2023 IEEE Symposium Series on Computational Intelligence (SSCI)
Mexico City, Mexico. December 5-8, 2023

Exploring Heterogeneous Open Multi-Agent Systems on Cloud using a Docker-based Architecture*

1st Gustavo Lameirão de Lima
Graduate Program in Computing (PPGC)
Federal University of Pelotas (UFPel)
Pelotas - Brazil
gustavolameirao@inf.ufpel.edu.br

2nd Marilton Sanchotene de Aguiar
Graduate Program in Computing (PPGC)
Federal University of Pelotas (UFPel)
Pelotas - Brazil
marilton@inf.ufpel.edu.br

***Abstract*—In Open Multi-Agent Systems (OMAS), heterogeneous agents in varying environments or models can transition from one system to another, retaining their attributes and knowledge. This migration process results in an augmented development complexity compared to conventional Multi-Agent Systems. Additionally, the intricacy of this transition arises from uncertainties and dynamic behaviors associated with the agents' changes, necessitating the formulation of techniques to analyze this complexity and comprehend the system's overall behavior. To address these challenges, we employed Docker, which enables a flexible architecture that accommodates different programming languages and frameworks for the agents. This paper introduces a Docker-based architecture that aids OMAS development, facilitating agent migration between various models operating in heterogeneous hardware and software setups. To validate the proposed approach, we conducted simulations using NetLogo's Open Sugarscape 2 Constant Growback and JaCaMo's Gold Miners. These simulations were executed locally, in the cloud, and in a hybrid mode to assess the feasibility of the proposed architecture.**

***Index Terms*—Multi-Agent Systems, Open Multi-Agent Systems, Agent-Based Simulation, Docker**

I. Introduction

Various domains leverage Artificial Intelligence concepts to tackle problems, often relying on Shallow Learning techniques like Gradient Boosting and Random Forest, Deep Learning approaches, or Heuristics such as Genetic Algorithms. These solutions excel when the problem is well-defined and the dataset is large and centralized [1]. However, dynamic problems involving multiple interacting entities require solutions with decentralized control that can adapt in real-time [1].

In this context, Multi-Agent Systems (MAS) offer a promising alternative to Artificial Intelligence. MAS comprises multiple agents, physical or virtual entities with autonomous behavior, capable of acting independently [2], [3]. These agents can perceive their environment and utilize actuators to influence it. In Multi-Agent Systems, agents exchange information for various purposes, aiming to collectively acquire knowledge, update beliefs, and optimize strategies.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001

A particular MAS category, Open Multi-Agent Systems (OMAS), facilitates interaction among agents participating in different models. OMAS involves analyzing heterogeneous agents, each possessing distinct characteristics and operating within diverse environments or models. These agents can migrate from one system to another, bringing along their attributes and knowledge [4]. The heterogeneity in OMAS arises from variations in architecture, objectives, or policies among the agents and their respective models [5]. However, different problems arise when developing applications in OMAS when compared to Multi-Agent Systems [6]. First, there may be implementation problems where agents and models can be created by different teams, in other programming languages, or various platforms/agent architectures. Often, conflicts of objectives may not guarantee that agents will act cooperatively and coordinately [7], [8]. Moreover, there is still the difficulty generated by the uncertainties and the dynamic behavior that the change of agents entails.

Open Multi-Agent Systems need to deal with problems not present in closed systems. For example, the migration of agents between models can occur at runtime, and the motivation for this migration of an agent from one system to another can be different, usually of the developer's choice, such as execution failures, self-will, or some trigger. In addition, conflicts of interest may occur between the new agents when not designed to work in that set [5]. Although the subject has been known for a long time [9], there is still research in the area, as in [10]–[15].

The concept of openness reduction is related to the model's overload from the perspective of the possibility of agents entering and exiting the system without changing the design of many components. Thus, the greater the degree of openness, the smaller the number of changes to a model to receive or send agents [4]. In this way, a perfect open system would not need transformations (0 steps) to accommodate new components, while on the other side, there are systems that would require a complete redesign when new agents arrive [4]. Therefore, given the context presented, it is necessary to make a research effort to reduce the complexity of Open Multi-Agent

Systems regarding the cited problems.

In this context, Docker has the potential to address these issues effectively. Docker is an open-source tool that automates the development of applications in the form of containers. Containers can run different programming languages and frameworks, holding the code and all the requirements.

This paper presents a Docker-based model to aid in developing Open Multi-Agent Systems and to facilitate the migration of agents between different models that can run in heterogeneous hardware and software scenarios, different development environments, and tools for MAS or other operating systems. Since the architecture is based on Docker, it can be run on cloud services, allowing the architecture to build simulations using the host machine resources, on the cloud, or even both (hybrid). In general, the approach aims to advance the generalization of OMAS, simplifying the opening process.

By implementing specific agent registration and routing modules to define system steps as the criteria for determining where agents will go, the original models only need to describe how they will send and receive agents, decreasing transaction complexity within the model. In addition, agents that move between models will transfer knowledge, carrying their attributes with them.

