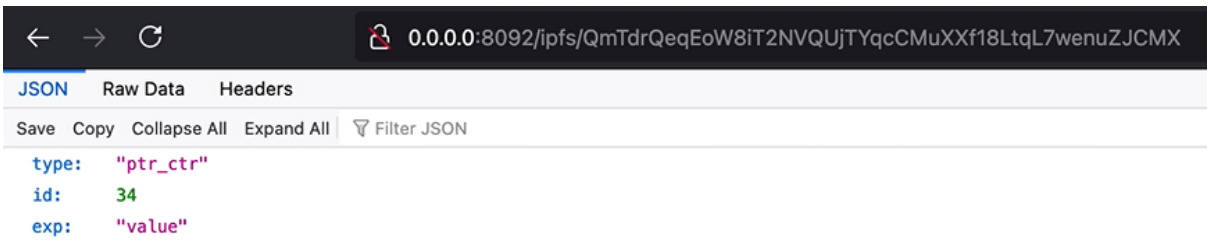

Figure 26: Access to the JSON that registers if a controller has the protected category in the Marketplace via its CID.

As explained in Section 3.3, after the evolution process has occurred it is possible to deploy a REST service based on a controller that, when receives HTT GET requests with a float input, produces an HTTP response with a JSON that contains the information about the controller and the output for the sent value (Figure 18). As for the deployment process, we recommend looking at the video demo. The final REST service is listening at port 5000 and uses the example controller with id 34. Figure 27 and Figure 28 show the response of two GET requests with different float values as input. The JSON produced by the service contains information about the deployed controller as well as the output value for each request.

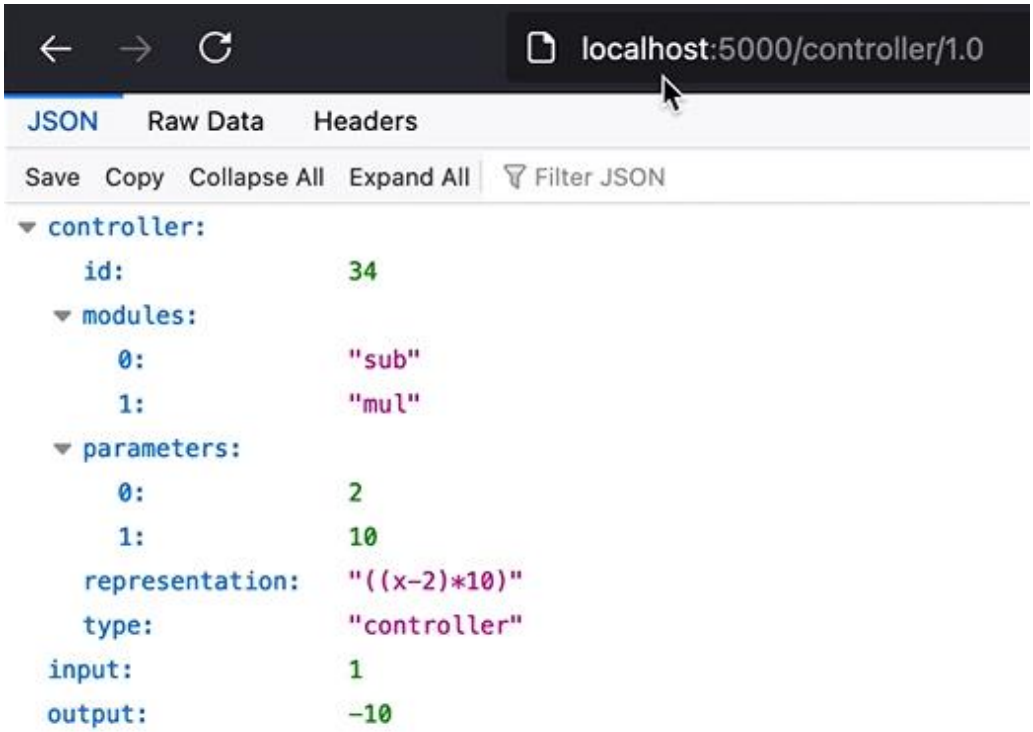

Figure 27: REST service response for the value 1.0.