We organized this paper as follows. First, we present related work in Section II. Then, Section III presents details of the proposed approach, flow, implementation details, and the schedule of activities. Then, Section IV explains the platforms and models used to verify the feasibility of the implementation. Next, Section V shows the results obtained from the simulations using local, remote, and hybrid scenarios. Finally, Section VI presents the final considerations of the study.

II. Related Work

The first perspective from most of the related work considers Open Multi-Agent Systems from an architecture/organization point of view, as in [4], [6], [12], [16]–[21]. These studies differ from the approach of this work, mainly because, among all OMAS problems, we focus on the openness part, allowing the agents to move between models. The effects of the conflicts of interest that could come with the agents moving between models are something model-related, requiring to be resolved by the model using any technique, such as the ones described in this study and others. We aim to provide mechanisms to simplify the agents' movement between models, reducing changes in the model's original code. Furthermore, [1], [5], [22]–[24] present similar aspects with some parts of our approach.

In [22], the authors propose a model that uses both NetLogo and Jason agents in an especially complex model for cognitive agents, a Disaster-Rescue simulation. The approach to bring those two different architectures of agents, connecting Jason to NetLogo, is to either include part of NetLogo's internal classes inside the Jason code or do the opposite, including Jason's internal classes inside the NetLogo code since both applications use Java code. Even though this study considers a closed multi-agent model (not an open one), it is similar to our research in that they communicate between agents with different architectures. The main difference is that, in this case, both applications use Java code. We point out a limitation of this work if some agent programmer wants to use this approach to communicate, for instance, NetLogo, to another agent platform that uses another language. Our approach creates a platform that runs isolated code (inside containers) that allows completely different programming environments, allowing the developer to run any code (Java, Python) through our API (Application Programming Interface) service.

The authors in [5] propose a methodology for specifying open multi-agent systems. The structure uses two main ideas: the independent modeling of each of the dimensions of the system (agent, environment, and organization); and the specification of edge concepts, providing information at design time that helps include at runtime the elements of the dimension open. Furthermore, the authors designed elements targeting code for the JaCaMo framework in the implementation. In addition, the study has two case studies, which allow viewing the results throughout the development phases. The main difference between this study and our approach is that the programmer must adapt the model and architecture to the solution. In contrast, our architecture can be adapted to the current running implementation that the programmer has, just using the I/O mechanisms that the agents must have to communicate with our solution. Compared to the parameters analyzed in the study, our platform works on the agent dimension and the implementation phase (the study deals with agent/environment/organization dimensions and analysis/project/implementation phases). We deal with the agent dimension because our primary focus is to transport agents between models, even though extending our platform to transport parts of the environment is possible, like artifacts. On the other hand, we focus the approach on the implementation phase because we want the programmer only to rebuild part of the model to the platform. In this way, the programmer needs to include the mechanisms to communicate with our platform. That way, the primary usage of our platform is in a model already built (implementation phase).

In [1], the authors present the development of a Java-based framework for the development of adaptive open multi-agent systems. The framework developed by the authors, named AMAK, uses three fundamental classes based on object-oriented principles: AMAS (Adaptive Multi-Agent System), Agent, and Environment. Whoever uses the framework must implement these abstract classes, adapting to the model. Furthermore, the authors use developing a socio-technical environmental system as a case study to bring environmental well-being. The architecture proposed in this study is mainly diverse from ours because it is almost a new programming language since the authors compare their results with other agent programming languages (such as Jade, NetLogo, and GAMA). However, our approach does not require the programmer to remake its model into something new. Instead, the programmer must insert the code to allow the agents to communicate with the architecture, which leads to less effort

to allow the agents to move between the models.

In [24], the authors propose a multi-agent system to monitor and manage container-based distributed systems. Their system allows users to observe and verify the quality and progress of the application over time, improving parameters such as QoS (Quality of Service). This study is unrelated to OMAS but has concepts similar to those we used in our approach. Furthermore, they encourage the connection between MAS and DevOps, using DevOps tools such as Docker. The main difference is that we do not use Docker to monitor MAS systems. Instead, we run our system in a container-based approach on Docker.

In the work [23], the authors use MAS in the IoT (Internet of Things) field since both share similarities (distributed devices, cooperation). They mainly use MAS in IoT to build large-scale and fault-tolerant systems. They propose a cloud-native MAP (Multi-Agent Platform), named cloneMAP, to use cloud-computing techniques to enable fault-tolerance and scalability. This approach is related to MAS rather than to OMAS. It is similar to our study using DevOps tools associated with MAS, such as Docker. Still, the study's primary goal is not related to openness and allowing agents to move between models. Also, similarly to [1], this study compares results directly to agent programming platforms, such as JADE (Java Agent DEvelopment Framework), which differs from our approach. In our approach, we keep the model similar to before the openness, making changes to insert the mechanisms to allow the model to communicate with our platform.

Table I summarizes the similar studies found, indicating if it is focused on the organization aspect of OMAS, if it uses some DevOps tools, and deals with more than one MAS platform. The DevOps aspect is important because it is a way to implement multiple parts of the systems that could run different languages and OS, making using all kinds of different MAS tools possible. The last aspect is how many MAS platforms/tools are being used. This aspect is essential to making the tool as embracing as possible.

TABLE I
SUMMARIZATION OF RELATED WORK.

Approach	Year	Org-related	DevOps Tools	Platform
[4]	2013	Yes	–	Single
[6]	2010	Yes	–	Single
[16]	1996	Yes	–	Single
[17]	2006	Yes	–	Single
[18]	2012	Yes	–	Single
[19]	2003	Yes	–	Single
[20]	2009	Yes	–	Single
[21]	2021	Yes	–	Single
[12]	2020	Yes	–	Single
[22]	2017	Other	–	Multiple
[5]	2018	Other	–	Single
[23]	2021	Other	Docker & Kubernetes	Single
[24]	2021	Other	Docker	Single
[1]	2018	Other	–	Single
Our	2022	Other	Docker	Multiple

From the 15 studies analyzed, the main focus of 9 of them is to deal with organizational problems that come with the openness from OMAS. Of the other 6, 3 are language dependent, not allowing the usage of MAS/OMAS tools that run in a different development/usage environment, as another tool that uses another language. Lastly, besides ours, only one study deals with more than one MAS platform. This study is only MAS, not OMAS-related, so it does not deal with the problem of transporting agents between models. Also, this study connects two MAS platforms that run in the same language (Java), not considering other tools that might use different languages.

In conclusion, our approach contributes to the state of the art of how the programmer will use the platforms. For example, some studies propose new approaches that require the programmer to rebuild their models according to the proposes. The programmers would not have to change their entire model in our architecture. Instead, they will add the structures needed to their models to communicate with our architecture, allowing an easier transition from a closed MAS to an open MAS. Another work already used container/cloud-based approaches related to the MAS system, showing that this approach is promising. Also, some studies have tested the communication between NetLogo and Jason agents.

III. The Proposed Approach

Our approach aims to develop an environment that facilitates the development of Open Multi-Agent Systems using Docker [25] as a basis, allowing the migration of agents between models that can run in heterogeneous scenarios. The structure allows code to run inside containers that contain the same operation structure where they were developed, avoiding problems like programs that run in the development stage but in the production stage have problems. In our approach, each image is generated based on a description file, called *DockerFile*, containing information on the structures needed in the container, such as the operating system, programming languages, and code to be executed. In addition, Docker has an image bank, the *Docker Hub Container Image Library*, which contains the images frequently used by developers, extending the approach's applicability.

The containers are responsible for executing each essential block of the architecture structure. Therefore, they can be easily replaced or added, making the architecture more modularized and adaptable to new scenarios not foreseen in the conception, making it more robust. If any part of the architecture has sensitive data, it is possible to create separate virtual networks; that way, each container has a partial view of the system, which leads to limited access to certain information. Finally, our approach can use online platforms such as Amazon's (AWS Docker) to run code in the cloud due to the entire structure being developed based on Docker, which enables high application portability, expanding the universe of applications in the architecture.

Figure 1 presents a general description of the proposed architecture in a block diagram format. In the **Docker-compose** block, we have the generation of images and services based on the *docker-compose.yaml* file and the *Dockerfile* of each

service. Each *Dockerfile* contains the sequence of instructions needed to assemble the image with all the requirements that each service needs. Then, based on the images, **Docker-compose** uses the parameters of each service (container name, exposed ports, network, volumes, command/file to be executed) and builds them. Finally, when we define all services on the **Docker-compose**, they can be executed using just one command.

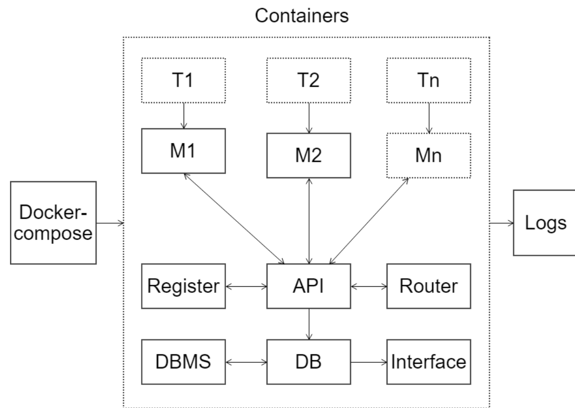

Fig. 1. Organization of the proposed architecture.

The $\mathbf{M}_1$, $\mathbf{M}_2$, ..., $\mathbf{M}_n$ blocks represent the containers responsible for running the models. The designer can define a parameter *auto run* in the main configuration file. When we set this parameter to *True*, the models are executed automatically when the container is mounted. If it is *False*, we must start the execution of the templates by a trigger container. $\mathbf{T}_1$, $\mathbf{T}_2$, ..., $\mathbf{T}_n$ are execution trigger containers, responsible for sending a message to the models to start their execution. We use these containers when necessary to implement a more robust logic to start executing the models. For example, they can run code that reads a sensor, wait for measurement, or run a particular model only when agents are assigned.

The **Application Programming Interface** (API) block is the container responsible for managing access to the database. This container acts as an intermediary when any system part must write, read, update, or delete information from the database. Currently, the API is implemented in Python, using the Flask framework [26]. All the methods are accessible via HTTP, using JSON (JavaScript Object Notation) format on every message, as a standard for APIs.