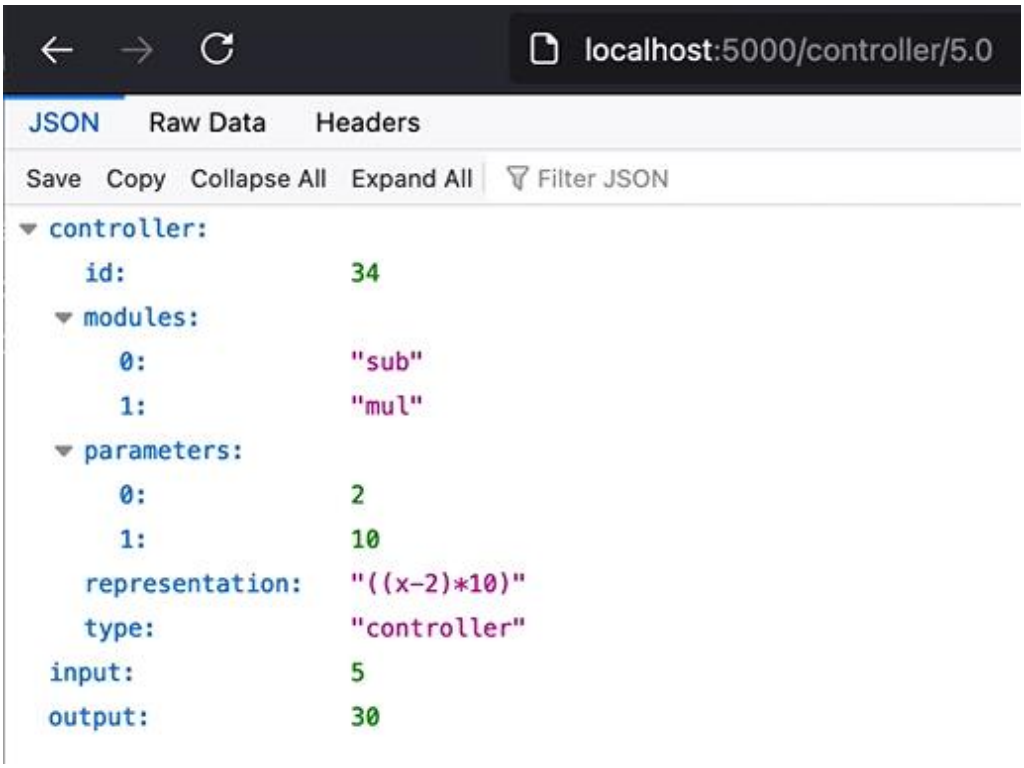

Figure 28: Example REST service response for the value 5.0.

5. CONCLUSIONS AND FUTURE WORK

The PoC described here presents an end-to-end close-loop decentralized architecture for the secure and auditable evolution of controllers. The code of the PoC is available in a GitHub repository [3], and a video demo is included as part of the submission of the "Digital Twins" team for BT.

This team has put its main efforts into laying the skeleton for an evolution framework based on the concepts described in [4], while leaving details regarding each of its components for further development. To this end, simplified implementations of modules and controllers are used for the PoC, as well as random combinations instead of a more complex evolutionary algorithm. Nevertheless, specific improvements on each of these examples can be made relying on the presented work, with little or no change in the architecture. That key feature makes this PoC valuable for the future development of the use cases for AN listed in Section 2.3.

One other key point of focus of our approach is the assurance of security and traceability of the evolution process through the implementation of a decentralized Marketplace. The design and operation of the Marketplace has been described throughout this report, and we believe it represents a strong feature of our approach as well, as, thanks to the implementation of Ethereum and IPFS, it successfully provides with security, auditability, privacy and scalability through decentralization and a distributed means of storage. Furthermore, the choice to use the popular and widely adopted open-source technologies Ethereum and IPFS facilitates its future development and accessibility.

On top of security and auditability, using JSON as the data format for exchanging the controller and experiment information in the Marketplace enhance the explainability of the process, since it is a human readable format. Moreover, the deployment based on docker make the PoC portable and by using open-source tools and providing the source code we have ensured its future use and reproduction.

Regarding future lines of work, some of the ideas are:

- Implement more complex definitions of Modules and Controllers for solving a specific problem in an AN.
- Add a list of experiments that is suitable for that use case.
- Implement more complex evolutionary algorithms than random combination that can make use of the feedback that the Evol_Ctr receives about the experiment results.
- Add the possibility of updating new Modules after the evolution process has started. In order to make this possible all nodes must be notified of the new upload. Fortunately, IPFS provides the tools for making this possible without changing the architecture.
- Implement more complex Smart Contracts that would allow for more complex access control, updating and deleting artifacts, as well as an easier and more intuitive access to files.
- Add a more efficient detection and communication of updates within the system.

6. REFERENCES

[1] "Report from Build-a-thon for ITU AI/ML in 5G Challenge 2022", ITU Focus Group on Autonomous Networks Input 239, FGAN-I-239. [Online], available: https://extranet.itu.int/sites/itu-t/focusgroups/an/input/FGAN-I-239-R5.docx

[2] "Use cases for Autonomous Networks", ITU Focus Group on Autonomous Networks Output 013, FGAN-O-013. [Online], available: https://extranet.itu.int/sites/itu-t/focusgroups/an/input/FGAN-I-013.docx

[3] Submission code github repository, team "Digital Twins", Jaime Fúster de la Fuente, Álvaro Pendás Recondo and Paul Harvey. [Online], available: https://github.com/FGAN-Digital-Twins/docker-network.git

[4] "High level architecture framework for Autonomous Networks", ITU Focus Group on Autonomous Networks Input 198, FGAN-I-198. [Online], available: https://extranet.itu.int/sites/itu-t/focusgroups/an/input/FGAN-I-198.docx

[5] "Building a Digital Twin using Graph Neural Networks", ITU Focus Group on Autonomous Networks Input 058, FGAN-I-058. [Online], available: https://extranet.itu.int/sites/itu-t/focusgroups/an/input/FGAN-I-058.pdf

[6] Python microframework Flask, official website. [Online], available: https://flask.palletsprojects.com/en/2.2.x/

[7] Python WSGI HTTP Server for UNIX Gunicorn, official website. [Online], available: https://gunicorn.org

[8] OASIS TOSCA Simple Profile in YAML v1.3. [Online], available: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html

[9] xOpera orchestrator Github repository. [Online], available: https://github.com/xlab-si/xopera-opera