The **Database** (DB) block represents the container responsible for the database where we store, for each model, all agent arrival/departure information to be accessed by other containers that may need this information. It is important to note that we do not implement business rules in the DB. Also, this container is responsible for carrying out database operations (read, write, update, and delete). How containers must manipulate the information before or after involving the database is their responsibility, which will only pass the final command to this one.

The **Database Management System** (DBMS) block represents the container that facilitates access to the DB. It provides a Web interface (by default exposed to the host machine) that can import/export SQL (Structured Query Language) content/files and view the information in real-time. This container can help with debugging and making logs of the application. Also, the DBMS must be chosen according to the DB because they must be compatible.

The **Register** block is a container responsible for managing all agents on the platform. Every agent must have an identification to be used by the architecture, so the first insertion requests for new agents coming from any model containers need a unique identification. So, the Register is responsible for generating a unique identification for every agent then forwarding them back to the environments.

The **Router** block is the model responsible for receiving all agents that left a given model, analyzing and judging which model to send the agent. This step specifies the agent's entry and exit protocols from different containers/models. This block is an essential part of the proposal because models delegate to the Router the distribution task of the agents among the models. When the simulation environment does not share (or only partially) information about the world, the Router can handle several problems related to the absence of this information. Judgment can occur differently, such as analyzing the most promising agents, machine learning codes, and executing a new MAS model that retrains agents. In the simulation scenario, there are two judgment options that the Router can make of the list of agents to be processed: i) randomly choose the agent and the target model (random mode), and ii) process agents in a single queue, considering all models and randomly defines the destination model (general sequential mode).

The **Interface** block is a container that receives agent movement information through the system and generates reports exposed and formatted to view metrics of the architecture's execution. In the current implementation, this container has an Apache Webserver running PHP (Hypertext Preprocessor) code that reads all information from agents that have already gone through the Router and generates a report with all agents, attributes, and the path they took. The general form of the report is through a front-end (HTML (HyperText Markup Language), CSS (Cascading Style Sheet), and JS (JavaScript) code), which the host can access by exposing the port that Apache runs on port 80 (by default). Each tuple contains all essential attributes for every agent interaction so far. In the end, it presented the longest path taken by agents. Notably, a search option exists to filter the results by any of the columns. Also, it is possible to filter results by agent, model, and whether the tuple has been processed by the model/router. Finally, each agent already processed by the Gold Miners model (JaCaMo container) has a *.asl* file (AgentSpeak Language) containing relevant information to the model. So, on this interface, the user can check the *.asl* to see its content.

Finally, the **Logs** block is not a container but a module responsible for generating logs of the outputs of each container in the system. We can obtain several types of records from the simulation, such as regular docker logs (via docker log

command), DB tuples via SQL export, and prints on txt files (using terminal operators when running the container on docker-compose). In addition, some structures have particular log types, such as JaCaMo's default log generator (based on Java's logging API). These logs can assist in debugging tasks or generating data for execution analysis.

IV. The Platforms & Models Used

In the proposed simulation scenario, we built containers to support two different development tools for Multi-agent systems: NetLogo [27] and JaCaMo [28]. We chose two models from each tool's documentation: NetLogo's Open Sugarscape 2 Constant Growback [29] and JaCaMo's Gold Miners [30], with adaptations. We used three model containers, running two isolated copies of Open Sugarscape and one of Gold Miners. The main objective of this simulation scenario is to validate the approach feasibility, allowing agents to move between models freely, even though the two tools use different agent architectures. We have examples of how to adapt and use our functions on GitHub [31].

The Open Sugarscape model is a simulation used in artificial societies, simulating a population with limited resources where we represent each agent with an ant that, to survive, must move around the environment in search of sugar. Each ant is born with an amount of sugar (from 5 to 25), metabolism (from 1 to 4), and vision (from 1 to 6). Metabolism defines how much energy the ant loses when moving. If the ant's energy reaches zero, it dies and leaves the model. Vision determines how many distance positions the ant can see sugar from its initial position. Figure 2 illustrates the NetLogo interface simulating the original model.

In Figure 2, we can view the environment, input parameters, and some metrics. For example, it is possible to configure the number of agents at startup and the buttons to start, execute, and pause the code on the left. In the center, we can see the environment. The red dots represent the agents, while the others represent the sugar inserted into the environment, ranging from yellow to white, respectively, from more to less intensity. Finally, graphs represent simulation output metrics, such as population, metabolism, and vision averages.

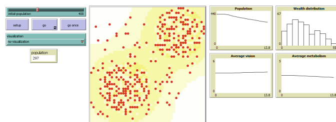

Fig. 2. Sample Sugar Scape Model in NetLogo IDE.

We made another adjustment when the agent returned to the simulation; instead of using his last information about food, we generated this parameter randomly, while the other parameters were the same as the previous simulation. We made that adjustment because of the way that the original simulation works. If the agent leaves the simulation when its food is zero, and we use that same information again when the agent returns, it would cause a loop because the agent would get to the simulation with zero food and die again.

The only two attributes added to the NetLogo agents are *agent_id* and *historic*. We used *agent_id* to check the agent's unique id for the architecture, while the regular id is used just for the NetLogo's simulation. The attribute *historic* shows the path of what models the agents went through.

The only dependency required is Py NetLogo's extension to adapt any NetLogo model to our platform. We need this extension to send/receive information from the API through Python code. We have small functions to do so, and all the programmer has to do is include those functions and use them whenever the model needs to send/receive information about the agents.

Figure 3 illustrates the JaCaMo interface simulating the Gold Miners model. This model simulates a set of agents representing miners whose role is to navigate the environment. For example, when an agent finds a gold node, it stops his current objective, gets the gold, and brings it back to the central deposit. In the Figure, it is possible to see the representation of the agents (blue dots), the deposit (box with X in the center), the environmental barriers (positions not accessible to the agents, black boxes), the gold nodes (yellow boxes), the positions already visited by agents (white boxes) as well as those not accessed (gray boxes).

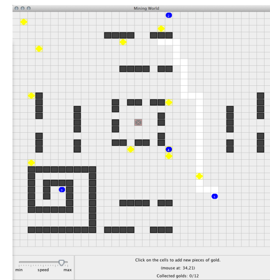

Fig. 3. Gold Miners running on JaCaMo GUI (Graphical User Interface).

In the current adaptation of the original model, we use only one miner agent, along with the architecture's agents (*check_new_agents* and *killer_agent*) and the leader agent. So, when the agent joins the simulation, it takes control of the miner agent. This agent navigates through the map, brings back his old information (about previous simulations), navigates through the map, gets gold, brings it back to the depot, and then leaves the simulation (sends a message to the *killer_agent* to be removed from the simulation). Finally, when the agent leaves the simulation, the model checks the next agent that will join the simulation and takes control of the miner agent so the simulation can continue its execution.

Some adaptations are needed to adapt any JaCaMo model to our platform. First, if the project is not already running via Gradle (which is the default option), we need to adapt the model to do so. The steps required to achieve this goal are presented in JaCaMo's GitHub documentation. After, we need to include two libraries on the build Gradle file: JSON-simple (to deal with JSON-format information) and HTTP client (to deal with HTTP requests). The last step is to include the killer/checker agents (with their ASL files) who deal with the

API. With these steps done, the model is ready to communicate with the API. Now, the programmer can use the system's functions to send and receive agents between the API and the model.

In the simulation scenario, $\mathbf{M}_1$ and $\mathbf{M}_2$ containers are running simultaneously two replicas of the Sugar Scape NetLogo model, and the $\mathbf{M}_3$ container is running the Gold Miners model, on JaCaMo. All three models run indefinitely and use the *auto-run* option when no trigger container starts the simulation. When an agent dies, the current model is responsible for sending it to the **Router** (and then to **DB** via **API**); it will go through a process that will decide which models the agent will return.

At the beginning of the execution, instead of creating the agents directly by the model, the models $\mathbf{M}_1$ and $\mathbf{M}_2$ ask the **Register** container to create the agents so that they receive an identification used in the Open Multi-Agent System. In the current implementation of the simulation scenario, only the NetLogo models ask for new agents. The JaCaMo model is receiving and sending agents but not creating them on the initialization, even though it is possible.

After the initial step of creating and registering all agents, the $\mathbf{M}_1$, $\mathbf{M}_2$ and $\mathbf{M}_3$ models continue their execution, along with the **Router** processing. Whenever an agent leaves any models, it goes to the **DB** container (via API). Once new information is on the **DB**, the **Router** knows that there are new agents to be processed, so it reads the agents and sends them to a model according to the routing type (by default, randomly). Both **Log** or **Interface** containers can access the flow of this information.

It is essential to point out that most adaptations are not substantial, mischaracterizing the original models to be adapted to the architecture; instead, they are due because these models need to be prepared to receive agents from outside, to be able to do so. We have provided a template of the trigger functions to send and receive information to the architecture. The users must add it to their model, adapt to what information they want to send or receive and use the triggers when they want the model to send or receive agents. We emphasized that this feature is important because we aim to reduce the complexity of the code adaptation. On the documentation, the user can find all the information needed.

Also, those adaptations show that the architecture allows completely different agent architectures to communicate. For example, NetLogo programming uses Logo language, while JaCaMo uses ASL on BDI (Belief, Desire, Intention) concepts. Though heterogeneous models, the agent carries all its information while moving in the architecture. Thus, it is possible to adapt information when we create an agent on each model because we included/excluded from the models in real-time.

Those simulations were essential to check the strengths of the architecture. With these tests, the platform allowed two MAS models, coming from the original documentation of each platform (NetLogo and JaCaMo), becoming from a closed to an open MAS execution with few extra steps. Furthermore, we could familiarly implement the original models. Instead, we had to insert the codes and the agents necessary to communicate with our platform. Finally, it is essential to point out that both platforms have access to all information from the agent. For example, JaCaMo's model could access any attribute from NetLogo's agent (sugar, metabolism, and vision) to make something useful in the simulation. The opposite is true: NetLogo has access to the *.asl* file of every agent, making it possible to use some information, like a belief, for something useful in the simulation. The data exchange and the openness with few steps are two key contributions of our work.

We have two videos; the first one[1] shows a quick run on the models using the whole architecture on Docker, and the last one[2] presents how JaCaMo and other platforms can communicate with the Docker architecture, making it possible to run applications with GUI.

In the scenarios, after downloading the platform from GitHub, the programmer starts the simulation with the command *docker-compose up -d*. This command will tell Docker to download and build all images and containers necessary, according to the *docker-compose.yml* file. After, all container logs are accessible via Docker Desktop or Docker CLI (Command Line Interface). In the API container, it is possible to check all the requests that the API receives and processes, while in the JaCaMo container, the user can check all alive agents, minds, artifacts, and so on. Besides the Docker logs, the DBMS and Interface blocks are accessible via the browser (address on docker-file), allowing users to access all agents' navigation through the models and their parameter values.

V. Results

A. Running Locally

In local mode, the complete architecture is assembled and executed within the same machine *host*, through the file *docker-compose-local*. All Docker services run locally and share the resources they need across the isolated network. We used this mode in the scenarios described above.

B. Running on Cloud – Remote Scenario

We used the Oracle Cloud [32] for the remote and hybrid scenarios. Besides the paid modality, the user can test the complete platform for 30 days free of charge or part of the services contains the *always free* modality, where we can use these resources indefinitely. In the remote mode, similarly to the local way, all services are mounted/executed on the same *host* machine. However, instead of the user running it on his machine, he can run it on a machine in the *cloud*.

In the tests carried out using the Oracle Cloud service, the machines provided by the platform were used. To do so, install Docker on the machine, clone the project from GitHub, and use the *docker-compose-local-arm64* file. This file slightly adapts some container's images because the original images were developed to use x86/x64 architectures, while the machines used in the Oracle Cloud free trial use arm64 machines

[1]https://streamable.com/rug2mr
[2]https://streamable.com/3rct7p

with different instructions. The change is simple, just indicating the corresponding file with the machine's architecture (x86/x64 or arm64). The platform already has files ready to run on both architectures.

C. Running on Cloud – Hybrid Scenario

Finally, we developed a hybrid test scenario where some architecture services ran on the local machine and others on the *cloud* machine. More specifically, the containers for the 2 NetLogo templates ran locally, and the rest of the architecture ran in *cloud*. This division of parts occurs for two reasons: (i) some structures need to be on the same network, such as the JaCaMo model and the *web* Interface, so that we can access ASL files, and (ii) the NetLogo models, in the cited simulation scenario, tend to spend more computational resources.

Just like the remote model, to run this scenario, install Docker on the machine, clone the project from GitHub, and use the respective machine files *host* (*docker-compose-hybrid-local*) and the remote server (*docker-compose-hybrid-server*). The same adaptations of the images are necessary since the remote Oracle machine can have a processor with arm64 instructions. For the two parts of the architecture to communicate, it is required to indicate that the remote machine needs to expose the communication port with the API, as this is where the models will send/receive agents. The only exposure of the machine is the communication port with the API (port 5000 by default). Furthermore, it is necessary to indicate the IP of the remote machine so that the models know how to communicate with the API. We indicated this in the *docker-compose-hybrid-local* file, which runs on the local machine, through the environment variable *host*.

When executing the two separate parts, the remote structure was assembled first, and then the local part was executed. This ordering is because the local structure depends on the rest of the architecture being ready for use, thus guaranteeing correct functioning. Since the local machine used on the tests had a good amount of resources, all the model's containers were executed on the local machine, while the other containers from the architecture were executed on the cloud machine. The communication between the models and the architecture was performed through the API container by exposing the port and IP from the container to the Internet.

The constructed scenarios show how we can extend the architecture differently, such as migrating from local systems to multi-agent systems that run on *cloud*. In addition, the use of architecture in Docker allows these services to be distributed, run on several different machines, and scalable in case the models run robust systems. Adaptations may occur according to the capacity of the available machines. Finally, it is also important to point out that when taking the platform to *cloud*, the resources used to execute the architecture are those of the remote machine, not the local one, making it possible to run part of the architecture on computers with fewer resources.

Finally, another possibility for extending test scenarios is using models outside Docker. As previously mentioned, once the structure is assembled and executed in Docker, it is possible to expose container ports to the *host* machine. In this way, it is possible, for example, to run the Gold Miners model (or any other) on the local machine and still make the model participate in a structure built in the architecture. To do so, expose the port of the API container and indicate this new *host* to the files that communicate with the architecture (in this case, the files *my_create_ag* and *my_delete_ag*). Both files already have a *using_docker* parameter, where the value *true* defines that the use of the files must follow the normal flow in integration with the platform, while the value *false* defines that the *host* ID is different from the default nomenclature. This way of executing the architecture makes it possible, for example, to run models with graphical interfaces, still using the architecture for the concept of openness.

Regarding architecture drawbacks, two things can be pointed out: overhead and GUI. While Docker has less overhead than VMs, it's not zero. Therefore, the Docker engine may use some resources compared to native models. However, in our tests on a resource-scarce cloud machine, the impact was minimal. Secondly, we designed Docker containers for command-line use, even though GUI options exist. Simulations requiring GUI must be run locally, with communication to the architecture via API port exposure, as demonstrated in one of our simulation scenarios. Refer to our documentation for an example.

VI. Conclusions

This paper presented a proposal of architecture to assist the development of Open Multi-Agent Systems. Similar work to this proposal was summarized and discussed. Then, we introduced the proposed approach, methodology, and implementation details. Finally, we showed the results obtained, emphasizing the case of the study developed in the architecture prototype, which is of great importance to verify the feasibility of the implementation. Our approach presents an environment that facilitates the development of Open Multi-Agent Systems using Docker.

This architecture allows the migration of agents between models that can run in heterogeneous scenarios. Furthermore, the structure provides code to run inside containers that contain the same operation structure where they were developed, avoiding problems like programs that run in the development stage but have issues in the production stage. In addition, the architecture acts as the union step between models. Docker allows models to be executed in different scenarios on various platforms, using distinct development languages that run on multiple operating systems. In the presented simulation scenarios, we used containers running NetLogo's Open Sugarscape 2 Constant Growback and JaCaMo's Gold Miners with minor adaptations to validate the approach feasibility.

The prototype presented allowed agents to move freely between models, sharing all agent information with respective models and reducing the complexity of the code adaptation, demonstrating that it is sufficient but straightforward to understand the proposed approach better. The migration of agents between models occurs at runtime. The motivation

for this migration of an agent from one system to another can be different, usually of the developer's choice, such as execution failures, self-will, or some trigger. In addition, we can deal with conflicts of interest between the new agents when designed to work outside of that model.

In future work, we want to explore new triggers that make agents switch models, such as geographic boundaries and parallel models. Geographic boundaries are models where, when the agent reaches the edge of the environment, it moves to the other model. Parallel models are models where the same agent participates in more than one model simultaneously, but each model evolves particular attributes of the agent. Finally, the architecture implementation supports two widely used agent platforms, NetLogo and JaCaMo. However, we want to go further and support other agent platforms, such as JADE (Java-based) or Mesa (Python-based).

REFERENCES

[1] A. Perles, F. Crasnier, and J.-P. Georgé, "Amak-a framework for developing robust and open adaptive multi-agent systems," in *International Conference on Practical Applications of Agents and Multi-Agent Systems*. Springer, 2018, pp. 468–479, doi: 10.1007/10719619_16.

[2] V. R. Lesser, "Cooperative multiagent systems: A personal view of the state of the art," *IEEE Transactions on knowledge and data engineering*, vol. 11, no. 1, pp. 133–142, 1999, doi: 10.1109/69.755622.

[3] M. Woolridge and M. J. Wooldridge, *Introduction to multiagent systems*. EUA: John Wiley & Sons, Inc., 2001, doi: 10.5555/1695886.

[4] W. Jamroga, A. Meski, and M. Szreter, "Modularity and openness in modeling multi-agent systems," *Electronic Proceedings in Theoretical Computer Science*, vol. 119, p. 224–239, Jul 2013, doi: 10.4204/eptcs.119.19.

[5] D. M. Uez, "Open aeolus: um método para especificação de sistemas multiagentes abertos," PhD Thesis, PhD on Automation and Systems Engineering, Federal University of Santa Catarina, Centro Tecnológico, Florianópolis, 2018, available in: https://repositorio.ufsc.br/handle/123456789/205584.

[6] F. Dalpiaz, A. K. Chopra, P. Giorgini, and J. Mylopoulos, "Adaptation in open systems: Giving interaction its rightful place," in *Proc. of the Conceptual Modeling – ER 2010*, J. Parsons, M. Saeki, P. Shoval, C. Woo, and Y. Wand, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 31–45, doi: 10.1007/978-3-642-16373-9_3.

[7] T. D. Huynh, N. R. Jennings, and N. Shadbolt, "Developing an integrated trust and reputation model for open multi-agent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 13, p. 119–154, 2004, doi: 10.1007/s10458-005-6825-4.

[8] S. Kaffille and G. Wirtz, "Modeling the static aspects of trust for open mas," in *2006 International Conference on Computational Intelligence for Modelling Control and Automation and International Conference on Intelligent Agents Web Technologies and International Commerce (CIMCA'06)*, IEEE. Australia: IEEE, 2006, pp. 186–186, doi: 10.1109/CIMCA.2006.150.

[9] R. M. van Eijk, F. S. de Boer, W. Van Der Hoek, and J.-J. C. Meyer, "Open multi-agent systems: Agent communication and integration," in *Proc. of the International Workshop on Agent Theories, Architectures, and Languages*. Orlando, Florida, USA: Springer, 1999, pp. 218–232, doi: 10.1007/10719619_16.

[10] J. M. Hendrickx and S. Martin, "Open multi-agent systems: Gossiping with deterministic arrivals and departures," in *Proc. of the 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. Monticello, IL, USA: IEEE, 2016, pp. 1094–1101, doi: 10.1109/ALLERTON.2016.7852357.

[11] M. Franceschelli and P. Frasca, "Proportional dynamic consensus in open multi-agent systems," in *Proc. of the IEEE Conference on Decision and Control (CDC)*. Miami, FL, USA: IEEE, 2018, pp. 900–905, doi: 10.1109/CDC.2018.8619639.

[12] Z. Houhamdi and B. Athamena, "Collaborative team construction in open multi-agents system," in *Proc. of the 21st International Arab Conference on Information Technology (ACIT)*. Giza, Egypt: IEEE, 2020, pp. 1–7, doi: 10.1109/ACIT50332.2020.9300116.

[13] S. Noh and J. Park, "System design for automation in multi-agent-based manufacturing systems," in *Proc. of 20th International Conference on Control, Automation and Systems (ICCAS)*. Busan, Korea (South): IEEE, 2020, pp. 986–990, doi: 10.23919/ICCAS50221.2020.9268357.

[14] W. Jiang, Y. Chen, and T. Charalambous, "Consensus of general linear multi-agent systems with heterogeneous input and communication delays," *IEEE Control Systems Letters*, vol. 5, no. 3, pp. 851–856, 2021, doi: 10.1109/LCSYS.2020.3006452.

[15] M. Franceschelli and P. Frasca, "Stability of open multiagent systems and applications to dynamic consensus," *IEEE Transactions on Automatic Control*, vol. 66, no. 5, pp. 2326–2331, 2021, doi: 10.1109/TAC.2020.3009364.

[16] Y. Demazeau and A. R. Costa, "Populations and organizations in open multi-agent systems," in *Proceedings of the 1st National Symposium on Parallel and Distributed AI (PDAI'96)*. India: University of Hyderabad, 1996, pp. 1–13.

[17] J. Gonzalez-Palacios and M. Luck, "Towards compliance of agents in open multi-agent systems," in *Proc. of the International Workshop on Software Engineering for Large-Scale Multi-agent Systems*. Shanghai, China: Springer, 2006, pp. 132–147, doi: 10.1007/978-3-540-73131-3_8.

[18] A. Artikis, "Dynamic specification of open agent systems," *Journal of Logic and Computation*, vol. 22, no. 6, pp. 1301–1334, 07 2011, doi: 10.1093/logcom/exr018.

[19] S. Paurobally, J. Cunningham, and N. R. Jennings, "Ensuring consistency in the joint beliefs of interacting agents," in *Proc. of 2nd International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*. Melbourne, Australia: ACM, 2003, pp. 662–669, doi: 10.1145/860575.860682.

[20] M. P. Singh and A. K. Chopra, "Programming multiagent systems without programming agents," in *Proc. of the International Workshop on Programming Multi-Agent Systems*. Budapest, Hungary: Springer, 2009, pp. 1–14, doi: 10.1007/978-3-642-14843-9_1.

[21] S. Hattab and W. Lejouad Chaari, "A generic model for representing openness in multi-agent systems," *The Knowledge Engineering Review*, vol. 36, p. e3, 2021, doi: 10.1017/S0269888920000429.

[22] W. A. L. Ramirez and M. Fasli, "Integrating netlogo and jason: a disaster-rescue simulation," in *2017 9th Computer Science and Electronic Engineering (CEEC)*. IEEE, 2017, pp. 213–218, doi: 10.1109/CEEC.2017.8101627.

[23] S. Dähling, L. Razik, and A. Monti, "Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing," *Autonomous Agents and Multi-Agent Systems*, vol. 35, no. 1, pp. 1–27, 2021, doi: 10.1007/s10458-020-09489-0.

[24] V. Pfeifer, W. F. Passini, W. F. Dorante, I. R. Guilherme, and F. J. Affonso, "A multi-agent approach to monitor and manage container-based distributed systems," *IEEE Latin America Transactions*, vol. 20, no. 1, pp. 82–91, 2021, doi: 10.1109/TLA.2022.9662176.

[25] J. Turnbull, *The Docker Book: Containerization Is the New Virtualization*. Melbourne, Australia: James Turnbull, 2014.

[26] M. Grinberg, *Flask web development: developing web applications with python*. O'Reilly Media, Inc., 2018, doi: 10.5555/2621997.

[27] S. Tisue and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in *International conference on complex systems*, vol. 21. Citeseer, 2004, pp. 16–21.

[28] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with jacamo," *Science of Computer Programming*, vol. 78, no. 6, pp. 747–761, 2013, doi: 10.1016/j.scico.2011.10.004.

[29] J. M. Epstein and R. L. Axtell, *Growing artificial societies: Social Science from the Bottom Up*, ser. Complex Adaptive Systems. Cambridge, MA: Bradford Books, Oct. 1996.

[30] R. H. Bordini, J. F. Hübner, and D. M. Tralamazza, "Using jason to implement a team of gold miners," in *International Workshop on Computational Logic in Multi-Agent Systems*. Springer, 2006, pp. 304–313, doi: 10.1007/978-3-540-69619-3_18.

[31] G. L. de Lima and M. S. de Aguiar, "Architecture's github page," 2022, available in: https://github.com/GustavoLLima/open_mas_docker_opt. Accessed in 26 oct. 2022.

[32] M. T. Jakóbczyk, *Practical Oracle Cloud Infrastructure*. Springer, 2020.